Linux Basics Course

Flavio Copes

Updated August 8, 2026

Contents

Learn how Ubuntu Linux is organized and administer packages, users, configuration, services, logs, disks, memory, and networking.

What you'll learn: Inspect and maintain an Ubuntu system, install software safely, manage services, read logs, and diagnose common resource and network problems.

Ubuntu foundations

Understand Linux, Ubuntu releases, the kernel, distributions, and system identity.

What Linux is

Understand the difference between the Linux kernel, a complete operating system, and the many distributions built around it.

Linux is the kernel at the center of many operating systems. It manages processes, memory, devices, filesystems, and communication with hardware.

Started by Linus Torvalds in 1991, it is open source and developed by volunteers and companies together, with no single vendor controlling it. It powers most of the servers on the internet, and Android runs on a modified version of it.

Kernel versus distribution

A kernel alone is not something you can install and use. It has no shell, no editor, no way to install software.

A **distribution** combines that kernel with system tools, libraries, a package manager, and a maintained software collection. Debian, Fedora, and Ubuntu are three well-known distributions, and hundreds more exist. They all run the Linux kernel; they differ in the software wrapped around it and in how it is maintained.

Ubuntu is one Linux distribution. This course uses Ubuntu because it is common on servers and has extensive documentation.

Checking what you are running

On any Linux machine, ask the kernel to identify itself:

```
uname -s
# Linux
```

Then ask which distribution wraps it:

```
grep PRETTY_NAME /etc/os-release
# PRETTY_NAME="Ubuntu 24.04.3 LTS"
```

Two answers, two layers: the kernel name is the same everywhere, the distribution name tells you which tooling to expect.

Why the distinction matters

Instructions are usually distribution-specific, not ”Linux-specific”. Ubuntu and Debian install software with `apt`. Fedora uses `dnf`. Follow a Fedora tutorial on an Ubuntu server and the first install command fails:

```
dnf install nginx
# Command 'dnf' not found
```

Nothing is broken. You are reading instructions for a different distribution. When a guide's package commands fail this way, check `/etc/os-release` and find the equivalent instructions for your distribution — on Ubuntu, that usually means swapping in `apt`.

Ubuntu releases and LTS

Choose a supported Ubuntu release and understand why long-term support is the safe default for servers and learning environments.

Ubuntu publishes a new release every six months. Every two years, the April release is a **Long Term Support** release, usually called LTS.

For a server, use a supported LTS release unless you have a clear reason not to. You get a longer standard-support window, fewer major upgrades, and documentation that remains useful for longer.

Interim releases contain newer software, but their support window is much shorter. They make sense when you need a specific kernel, driver, or package version and you are ready to upgrade the machine again soon.

Check the release you are running

Use the release file instead of guessing from a screenshot or tutorial:

```
cat /etc/os-release
```

Look for `VERSION_ID` and `VERSION_CODENAME`:

```
VERSION_ID="24.04"
VERSION_CODENAME=noble
```

The codename appears in repository names and support documentation. Commands written for another Ubuntu release may install a different package version or use a configuration layout your system does not have.

Package updates are not release upgrades

`sudo apt upgrade` updates packages **inside the current Ubuntu release**. It does not move a 24.04 system to a later Ubuntu release.

A release upgrade uses `do-release-upgrade`. It is an operational change: read the new release notes, update the current system, confirm free disk space, back up important data, and plan downtime before starting.

LTS systems normally upgrade to the next LTS in sequence. Do not assume you can skip several releases.

Check support before following old instructions

An old tutorial can look correct while pointing at an unsupported repository. Before copying commands, confirm:

- your Ubuntu release is still supported
- the instructions name your release or codename
- the package source is still maintained
- you have a recovery path if the change fails

Try this: run `cat /etc/os-release` on one Linux machine. Record its version, codename, and whether it is an LTS release.

Kernel and user space

Separate the privileged kernel from the programs, services, shells, and applications that run above it.

The kernel controls access to hardware and shared resources. Programs normally run in **user space** with limited privileges.

This split exists for protection. If every program could write to the disk hardware directly, one buggy program could corrupt any file, read another program's memory, or take the network down. Instead, the kernel is the only code with full control, and everything else — your shell, your editor, the web server — must go through it.

System calls: the controlled door

A **system call** is the controlled path a program uses to ask the kernel for work such as opening a file or creating a process.

When you run `cat /etc/hostname`, the `cat` program does not touch the disk. It asks the kernel to open the file, asks it to read the bytes, and asks it to write them to your terminal. Three requests, all answered by the kernel, all checked against permissions before anything happens.

Seeing both sides

The kernel itself shows up in a process listing as bracketed names:

```
ps -e | head -5
#     PID TTY          TIME CMD
#       1 ?        00:00:04 systemd
#       2 ?        00:00:00 kthreadd
```

`systemd` is the first user space process. Entries like `kthreadd` are kernel threads. Everything you will start on this machine joins the user space side of that list.

One detail that surprises people: even `root` runs in user space. Root is a powerful *account*, and the kernel grants its requests broadly, but a root shell still works through system calls like every other program. Root is not ”inside” the kernel.

Why this boundary helps you

This boundary is one reason a broken application does not normally crash the entire operating system. When a program misbehaves — reads memory it does not own, for example — the kernel kills that one process and everything else keeps running.

That gives you a diagnostic rule. One application crashing repeatedly is a user space problem: look at that program, its configuration, its logs. The whole machine freezing or rebooting on its own points at the kernel, a driver, or hardware — a much rarer and more serious class of failure.

Identify the system

Read the operating-system release, kernel, hostname, architecture, current user, and machine uptime before changing anything.

Before changing an unfamiliar machine, identify what you are connected to. A hostname in the prompt is not enough: two servers can have similar names and very different operating-system versions.

Start with the operating-system release:

```
cat /etc/os-release
```

Then collect the kernel, architecture, hostname, current account, and uptime:

```
uname -r
uname -m
hostnamectl
whoami
uptime
```

These commands answer different questions:

- `/etc/os-release` identifies the distribution and release.
- `uname -r` shows the running kernel.
- `uname -m` shows the hardware architecture, such as `x86_64` or `aarch64`.
- `hostnamectl` shows the configured hostname and system details.
- `whoami` confirms which account will run the next command.
- `uptime` shows how long the machine has been running and its load averages.

Why this matters

A package name can differ between releases. A downloaded binary built for `x86_64` will not run on an ARM server. A reboot may have loaded a different kernel than the one you expected.

Also verify the target before a destructive or administrative command:

```
printf 'host=%s user=%s\n' "$(hostname)" "$(whoami)"
```

Do not include secrets, private IP addresses, or customer data in a public bug report. Include only the system facts needed to reproduce the problem.

Create a small diagnostic block

When asking for help, a useful report might contain:

```
Ubuntu: 24.04 LTS
Kernel: 6.x
Architecture: x86_64
Service: notes-api.service
Started failing: after package update
```

The exact versions will vary. The important part is replacing assumptions with observed facts.

Try this: identify one Linux system, then explain which command proves the distribution and which proves the architecture.

Check your understanding: Ubuntu foundations

Check kernels, distributions, Ubuntu releases, LTS support, user space, architecture, hostnames, and system identity.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Filesystem and configuration

Learn where programs, configuration, logs, user data, devices, and temporary files live.

The Linux filesystem tree

See one directory tree rooted at slash instead of separate drive letters and connect devices through mount points.

Linux has no drive letters. There is one directory tree, and every file on the machine lives somewhere inside it.

Every absolute path begins at `/`, the filesystem root. Read a path like `/home/ada/notes.txt` as a walk: start at the root, enter `home`, enter `ada`, arrive at `notes.txt`. User home folders normally live under `/home`.

List the top of the tree:

```
ls /
# bin  boot  dev  etc  home  lib  media  mnt  opt
# proc  root  run  sbin  srv  tmp  usr  var
```

Those first-level directories are the fixed landmarks of every Linux system. The next lesson covers what the important ones contain.

Where do other disks go?

On Windows, a second disk becomes `D:`. On Linux, a separate disk still appears somewhere inside this one tree after it is **mounted**. The mount point is an ordinary directory used as the entry to that filesystem.

You can ask which device backs any directory:

```
findmnt /
# TARGET SOURCE         FSTYPE OPTIONS
# /      /dev/vda1      ext4   rw,relatime
```

Here the root of the tree is served by the disk `/dev/vda1`. Mount a second disk at `/mnt/data` and everything under that path is stored on the second disk, while the rest of the tree stays on the first. Programs never notice — they just use paths.

The classic mount-point mistake

A directory used as a mount point is a normal directory, and that enables a confusing failure. If the disk is *not* mounted and you copy files into `/mnt/data`, those files land on the root disk. When the disk mounts there later, it covers them. The files look gone, and disk usage numbers stop adding up. Unmount, and they reappear.

So before writing into a mount point, check it with `findmnt` — if it prints nothing for that path, nothing is mounted there.

Complete the Shell Commands Course first if absolute paths, relative paths, and `..` are new to you.

Important system directories

Recognize the small set of directories you will inspect repeatedly while administering Ubuntu.

A handful of top-level directories account for almost everything you will do on a server. Learn their jobs and the rest of the tree stops looking mysterious.

`/etc` contains system-wide configuration. `/var` contains changing data such as logs, caches, and application state. `/usr` contains installed programs and shared data.

A useful rule of thumb: configuration lives in `/etc`, data the system writes lives in `/var`, and the programs themselves live in `/usr`.

You can peek at each one right now:

```
cat /etc/hostname
# webserver-01

ls /var/log | head -3
# apt
# auth.log
# syslog

ls /usr/bin | wc -l
# 1147
```

One machine name in a plain text file, a directory of logs, and over a thousand installed commands — each directory doing exactly its job.

Home directories and the rest

`/home` contains regular user homes, one folder per account. `/root` is the root user home — note that it sits outside `/home`, and regular users cannot even list it:

```
ls /root
# ls: cannot open directory '/root': Permission denied
```

That error is correct behavior, not a problem.

`/tmp` stores temporary files, and its contents do not survive long: anything there can disappear on reboot. `/dev` exposes device interfaces as files — `/dev/null`, the writable black hole you will meet in shell tutorials, lives there.

The mistake to avoid

When you need to change how an installed program behaves, resist editing its files under `/usr`. Those files belong to the package manager, and the next upgrade overwrites them, silently reverting your change. The supported knobs are in `/etc` — that separation is the whole reason the two directories exist.

Do not memorize every directory. Learn the purpose of the major ones and use package documentation to find specific files. When a tutorial mentions a path, you should now be able to guess what kind of file it is from the first component alone.

Configuration and generated files

Edit the source configuration a program owns instead of changing generated output that will be overwritten.

System-wide configuration commonly lives under `/etc`. A service may use one main file, a directory of smaller fragments, environment files, or configuration generated by another tool.

Do not edit the first file that contains the value you want. First find which file is the **source of truth**.

For example, a file may begin with a warning like this:

```
# This file is managed automatically. Changes will be overwritten.
```

Editing it may appear to work until the next package update, deployment, or configuration-generation command removes your change.

Inspect ownership before editing

Check which package owns a file:

```
dpkg -S /path/to/file
```

Read the service documentation and inspect its unit definition:

```
systemctl cat notes-api.service
```

This can reveal environment files, command-line flags, and drop-in configuration. For systemd units, prefer a supported drop-in created with `systemctl edit` over modifying a package-owned unit file directly.

Use a safe change sequence

1. Save the current working file or record its version-control state.
2. Change one thing.
3. Run the application's configuration check.
4. Reload or restart only when the check succeeds.
5. Verify the service and its real behavior.

Nginx, for example, can validate its complete configuration before a reload:

```
sudo nginx -t
sudo systemctl reload nginx
systemctl status nginx --no-pager
```

Do not reload after a failed validation. Restore the previous file if the new configuration does not behave as expected.

Package updates can ask about local changes

When you modify a package-managed configuration file, an update may ask whether to keep your version or install the maintainer's version. Read the diff. Neither choice is automatically correct.

My advice is to keep custom configuration small and placed in supported fragment directories. This makes upgrades and review much easier.

Try this: choose one service on a test machine. Find its unit file, configuration path, validation command, and reload command without changing anything.

Logs, temporary files, and mounts

Know where changing system data lives and inspect it without filling the disk or deleting active files blindly.

Linux stores changing system data in several places. Knowing what each directory represents helps you investigate a full disk without deleting active data blindly.

Traditional text logs usually live under `/var/log`. Services managed by systemd often send structured messages to the journal instead:

```
sudo journalctl -u nginx --since today
```

Some applications use both. Check the service configuration before assuming a log is missing.

Temporary directories are not permanent storage

`/tmp` contains short-lived temporary data. `/var/tmp` is intended to survive longer, often across reboots, but it is still temporary.

Cleanup policies vary. A scheduled service can remove old files while your application still expects them. Store durable uploads and application state in an intentional data directory, not in `/tmp`.

Before deleting a temporary file, find which process uses it:

```
sudo lsof /tmp/suspicious-file
```

A mount changes what a path points to

A mounted filesystem appears at an ordinary directory. Use `findmnt` to see the real relationship:

```
findmnt
findmnt /var/lib
```

`df -h` reports capacity by filesystem:

```
df -h
```

`du` adds the visible files under a directory:

```
sudo du -xhd1 /var | sort -h
```

The `-x` option stays on one filesystem. This prevents the command from walking through other mounts by accident.

When `df` and `du` disagree

A process can keep a deleted file open. The directory entry disappears, so `du` no longer sees it, but the filesystem keeps the data until the process closes the file.

Find deleted open files with:

```
sudo lsof +L1
```

Restart or reload the owning service only after you understand the impact. Do not delete random files from `/var/lib`; that directory often contains live databases and package state.

Try this: use `findmnt`, `df -h`, and `du -xhd1` on a test machine. Explain why each command answers a different question.

Check your understanding: filesystem and configuration

Check root paths, mount points, system directories, configuration sources, logs, temporary data, and disk inspection.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Software and users

Manage APT packages, updates, users, groups, sudo access, and ownership boundaries.

How APT works

Separate repository metadata, available packages, installed package files, and the updates proposed for the system.

Ubuntu repositories contain package files and signed metadata that describes their names, versions, dependencies, and checksums.

APT works with that metadata. It decides which packages and dependencies are needed, downloads them, verifies them, and asks `dpkg` to install the package files.

`apt update` updates the index

Your machine keeps a local package index. Refresh it with:

```
sudo apt update
```

This command does **not** upgrade installed software. It downloads current repository metadata so later commands know which versions are available.

Ubuntu 24.04 stores the default repository definition in `/etc/apt/sources.list.d/ubuntu.sources`. Older releases commonly use `/etc/apt/sources.list`. Do not replace repository files with examples written for a different release.

Inspect installed and candidate versions

Use `apt policy` to compare the installed version with the version APT would choose:

```
apt policy nginx
```

A typical result identifies an `Installed` version, a `Candidate`, and the repositories offering each version.

List pending upgrades with:

```
apt list --upgradable
```

Read this list before changing an important server. An update may affect a public service, a database client, or the kernel.

Dependencies are part of the change

Installing one package can add several dependencies. Removing one can make dependencies eligible for later cleanup.

APT prints a transaction summary before it acts. Read which packages will be installed, upgraded, or removed. Stop if the proposed change is broader than expected.

For scripts, prefer `apt-get`; the `apt` command is designed primarily for interactive use. In either case, avoid automatic `-y` confirmation until you trust and understand the transaction.

Try this: run `apt policy` for one installed package. Identify its installed version, candidate version, and repository without changing the system.

Install and remove packages

Search, inspect, install, remove, and clean packages without copying an unknown installation script into a root shell.

Search for a package and inspect it before installing:

```
apt search '^tree$'
apt show tree
```

The description, repository, download size, and dependencies help you confirm that you found the intended package.

Install it with:

```
sudo apt install tree
```

Read the transaction summary before confirming. If APT proposes removing unrelated packages or changing a large part of the system, stop and investigate.

Remove, purge, and autoremove

These commands have different effects:

```
sudo apt remove tree
sudo apt purge tree
sudo apt autoremove
```

`remove` deletes the installed program but can keep package-managed configuration. `purge` removes that configuration too. Neither command promises to remove data the application created in your home directory or under `/var/lib`.

`autoremove` proposes dependencies that APT considers no longer required. Review the list. Do not accept a surprising removal just because the command describes the packages as automatic.

Prefer known package sources

Use Ubuntu repositories when they contain a suitable version. A command such as this deserves extra scrutiny:

```
curl https://vendor.invalid/install.sh | sudo sh
```

It downloads changing code and executes it as root without giving you a review point. If you need a third-party repository, use the vendor's current official instructions, understand its signing key and

repository scope, and record how to remove it later.

Verify what was installed

Check the binary and package record:

```
command -v tree
dpkg -L tree | less
apt policy tree
```

After removal, check whether the service, configuration, and application data you intended to remove are truly gone.

Try this: inspect one small package, write down every package APT proposes adding, install it, verify the binary, then remove it on a test machine.

Update the system safely

Apply security and package updates with a recovery plan and know when a service or reboot is still required.

Package updates fix security issues and bugs, but they also change a running system. Treat an important server update as a small deployment.

Prepare before changing packages

Confirm you are on the intended host and check basic capacity:

```
hostname
df -h /
```

Make sure current backups are restorable. If the machine serves users, choose a maintenance window or confirm that its services can tolerate a restart.

For a remote server, keep one working SSH session open while you update. Do not close it until a second connection succeeds after the change.

Refresh, review, then upgrade

```
sudo apt update
apt list --upgradable
sudo apt upgrade
```

`apt update` refreshes metadata. The list shows what can change. `apt upgrade` presents the transaction before applying it.

Read packages that are held back instead of forcing them individually. Ubuntu may phase some non-security updates while monitoring them. A held package is not automatically a broken system.

Verify the running services

An updated package can restart a service. A process can also continue using an old library until it restarts.

Check failed units and the services your application needs:

```
systemctl --failed
```

```
systemctl status nginx --no-pager
```

Run a real health check too. A green service state does not prove the application returns the expected response.

Ubuntu may report that processes or the kernel need a restart. Check whether a reboot is requested:

```
test -f /run/reboot-required && cat /run/reboot-required.pkgs
```

Plan the reboot. Do not reboot automatically in the middle of important work.

Know the recovery path

If a service fails, inspect its journal and package history before making another broad change:

```
sudo journalctl -u nginx -b --no-pager
less /var/log/apt/history.log
```

Restore a known configuration or package version only when you understand the dependency impact. A backup or server snapshot is not useful until you know how to restore it.

Try this: write an update checklist for one server: pre-checks, commands, application health check, reboot decision, and recovery contact.

Users, groups, and sudo

Create separate accounts, grant only required group membership, and use sudo instead of working as root all day.

Linux uses users and groups to decide which files and operations a process may access. Separate accounts reduce the damage caused by a mistake or compromised service.

Do not work as `root` all day. Use a regular account and elevate only the command that needs administrative access.

Create a regular user

```
sudo adduser ada
```

`adduser` creates the account, home directory, and initial group. Confirm the result:

```
id ada
getent passwd ada
```

If this person needs administrative access, add the account to Ubuntu's `sudo` group:

```
sudo usermod -aG sudo ada
```

The `-a` matters. It appends a supplementary group. Omitting it can replace the user's existing supplementary group list.

New group membership normally appears after the user starts a new login session. Test it in a second session before changing the account you currently depend on.

Use groups for shared access

Suppose two deployers need access to `/srv/notes`. Create a dedicated group instead of making the directory world-writable:

```
sudo groupadd notes-deploy
sudo usermod -aG notes-deploy ada
sudo chgrp -R notes-deploy /srv/notes
```

Then choose directory permissions that match the workflow. Do not apply recursive permission changes until you have inspected the tree and understand which files should be executable.

`sudo` is a boundary

`sudo` runs one command with elevated privileges and records the attempt. Read the complete command, resolved path, and target before pressing Return.

Redirection is a common trap:

```
sudo echo enabled > /etc/notes.conf
```

The shell opens the file before `sudo` runs, so this usually fails for a regular user. Use a privileged editor or `sudo tee` when appropriate, and review the value before writing it.

Try this: on a disposable system, create a user without sudo access. Verify its identity, then explain which exact additional permission you would grant for one administrative job.

Check your understanding: software and users

Check repositories, apt update and upgrade, package removal, third-party sources, users, groups, root, and sudo.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Services and logs

Control systemd services, understand startup, and investigate the journal and application logs.

What a service is

Understand long-running background programs and how systemd represents them as units with dependencies and state.

A service is a program that runs in the background, often starting during boot and waiting for work. A web server waiting for requests, the SSH daemon waiting for logins, a database waiting for queries — all services.

Services exist because a server must do its job with nobody logged in. You cannot keep a terminal open running the web server by hand. Something has to start these programs at boot, restart them when they crash, and keep them running after you disconnect.

systemd and units

Ubuntu uses systemd to manage services and other **units**. A unit file describes how to start a program, its dependencies, restart behavior, and installation target. A minimal one looks like this:

```
[Unit]
Description=Notes API
```

```
[Service]
ExecStart=/usr/local/bin/notes-api
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

Read it top to bottom: what this is, what command runs it and what to do when it fails, and when it should start during boot.

Reading a service's state

Ask systemd about any service:

```
systemctl status ssh
```

Two lines in the output carry the state you care about:

```
Loaded: loaded (/usr/lib/systemd/system/ssh.service; enabled; ...)
Active: active (running) since Mon 2026-08-03 09:14:02 UTC
```

Do not confuse a running service with an enabled service. Running describes now. Enabled describes whether it should start during boot. The `Active:` line answers the first question; the word after the unit path in `Loaded:` answers the second.

The two are independent, and that independence causes a classic failure. You install something, start it, it works — then the server reboots weeks later and the service is down. It was running but never enabled, so nothing brought it back. When a service is mysteriously dead after a reboot, check the `Loaded:` line first: `disabled` there is the diagnosis.

The next lessons cover starting, stopping, and enabling services yourself.

Control a systemd service

Inspect, start, stop, restart, reload, enable, and disable a service while choosing the least disruptive action.

Systemd manages long-running services on Ubuntu. A **unit** describes how a service starts, stops, restarts, and relates to other parts of the system.

Start by inspecting the current state:

```
systemctl status nginx --no-pager
systemctl is-active nginx
systemctl is-enabled nginx
```

`active` describes the current runtime state. `enabled` describes whether the unit is configured to start through its boot dependencies. A service can be active but disabled, or enabled but currently failed.

Choose the least disruptive action

Use `start` and `stop` to change the current runtime state:

```
sudo systemctl start nginx
sudo systemctl stop nginx
```

`restart` stops and starts the service. Active connections may be interrupted.

`reload` asks a running service to reread configuration without a full restart:

```
sudo systemctl reload nginx
```

Not every service supports reload. Check its documentation and unit definition. A successful reload command also does not prove the new configuration is valid or serving the intended site.

Boot behavior is separate

```
sudo systemctl enable nginx
sudo systemctl disable nginx
```

These commands change future boot behavior. Add `--now` only when you intentionally want to change both boot and current runtime state.

Validate before restart or reload

Use the application's own check first:

```
sudo nginx -t
sudo systemctl reload nginx
```

Then verify the unit, logs, listening port, and real request:

```
systemctl status nginx --no-pager
sudo journalctl -u nginx -n 30 --no-pager
ss -lnt
curl -I http://127.0.0.1/
```

If the new behavior fails, restore the last working configuration and reload again. Do not keep restarting a service without reading the error.

Try this: choose one harmless service on a test machine. Record whether it is active and enabled, find its unit definition with `systemctl cat`, and identify its validation command.

Read the system journal

Filter journal messages by service, boot, time, and severity instead of scrolling through unrelated output.

Systemd services normally send standard output, errors, and lifecycle messages to the journal. `journalctl` reads those structured records.

Running `journalctl` without filters can produce thousands of unrelated lines. Start with the service and time of the failure.

Filter one service

```
sudo journalctl -u nginx --since '20 minutes ago' --no-pager
```

Add `-b` to limit the query to the current boot:

```
sudo journalctl -u nginx -b --no-pager
```

Show the latest 50 lines with useful explanations:

```
sudo journalctl -u nginx -n 50 -e --no-pager
```

Follow new messages while reproducing a problem:

```
sudo journalctl -u nginx -f
```

Press `Ctrl-C` when you have captured the failure.

Compare boots and severity

List known boots:

```
journalctl --list-boots
```

Read kernel messages from the previous boot:

```
sudo journalctl -k -b -1 --no-pager
```

Filter warnings and more severe messages for the current boot:

```
sudo journalctl -b -p warning --no-pager
```

Severity is a clue, not a verdict. A service can log an important failure as ordinary text, while an old warning may be unrelated.

Read the event in context

Find the first relevant error, then read the lines before and after it. A final `connection refused` message may be the result of an earlier configuration or permission error.

Keep timestamps, unit name, and exact message when reporting a problem. Redact tokens, email addresses, and customer data before sharing logs.

Try this: restart a harmless test service, then use one journal query to show only that service's messages from the current boot and last five minutes.

Boot, reboot, and scheduled work

Know when a reboot is necessary and choose between cron and systemd timers for repeated background work.

A reboot stops every process on the machine. Do not use it as the first response to a service problem.

Before rebooting a remote server, check why the reboot is needed, who is connected, and whether important work is running:

```
test -f /run/reboot-required && cat /run/reboot-required.pkgs
who
systemctl --failed
```

Confirm that services start automatically, backups are current, and you have console access if SSH does not return.

Reboot with:

```
sudo systemctl reboot
```

Keep your current SSH session open until the command disconnects it. After the machine returns, verify the new boot, failed units, listening ports, and application health.

Schedule repeated work deliberately

Cron is a small, widely available scheduler. Edit your user crontab with:

```
crontab -e
```

This entry runs a backup script every day at 02:15:

```
15 2 * * * /home/ada/bin/backup-notes >> /home/ada/log/backup.log 2>&1
```

Scheduled jobs run with a smaller environment than your interactive shell. Use absolute paths, set required variables explicitly, and collect both output and errors.

Test the command manually with the same user before scheduling it.

When a systemd timer is a better fit

Systemd timers pair scheduling with a service unit. They provide journal logs, dependency ordering, resource controls, and options for missed runs.

Inspect existing timers with:

```
systemctl list-timers --all
```

Use a timer when the job is part of system operation and you want the same service management and logging as other units. Cron remains fine for a small personal job.

Avoid overlapping runs

A backup that takes longer than its interval can start twice. Design the job to detect or prevent overlap, and make repeated execution safe when possible.

After scheduling, verify that the job ran and produced the expected artifact. A scheduler reporting success only proves that it launched the command.

Try this: write a schedule for a harmless command in a disposable environment. Include an absolute command path, an output destination, and a way to verify the result.

Check your understanding: services and logs

Check services, systemd units, running and enabled state, reloads, journal filters, reboots, cron, and scheduled environments.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Resources and networking

Inspect storage, memory, CPU, interfaces, listening ports, DNS, and common failures.

Inspect disk and memory

Distinguish filesystem capacity, directory usage, RAM pressure, and swap before deciding what resource is exhausted.

Disk capacity and memory pressure are different problems. Diagnose the resource that is actually exhausted before deleting files or restarting services.

Filesystem capacity

Use `df` to inspect mounted filesystems:

```
df -h
df -ih
```

`df -h` reports byte capacity. `df -ih` reports **inodes**, the filesystem records used for files and directories. A filesystem can have free gigabytes but no free inodes after creating millions of tiny files.

Find large top-level directories without crossing into other mounted filesystems:

```
sudo du -xhd1 /var | sort -h
```

Move deeper only into the large area you found. Do not start by deleting `/var/log` or `/var/lib`; active logs and database files may live there.

If you prefer an interactive view, `ncdu` scans a directory tree, sorts entries by size, and lets you drill into the largest directories. Run it on the filesystem or directory that `df` identified instead of scanning every mounted filesystem blindly:

```
sudo ncdu -x /var
```

If `df` shows full space but `du` cannot account for it, check for deleted files still held open:

```
sudo lsof +L1
```

The space returns when the owning process closes the file. Restart that service only after you understand the impact.

Memory and swap

Use:

```
free -h
```

Linux uses otherwise idle RAM for filesystem cache. Low `free` memory alone is not an emergency. The `available` estimate is more useful because it includes memory the kernel can reclaim.

Swap use is also not automatically a failure. Look for sustained swapping, slow responses, out-of-memory messages, and a process whose memory keeps growing.

Inspect the largest processes:

```
ps -eo pid,user,%mem,rss,comm --sort=-rss | head
```

Then correlate the process with application metrics and logs. A database using memory for a configured cache may be healthy; an unknown worker growing without limit may not be.

Verify the fix

After cleanup or a service change, rerun the same commands and test the application. Record what freed the resource so the problem can be prevented with rotation, limits, alerts, or capacity changes.

Try this: inspect a test machine with `df -h`, `df -ih`, `du`, and `free -h`. Write one sentence

explaining what each command can prove.

Inspect CPU and processes

Find the process using resources and collect evidence before killing it or restarting the whole machine.

When a machine feels slow, collect evidence before killing the busiest process or rebooting everything.

Start with a system-level view:

```
uptime
top
```

`uptime` shows load averages for roughly the last 1, 5, and 15 minutes. Load is not the same as CPU percentage. It includes runnable work and some tasks waiting on uninterruptible operations such as disk I/O.

Compare load with the number of CPU cores:

```
nproc
```

A load of 4 means something different on a 2-core machine and a 32-core machine.

Capture a repeatable process snapshot

Use `ps` when you want output you can sort and save:

```
ps -eo pid,ppid,user,stat,%cpu,%mem,etime,comm --sort=-%cpu | head
```

Look at elapsed time, process state, CPU, memory, owner, parent, and recent service logs together. A short CPU spike during a build is normal. A worker consuming one core continuously after every request may need investigation.

Find the service behind a process before acting:

```
systemctl status 1234
```

Systemd can often map a PID to its unit. If not, inspect `/proc/1234` or the process command line without exposing secrets.

Stop gracefully first

For a managed service, prefer its service command:

```
sudo systemctl stop notes-api
```

For an individual process, ordinary `kill PID` sends `SIGTERM`, giving the program a chance to close files and connections.

Use `kill -KILL PID` only when the process cannot respond. `SIGKILL` cannot be handled, so cleanup code does not run. Killing a database or writer this way can require recovery.

Verify the result

Confirm that the intended process stopped, system pressure changed, and the application recovered. Then find the cause. Restarting a leaking process restores capacity temporarily but does not fix the leak.

Try this: run one harmless CPU-intensive command on a test system. Find its PID with `ps`, inspect its elapsed time and state, then stop it with `SIGTERM`.

Interfaces, addresses, and ports

Inspect how the machine is connected and identify which process is listening before changing a firewall or service configuration.

A network problem can live at several boundaries: interface, address, route, listening socket, host firewall, cloud firewall, or application protocol.

Inspect them in that order instead of changing everything at once.

Interfaces and addresses

```
ip address
```

Look for the expected interface, an `UP` state, and the address assigned to it. `127.0.0.1` and `::1` are loopback addresses. A service bound only to loopback cannot accept connections sent to the machine's external address.

Routes

```
ip route
ip route get 1.1.1.1
```

The route table shows which gateway and interface Linux will use. `ip route get` asks the kernel which route it would select for one destination without sending a packet.

Listening sockets

```
sudo ss -lntup
```

The flags show listening TCP and UDP sockets, numeric addresses, and owning processes when permissions allow.

Compare these listeners:

```
127.0.0.1:3000
0.0.0.0:3000
[::]:3000
```

The first listens only on IPv4 loopback. `0.0.0.0` means all IPv4 interfaces. `[::]` is the IPv6 unspecified address; exact dual-stack behavior depends on the system and application.

Listening does not mean reachable

A socket can listen while traffic is blocked by UFW, a provider firewall, a router, or a security group. Test locally first:

```
curl http://127.0.0.1:3000/
```

Then test the external path from another machine. Do not open a firewall port until you have confirmed which service owns it and whether it should be public.

Record evidence at each layer. If the local request works but the remote TCP connection times out, changing application code is unlikely to help.

Try this: choose one local service. Find its listening address and process, test it through loopback, and explain whether another machine should be able to reach it.

DNS and a first diagnostic

Separate name resolution, network reachability, port access, service health, and application behavior during troubleshooting.

When a service appears down, test one layer at a time. The message `site unavailable` does not tell you whether DNS, routing, a firewall, the process, TLS, or the application failed.

Use a sequence that moves from name to behavior.

1. Resolve the name

Use the system resolver:

```
getent ahosts notes.example
```

Use `dig` when you need DNS details:

```
dig notes.example A
dig notes.example AAAA
```

Confirm that the returned address is the one the service should use. If different resolvers disagree, caching or delegation may be involved.

2. Check the network path and port

`ping` can show IP reachability when the destination permits ICMP, but many healthy servers block it. A failed ping does not prove the web service is down.

Test the actual TCP port:

```
nc -vz notes.example 443
```

A timeout suggests filtering or an unreachable path. `Connection refused` normally means the host answered but nothing accepted that port.

3. Check the service locally

On the server:

```
systemctl status nginx --no-pager
sudo ss -lntp
curl -v http://127.0.0.1/
```

This separates process state, listening socket, and application response.

4. Check the complete public request

```
curl -v https://notes.example/
```

Read the resolved address, connection, TLS handshake, status code, and response headers. A `500` response proves that DNS and the connection worked; the application failed later.

Change only the failing layer

Changing DNS will not fix a stopped process. Restarting the process will not fix a record pointing to the wrong IP. Opening every firewall port can turn diagnosis into a security incident.

Write down the first failing check and the last successful one. That boundary tells you where to investigate next.

Try this: choose a public site and run only read-only checks for DNS, TCP port 443, and HTTP. Explain what each successful result proves—and what it does not.

Check your understanding: resources and networking

Check disk, directory usage, memory, swap, load, processes, interfaces, routes, ports, DNS, and layered diagnostics.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/linux-basics/.