Linux Storage and Backups Course

Flavio Copes

Updated August 3, 2026

Contents

Learn Linux server disks, partitions, filesystems, mounts, LVM, capacity management, backup design, restores, and safe storage changes.

What you'll learn: Attach and mount storage, extend it safely, diagnose capacity problems, and prove a server backup through restoration.

Disks and filesystems

Identify block devices, partitions, filesystems, UUIDs, and the data each layer owns.

Map the storage stack

Separate physical or virtual devices, partitions, volume management, filesystems, and mount points.

Server storage is a stack. A filesystem is not the same thing as the disk or logical volume underneath it.

When a disk fills up or fails, you fix the problem at one specific layer. If you can't name the layers, you end up guessing.

The layers

From bottom to top:

- A **block device**: a physical disk, cloud volume, or virtual disk. Cloud volumes and virtual disks still appear as block devices inside Linux, with names like `/dev/sda` or `/dev/nvme0n1`.
- An optional **partition**, a slice of that device.
- An optional **volume management** layer such as LVM, which pools devices and hands out flexible volumes.
- A **filesystem** such as ext4 or XFS, which turns raw blocks into files and directories.
- A **mount point**, the directory where that filesystem becomes visible.

Capacity is owned near the bottom. Files are owned at the top. You grow a disk at the block layer, but you free space at the filesystem layer.

See the whole stack

Use `lsblk -f` to see devices, partitions, filesystem types, labels, UUIDs, and mount points:

```
lsblk -f
```

```
NAME           FSTYPE      LABEL UUID                                   MOUNTPOINTS
sda
 sda1          vfat              70D2-1F02                              /boot/efi
 sda2          ext4              1c24164b-92e0-43d9-a5cb-9b1d2f3a8a11 /
 sda3          LVM2_member       pXlZ2f-Qm3t-0dKe-7hgB-Vc2a-N9uY-4rTq
   vg0-data    ext4              9a7f43c2-6c1d-4c2e-8f3a-2b64d0e51c77 /srv/data
sdb             ext4              4e0c2ab8-11f9-4b62-9d34-8a2f6c0d91be /mnt/backups
```

The indentation is the stack itself. `sda3` is a partition, it belongs to LVM, and the logical volume `vg0-data` carries the ext4 filesystem mounted at `/srv/data`.

Trace one path

To answer "which device holds this directory?", ask `findmnt`:

```
findmnt /srv/data
```

```
TARGET    SOURCE                FSTYPE OPTIONS
/srv/data /dev/mapper/vg0-data ext4   rw,relatime
```

Here's a classic mistake this catches. A big data volume exists, but it was never mounted, so `/srv/data` is just an empty directory on the root filesystem. The application writes there happily until the small root disk fills. `findmnt` would have shown the source as `/dev/sda2`, not the data volume.

Draw the stack for one server path from `/srv/data` down to its block device. Mark which layer owns capacity and which layer owns files.

Identify a new device

Confirm a newly attached disk by stable facts before running any command that can overwrite data.

Device names such as `/dev/sdb` can change when hardware or attachment order changes.

The kernel hands out names in discovery order. Add a disk, reboot, change a cable or a cloud attachment, and yesterday's `/dev/sdb` can become today's `/dev/sdc`. Formatting a device is irreversible, so identifying the target by name alone is how people destroy the wrong disk.

Never choose a destructive target from position alone. Confirm it with facts that don't depend on ordering: size, model, serial number, and what's already on it.

Collect the facts

Start with `lsblk` and ask for the columns that identify hardware:

```
lsblk -o NAME,SIZE,TYPE,MODEL,SERIAL
```

```
NAME   SIZE TYPE MODEL            SERIAL
sda     80G disk Samsung SSD 870  S5XANG0N123456A
sdb    500G disk WDC WD5000AZLX   WD-WCC6Z4123456
```

`SIZE` should match what you attached. `MODEL` and `SERIAL` should match the label on the physical disk or the volume metadata in your cloud provider's console.

Then check for existing filesystem signatures:

```
sudo blkid /dev/sdb
```

If the disk is genuinely new, `blkid` prints nothing. Output like `TYPE="ext4"` means there is already a filesystem there. Stop and find out whose data that is before going further.

The kernel log confirms what was just attached:

```
sudo dmesg | tail -5
# [9128.442] sd 2:0:1:0: [sdb] 976773168 512-byte logical blocks: (500 GB/465 GiB)
# [9128.451] sd 2:0:1:0: [sdb] Attached SCSI disk
```

If the disk isn't brand new, `smartctl -i /dev/sdb` from the `smartmontools` package gives you the model and serial again, plus SMART health data. A used disk that's already failing is worth knowing about before you trust it.

The rule

Compare `lsblk`, `blkid`, size, model, serial, filesystem signatures, and provider metadata. You want at least three independent facts pointing at the same device. "It's the second one in the list" is not a fact. Unmount and back up anything uncertain.

Attach a disposable volume. Record at least three independent facts that identify it, then explain why another existing device is not the target.

Partition or use the whole device

Choose a partition table or whole-device filesystem from operational needs instead of habit.

A data disk can hold a filesystem directly or contain a partition table with one or more partitions.

Both work. The choice is operational, not technical purity.

The whole-device option

You can put a filesystem straight on `/dev/sdb` with no partition table. One less layer, nothing to size, and growing a cloud volume later means growing just the filesystem.

The downside is discoverability. Partitioning tools and installers see a disk with no partition table and may present it as blank. A colleague, or an automated provisioning script, is one confirmation prompt away from wiping it. A partition table is a visible ”this disk is in use” marker.

The partitioned option

GPT is the normal partition-table choice for modern disks. The older MBR format has a 2 TiB limit and no real advantages on a data disk.

Inspect what's there first:

```
sudo parted /dev/sdb print
# Error: /dev/sdb: unrecognised disk label
```

That error is what a truly blank disk looks like. Now create a GPT table and one partition spanning the disk:

```
sudo parted /dev/sdb -- mklabel gpt mkpart data ext4 1MiB 100%
```

The `1MiB` start keeps the partition aligned. Verify the result:

```
sudo parted /dev/sdb print
```

```
Partition Table: gpt
Number  Start   End    Size   File system  Name  Flags
 1      1049kB  215GB  215GB               data
```

The `File system` column stays empty until you actually run `mkfs` on `/dev/sdb1`, in the next lesson. `parted` only reserved the space.

How to decide

Partitions can separate roles on one disk, and boot disks need them because firmware and bootloaders expect specific partitions. My advice for a single-purpose data disk: one GPT partition covering the whole device. You keep the ”disk is in use” marker and give up almost nothing.

Whatever you pick, document the chosen layout. Six months from now, ”why does this disk have no partition table?” should have an answer in your notes, not a guess.

For a hypothetical 200 GB data disk, write the exact intended layout before creating it. Include device, table, partition boundaries, filesystem, and mount point.

Create and identify a filesystem

Format only a verified empty target and record a stable UUID or label for future mounts.

Formatting creates filesystem metadata and destroys access to existing filesystem contents on the target.

`mkfs` doesn't erase every block, but it overwrites the structures that locate your files. From the system's point of view, the old data is gone. So the command itself is easy; the discipline is in what you do before running it.

Verify, then format

Choose a filesystem supported by your distribution and workload. For a general-purpose data disk on Ubuntu or Debian, ext4 is the safe default. Run the appropriate `mkfs` command only after target verification:

```
sudo blkid /dev/sdb1        # no output = no existing signature
sudo mkfs.ext4 /dev/sdb1
```

```
Creating filesystem with 52428800 4k blocks and 13107200 inodes
Filesystem UUID: 9a7f43c2-6c1d-4c2e-8f3a-2b64d0e51c77
Writing superblocks and filesystem accounting information: done
```

If `mkfs.ext4` finds an existing filesystem it stops and asks `Proceed anyway? (y,N)`. That prompt is a stop sign, not an inconvenience. Answer `n` and go figure out what's on that device.

Use `blkid` to record its UUID and type afterward:

```
sudo blkid /dev/sdb1
# /dev/sdb1: UUID="9a7f43c2-6c1d-4c2e-8f3a-2b64d0e51c77" TYPE="ext4" PARTLABEL="data"
```

The **UUID** is generated at format time and stays with the filesystem forever, even if the device name changes. That's the identifier you'll use in `/etc/fstab` later, never `/dev/sdb1`.

You can also assign a human-readable **label** at format time with `mkfs.ext4 -L data /dev/sdb1`, or later with `e2label`. Labels are nicer to read, but nothing stops two disks from carrying the same label, and then which one mounts is a coin flip. UUIDs don't have that problem. My advice: set a label for humans, mount by UUID for the machine, and write both down somewhere your future self will look.

Practice on a loop device

You don't need spare hardware to rehearse this. A **loop device** turns a plain file into a block device:

```
truncate -s 1G /tmp/disk.img
sudo losetup --find --show /tmp/disk.img   # prints /dev/loop0
sudo mkfs.ext4 /dev/loop0
sudo mount /dev/loop0 /mnt/lab
echo "hello" | sudo tee /mnt/lab/test.txt
sudo umount /mnt/lab
sudo losetup -d /dev/loop0
```

Same commands, zero risk. This is how I test any formatting or mount procedure before touching a real disk.

Use a loop device or disposable volume. Verify it, create one filesystem, inspect its UUID, mount it temporarily, write a test file, and unmount it.

Check your understanding: disks and filesystems

Check block devices, partitions, filesystems, formatting, UUIDs, labels, and destructive boundaries.

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

Take this interactive quiz online

Mounts and capacity

Mount storage safely, configure startup mounts, and investigate space and inode exhaustion.

Mount and unmount safely

Attach a filesystem at an intentional empty directory and verify users and processes release it before removal.

A mount hides any files already present under its mount point until the filesystem is unmounted.

This surprises people. If a service writes to `/srv/data` before the data volume gets mounted there, those files land on the root filesystem. Mount the volume and they vanish from view — still consuming root disk space, invisible until you unmount again. Mount points should be dedicated, empty directories for exactly this reason.

Mounting

Create a dedicated directory, mount by device for the first test, and verify with `findmnt`:

```
sudo mkdir -p /mnt/lab
sudo mount /dev/sdb1 /mnt/lab
findmnt /mnt/lab
```

```
TARGET   SOURCE     FSTYPE OPTIONS
/mnt/lab /dev/sdb1  ext4   rw,relatime
```

`TARGET` is where the filesystem appears, `SOURCE` is the device behind it, and `OPTIONS` shows how it was mounted — `rw` here, so writes are allowed. If `findmnt` prints nothing, the mount didn't happen.

A manual `mount` lasts until reboot. Persistent mounts belong in `/etc/fstab`, which gets its own lesson because a typo there can break boot.

Unmounting

Before unmounting, stop writers and check nothing still holds the filesystem open:

```
sudo umount /mnt/lab
# umount: /mnt/lab: target is busy.
```

`target is busy` means some process still has a file open there, or its working directory inside it. Use `fuser` or `lsof` to find processes holding paths open:

```
sudo fuser -vm /mnt/lab
```

```
                     USER    PID ACCESS COMMAND
/mnt/lab:            flavio  4211 ..c.. bash
```

The `c` under `ACCESS` means that process's current directory is inside the mount. The most common offender is your own shell — you `cd`'d into the mount earlier. `cd` out, stop any services that write there, and retry the `umount`.

Never unplug or detach a mounted device without unmounting. Cached writes that haven't reached the disk yet are lost, and the filesystem can be left needing repair.

Mount a disposable filesystem under `/mnt/lab`. Create a file, inspect the mount, leave the directory, unmount it, and verify the original mount point is visible again.

Configure fstab without losing boot

Create a persistent mount by UUID and test the complete fstab before relying on the next reboot.

`/etc/fstab` describes filesystems systemd should make available, but one invalid required entry can complicate boot.

Prefer UUID or a deliberate label over an unstable device name. Create the mount point first. After editing, run `findmnt --verify` and test with `mount -a` while recovery access still works.

Add a disposable filesystem to fstab using its UUID and `nofail`. Test it without rebooting, then explain whether the real workload should tolerate the disk being absent.

Use the filesystem UUID, create the mount point, and test without rebooting:

```
sudo blkid /dev/sdb1
sudo mkdir -p /srv/data
sudoedit /etc/fstab
sudo mount -a
findmnt /srv/data
```

Keep a root shell or provider console open. A typo in fstab can delay or break boot. Run mount -a and verify the expected device, filesystem, options, and writable path before scheduling a reboot.

Diagnose space and inode exhaustion

Separate filesystem block use, directory totals, reserved space, and inode exhaustion when writes fail.

`df -h` reports allocated filesystem blocks. `du` totals reachable files under a path. The numbers answer different questions.

When writes start failing with `No space left on device`, don't guess. Work through the layers in order.

Which filesystem is actually full?

Start with the exact failing path:

```
df -h /var
```

```
Filesystem           Size  Used Avail Use% Mounted on
/dev/mapper/vg0-var   20G   19G     0 100% /var
```

`Size` is the filesystem's capacity, `Used` is allocated blocks, `Avail` is what unprivileged processes can still write, and `Mounted on` tells you which mount you're really measuring. Note that `Used` plus `Avail` may not equal `Size`: ext4 reserves a slice (5% by default) for root, so ordinary users hit 100% first.

Blocks, or inodes?

Every file needs an **inode**, the metadata record that holds its owner, permissions, and block locations. On ext4 the inode count is fixed at format time. Many tiny files can exhaust inodes while bytes remain:

```
df -i /var
```

```
Filesystem            Inodes  IUsed IFree IUse% Mounted on
/dev/mapper/vg0-var  1310720 1310720   0  100% /var
```

`IUse%` at 100% means "no space left" errors even though `df -h` shows free gigabytes. The usual culprit is a runaway directory of session files, cache entries, or undelivered mail.

Find what's growing

Use targeted `du` to walk down toward the biggest directory:

```
sudo du -xh --max-depth=1 /var | sort -h | tail -5
```

The `-x` flag stops `du` from crossing into other mounted filesystems, so you measure only the full one. Repeat one level deeper each time until you find the growth source.

For inode hunts, count entries instead of bytes:

```
sudo find /var/lib/php/sessions -maxdepth 1 | wc -l
```

Then use application knowledge; do not delete unknown files from `/var` blindly. Find the owning service and use its own cleanup mechanism. And if `df` says full while `du` finds much less, a deleted-but-open file is holding the space — that's the next lesson.

Create a capacity checklist that starts with the exact failing filesystem. Include blocks, inodes, largest directories, growth source, and a safe cleanup owner.

Find deleted open files and set quotas

Recover space held by running processes and use quotas when shared storage needs enforceable ownership boundaries.

Deleting a file removes its directory entry, but a process with the file open can keep its blocks allocated.

The kernel only frees the blocks when the last open file descriptor closes. This is the classic mystery: `df` says the disk is full, `du` finds far less, and nothing you delete makes `df` move. Someone removed a huge log file, but the service that writes it is still running and still holding it open.

Find the holder

Use `lsof +L1` to find deleted open files. It lists open files with a link count below one, meaning no directory entry points at them anymore:

```
sudo lsof +L1
```

```
COMMAND  PID USER  FD  TYPE DEVICE   SIZE/OFF NLINK NODE    NAME
nginx   1214 root   5w  REG  253,2 8241733632     0 5311 /var/log/nginx/access.log (deleted)
```

Read the columns: `COMMAND` and `PID` identify the process, `FD 5w` means file descriptor 5 open for writing, `SIZE/OFF` shows almost 8 GB still allocated, `NLINK` is 0 because the file is deleted, and `NAME` confirms which file it was.

Release the space

Restart or signal the owning service through its normal lifecycle:

```
sudo systemctl restart nginx
df -h /var/log    # space is back
```

For log files specifically, the service's reload or log-rotation signal is even gentler than a restart. What you should not do is delete active log files with `rm` in the first place — truncate them instead (`truncate -s 0 access.log`), which frees the blocks without breaking the writer.

Try it on a test system: run `tail -f /tmp/big.log` in one shell, delete the file from another, and watch it appear in `lsof +L1`. Stop `tail` and the space returns. No reboot needed.

Quotas for shared storage

On multi-user or multi-tenant storage, cleanup after the fact is the wrong tool. Filesystem quotas can bound users or groups: on ext4 you enable the `usrquota` or `grpquota` mount options, then set per-user block and inode limits with `setquota` and review usage with `repquota`.

Quotas need monitoring and clear failure handling. A user hitting their quota sees the same ”no space” error as a full disk, so document who owns which limit and what happens when it's reached.

On a test system, open a file, delete it, and observe the open descriptor. Release it normally and verify space returns without rebooting the server.

Check your understanding: mounts and capacity

Check temporary and persistent mounts, fstab safety, capacity, inodes, deleted files, and quotas.

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

Take this interactive quiz online

LVM and growth

Understand physical volumes, volume groups, logical volumes, snapshots, and safe expansion.

Understand LVM layers

Follow storage through physical volumes, volume groups, logical volumes, filesystems, and mount points.

LVM creates a flexible allocation layer between block devices and filesystems.

Without it, a filesystem is welded to its partition. Resizing means repartitioning, and one filesystem can't span two disks. LVM solves this by pooling storage and handing it out in adjustable pieces.

The three layers

- A **physical volume** (PV) is a block device handed over to LVM. It contributes **extents**, fixed-size chunks (4 MiB by default), to a pool.
- A **volume group** (VG) is that pool. It can contain several physical volumes.
- A **logical volume** (LV) allocates extents from the pool and behaves like a normal block device. It normally holds a filesystem.

One inspection command per layer:

```
sudo pvs
#  PV         VG   Fmt  Attr PSize    PFree
#  /dev/sda3  vg0  lvm2 a--  <199.00g 49.00g

sudo vgs
#  VG  #PV #LV #SN Attr   VSize    VFree
#  vg0   1   2   0 wz--n- <199.00g 49.00g

sudo lvs
#  LV   VG  Attr       LSize   Pool Origin Data%
#  root vg0 -wi-ao---- 50.00g
#  data vg0 -wi-ao---- 100.00g
```

`pvs` shows each device and how much of it is unallocated (`PFree`). `vgs` sums the pool: `VFree` is what's left for new or growing volumes. `lvs` lists the volumes carved out of it; the `o` in `Attr` means the volume is open, so something (a filesystem) is using it.

Match a path to its layers

Take a mounted directory and walk down:

```
findmnt /srv/data
# TARGET    SOURCE                FSTYPE OPTIONS
# /srv/data /dev/mapper/vg0-data ext4   rw,relatime
```

The source `/dev/mapper/vg0-data` names the volume group and logical volume. `lsblk` shows the same relationship as a tree, from `sda3` up through `vg0-data`.

One thing to internalize now, because the next lesson depends on it: growing the logical volume and growing the filesystem are separate operations. `lvs` can report 150 GB while `df` still reports 100 GB, because the filesystem hasn't been told about the new space yet. Each layer only knows its own size.

Run `pvs`, `vgs`, and `lvs` on a system using LVM. Match one mounted path to its logical volume, volume group, and physical volume.

Extend a logical volume

Verify free extents, backups, device identity, and filesystem support before increasing usable capacity.

Growth normally moves outward in two steps: enlarge the logical volume, then enlarge the filesystem inside it.

The good news: for ext4 and XFS both steps work online, on a mounted filesystem, with the application running. Growing is the safe direction. You're adding blocks, not removing any.

Check before you grow

Inspect the current stack and free volume-group space first:

```
sudo vgs vg0
#  VG  #PV #LV #SN Attr   VSize    VFree
#  vg0   1   2   0 wz--n- <199.00g 49.00g

df -h /srv/data
#  Filesystem           Size  Used Avail Use% Mounted on
#  /dev/mapper/vg0-data  98G   88G  5.2G  95% /srv/data
```

`VFree` at 49 GB means the pool can cover a 20 GB extension. If `VFree` is near zero you first need to add a physical volume or grow the underlying disk. Confirm you're targeting the right volume, and confirm a recent backup exists — this is a low-risk operation, but it still rewrites volume metadata.

Extend both layers

Use explicit sizes. `+20G` says "grow by 20 GB", which is harder to get wrong than an absolute target:

```
sudo lvextend -L +20G --resizefs vg0/data
```

`--resizefs` (short form `-r`) tells `lvextend` to also grow the filesystem, calling `resize2fs` for ext4 or `xfs_growfs` for XFS on your behalf. Some tools can resize the filesystem with the logical volume like this, but you should still verify both layers afterward:

```
sudo lvs vg0/data
#  LV   VG  Attr       LSize
#  data vg0 -wi-ao---- 120.00g

df -h /srv/data
#  Filesystem           Size  Used Avail Use% Mounted on
#  /dev/mapper/vg0-data 118G   88G   25G  79% /srv/data
```

Both numbers moved. That's the check that matters.

The classic failure

Someone runs `lvextend` without `--resizefs`, sees it succeed, and walks away. `lvs` shows the new size, but `df` doesn't move and the application keeps failing with a full disk. The volume grew; the filesystem was never told.

The fix is harmless — run the missing step (`sudo resize2fs /dev/vg0/data` for ext4). But recognize the signature: LV size and filesystem size disagree, and the difference is exactly the extension you just made.

Write a dry-run expansion plan for a test volume. Include current sizes, target size, backup, commands, validation, and the point where rollback stops being simple.

Treat shrinking as a migration

Avoid casual in-place shrinking and choose a copy-and-verify migration when filesystem support or risk is uncertain.

Shrinking is more dangerous than growing because removing blocks can destroy live filesystem data.

Growing adds empty space nobody uses yet. Shrinking takes blocks away, and if the filesystem hasn't fully vacated them first, you cut into real data. The two operations look symmetric on the command line. They are not symmetric in risk.

Why in-place shrinking is hostile

Some filesystems cannot shrink at all. XFS is the big one: there is no shrink support, full stop. If your data volume is XFS, migration is your only option.

ext4 can shrink, but only offline. The supported workflow requires an unmounted filesystem, filesystem checks, exact ordering, and backups:

```
sudo umount /srv/data
sudo e2fsck -f /dev/vg0/data
sudo resize2fs /dev/vg0/data 80G     # shrink the filesystem FIRST
sudo lvreduce -L 80G vg0/data        # then the volume, never below the fs size
```

Order is everything. Reduce the logical volume before the filesystem, or reduce it below the filesystem's new size, and you've amputated live data. Unit confusion between the two commands has destroyed plenty of filesystems. `lvreduce --resizefs` exists and sequences both steps for you, but the operation still happens in place, on your only copy, with downtime.

The migration alternative

A new smaller volume plus verified copy is often clearer:

```
sudo lvcreate -L 80G -n data-new vg0
sudo mkfs.ext4 /dev/vg0/data-new
sudo mount /dev/vg0/data-new /mnt/data-new
sudo rsync -aHAX /srv/data/ /mnt/data-new/
```

Then verify before switching anything:

```
sudo rsync -aHAX --checksum --dry-run /srv/data/ /mnt/data-new/
# no output = the trees match
```

Stop writers, run one final `rsync` to catch late changes, swap the mounts, and keep the old volume around until the application has run happily for a few days. That's your rollback: remount the original.

My advice: treat any shrink request as a migration by default. The copy approach costs temporary disk space and a short cutover window. The in-place approach bets the only copy of your data on getting order and units right.

Given an oversized volume, compare in-place shrink with new-volume migration. Choose the safer plan and state how you will verify every required file arrived.

Use snapshots with limits

Use LVM snapshots for short consistency windows without treating them as independent backups.

An LVM snapshot initially shares data with its origin and stores changed blocks separately.

Creating one is nearly instant because nothing is copied up front. The snapshot presents the volume exactly as it was at creation time. As the origin changes, the original version of each changed block gets written into the snapshot's space — copy-on-write.

That gives you the one thing file-copy backups lack on a busy volume: a frozen, consistent view. Copying a live filesystem for an hour captures files from different moments. Copying from a snapshot captures one moment.

Create, use, remove

Give the snapshot enough space to absorb the changes you expect during the backup window:

```
sudo lvcreate --snapshot --size 5G --name data-snap /dev/vg0/data
sudo mount -o ro /dev/vg0/data-snap /mnt/snap
sudo rsync -aHAX /mnt/snap/ /mnt/backups/data/
```

Mounting read-only makes the intent clear: this is a source to copy from, not a place to work in.

Snapshots consume space as the origin changes and can fill. Monitor while the copy runs:

```
sudo lvs vg0
#  LV        VG  Attr       LSize   Origin Data%
#  data      vg0 owi-aos--- 100.00g
#  data-snap vg0 swi-a-s---   5.00g data   38.20
```

`Data%` is the number to watch. If it reaches 100 the snapshot becomes invalid, and your backup source vanishes mid-copy. The origin volume keeps working; only the snapshot dies. If `Data%` climbs fast, the write load is heavier than you sized for.

When the copy finishes, remove the snapshot:

```
sudo umount /mnt/snap
sudo lvremove vg0/data-snap
```

Don't leave snapshots around. An old, forgotten snapshot keeps absorbing writes, slows the origin, and eventually fills.

What a snapshot is not

They usually share the same failure domain as the source disk. If the disk dies, the origin and the snapshot die together. A snapshot is a consistency tool, not a backup. Use them to obtain a consistent backup view, monitor them, then remove them — and send the actual copy somewhere independent.

Plan a snapshot-assisted backup for one busy filesystem. Define quiescing, snapshot size, monitoring, copy destination, validation, and cleanup.

Check your understanding: lvm and growth

Check LVM layers, free extents, safe expansion, shrinking risk, snapshots, monitoring, and migration.

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

Take this interactive quiz online

Backup design

Choose data, recovery targets, copies, retention, consistency, encryption, and automation.

Start from restore requirements

Define what must return, acceptable data loss, and acceptable downtime before choosing a backup tool.

A backup plan begins with recovery, not a command.

Most people start from the tool: install something, point it at `/`, schedule it nightly, done. Then the disk dies and they discover the nightly run means losing a full day of orders, or that restoring takes six hours while the RAM-cached database they never dumped is gone entirely. The tool was fine. The requirements were never written.

Two numbers first

The **recovery point objective** (RPO) describes tolerable data loss in time. If the server dies right now, how old can the data you restore be? An RPO of 24 hours permits nightly backups. An RPO of 15 minutes demands something continuous or near-continuous.

The **recovery time objective** (RTO) describes tolerable downtime. How long can the service stay down while you rebuild? An RTO of a week tolerates manual, documented restores. An RTO of an hour requires rehearsed automation and locally available copies.

Both numbers are business decisions, not technical ones. Your job is to write them down and design backward from them.

Inventory what must return

Inventory application data, database state, configuration, secrets, and rebuildable artifacts separately, because each restores differently:

```
sudo du -sh /var/lib/postgresql /srv/uploads /etc/nginx /etc/letsencrypt
# 4.2G  /var/lib/postgresql
# 18G   /srv/uploads
# 124K  /etc/nginx
# 36K   /etc/letsencrypt
```

Database state needs a consistent dump, not a file copy. Secrets need protected storage. And rebuildable artifacts — packages, containers, compiled output — should usually be excluded: backing them up wastes space and restore time on things `apt` or your deploy pipeline recreates.

Then write the answers where the restorer will find them:

```
service: shop.flaviocopes.com
rpo: 1 hour          # max tolerable data loss
rto: 4 hours         # max tolerable downtime
restore owner: flavio
must restore: postgres DB, /srv/uploads, nginx config, TLS certs
rebuildable: OS, packages, app code (git), containers
first test: place a test order end to end
```

That last line matters. "Backup completed" proves nothing; a named user-visible test proves recovery.

Choose one service. Write its RPO, RTO, restore owner, required data, and the first user-visible test that proves recovery.

Keep independent copies

Protect backups from server loss, operator mistakes, account compromise, and ransomware with separate failure boundaries.

A second directory on the same disk is a copy, but it does not survive disk or server loss.

The question to ask of every backup copy is: which failures does this one survive? A copy on the same disk survives an accidental `rm`. It does not survive the disk dying, the server being deleted, or ransomware encrypting the filesystem. Each copy is only as good as the failure boundary between it and the original.

Layer the copies

Keep multiple copies on different media, with at least one off-site. The classic formulation is the **3-2-1 pattern**: three copies of the data, on two different media, one of them off-site.

A local copy on a second disk handles the common cases fast:

```
sudo rsync -aHAX --delete /srv/data/ /mnt/backups/data/
```

An off-site copy over SSH survives losing the whole machine:

```
sudo rsync -aHAX /srv/data/ backup@backups.flaviocopes.com:/srv/copies/web01/data/
```

Note the `--delete` flag appears only on the local mirror. On the off-site copy I leave it off, or better, use a snapshotting tool like restic or borg that keeps history. Here's why that matters.

Replication is not backup

Replication alone can faithfully copy deletion and corruption. A mirror that syncs every hour will dutifully propagate tonight's ransomware encryption, or this afternoon's fat-fingered `rm -rf`, to every replica by morning. What saves you is **history**: older versions that no current process can rewrite.

Independent credentials

If one stolen SSH key or cloud token can delete both the server and its backups, they are not independent copies. Give the backup destination its own credential that can only append, and administer retention from the storage side. Use separate credentials and immutable or offline retention where appropriate — object-lock on a bucket, or a disk that spends most of its life unplugged.

Map three plausible failures against your current copies. For each failure, name the copy that survives and the credential needed to restore it.

```
failure: data disk dies       -> survives: off-site copy  (backup SSH key)
failure: rm -rf on /srv/data  -> survives: local mirror   (root on server)
failure: server account stolen -> survives: ???           <- fix this one
```

If any row ends in question marks, that's your next piece of work.

Make application-consistent backups

Coordinate files, databases, and writers so a completed backup represents a state the application can restore.

Copying files while an application writes them can capture related data from different moments.

Picture a backup that walks `/var/lib/mysql` file by file while the database commits transactions. The first file is copied at 02:00, the last at 02:40, and no single instant of the database's life looks like what landed in the backup. The copy completes with exit code 0. It may still be unrestorable.

Application consistency means the backup represents one moment the application itself could have been stopped at.

Three ways to get a consistent boundary

Use database-native dumps or backup APIs, filesystem snapshots with quiescing, or a documented stop window.

The database dump is the everyday choice. The engine produces a consistent export while staying online:

```
sudo -u postgres pg_dump --format=custom shopdb > /srv/backups/shopdb-2026-08-03.dump
```

The dump is consistent by construction: it sees one transaction-level view of the database no matter how long it takes.

A filesystem snapshot works when the data isn't in a database, or is too big to dump. The catch is **quiescing**: telling writers to flush and pause so the frozen view is clean, not a crash image.

The stop window is the honest fallback. Stop the service, copy, start it. Downtime, but perfect consistency, and for a small internal tool that's often the simplest correct answer.

Databases are more than data

Preserve schema, roles, extensions, and external objects required by the application. A `pg_dump` of one database does not include the cluster's roles — capture them separately:

```
sudo -u postgres pg_dumpall --globals-only > /srv/backups/pg-globals-2026-08-03.sql
```

Skip this and the restore fails with ”role does not exist” errors at the worst possible time.

The cross-boundary problem

Now the hard part: an application with a database and uploaded files. Dump the database, then copy the uploads, and a file uploaded in between exists on disk with no database row — or the reverse, depending on order.

Decide which inconsistency your application tolerates, and order the backup so you get that one. An orphaned file on disk is usually harmless; a database row pointing at a missing file is a broken page. So dump the database first, then copy files. If neither direction is tolerable, you need a brief pause of writers around the boundary.

Design a backup sequence for an application with a database and uploaded files. State how you create one consistent boundary across both.

Automate retention and alerts

Schedule backups, prune them deliberately, verify results, and alert when expected recovery points stop arriving.

A timer that exits successfully does not prove useful data reached durable storage.

I've seen backup jobs ”succeed” for months while the destination volume was unmounted, so every

run wrote into an empty directory on the root disk. Exit code 0 the whole time. Automation without evidence is just scheduled false confidence.

A backup job worth trusting

A production backup script needs locking, a timeout, and evidence. Locking so overlapping runs can't corrupt each other, a timeout so a hung run can't block tomorrow's, and logged evidence so you can audit what actually happened:

```
#!/usr/bin/env bash
set -euo pipefail
exec 9>/run/lock/backup-data.lock
flock -n 9 || { echo "previous run still active"; exit 1; }

start=$(date +%s)
snapshot=$(restic backup /srv/data --json | jq -r 'select(.message_type=="summary") | .snapsh
restic check --read-data-subset=5%
echo "$(date -Is) snapshot=$snapshot bytes=$(du -sb /srv/data | cut -f1) duration=$(( $(date
  >> /var/log/backup-data.log
```

Log the source, destination, bytes, duration, and snapshot identifier. When something looks wrong later, this log answers ”when did backups quietly shrink to 2 MB?”. Run it from a systemd service with `TimeoutStartSec=2h` and a daily timer, rather than a bare cron line — you get journal logging and timeout handling for free.

Retention is a policy, not a disk-full reaction

Apply a documented retention policy. Deleting old backups because the disk filled means deleting them at the moment you least control. Decide the policy up front and let the tool enforce it:

```
restic forget --keep-daily 7 --keep-weekly 4 --keep-monthly 6 --prune
```

Recent history dense, older history sparse. Write down why those numbers, so nobody ”cleans up” the monthlies during an incident.

Alert on silence, not just failure

Failure alerts miss the worst case: the job that never ran. Server off, timer disabled, credential expired — nothing fails, so nothing alerts.

The fix is a **dead man's switch**: the job pings a monitor on success, and the monitor alerts when the ping stops arriving. Alert on failed runs, missing recent backups, unexpected size changes, and repository integrity failures. The size check is the sneaky one — a backup that suddenly drops from 18 GB to 40 KB ”succeeded”, and it's telling you the source path is wrong.

Create a daily backup service and timer design. Include locking, timeout, retention, success evidence, and an alert that detects a silent missed run.

Check your understanding: backup design

Check recovery objectives, copy independence, consistency, retention, encryption, credentials, and automation.

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

Take this interactive quiz online

Restore and recover

Test restores, recover databases and configuration, handle failures, and review the plan.

Test a restore in isolation

Restore into a separate target and verify application behavior without overwriting the only production copy.

A backup is unproven until you restore it. Listing archive files proves only that an archive can be listed.

Use an isolated host, directory, database, or account. Verify checksums, ownership, permissions, service startup, representative records, and a user-visible workflow. Record elapsed recovery time.

Restore one small backup into isolation. Compare file counts and checksums, start the service if safe, and record whether the result met RPO and RTO.

Restore to a new directory, never over the live data first:

```
sudo mkdir -p /srv/restore-test
sudo rsync -aHAX /mnt/backup/app/ /srv/restore-test/
sudo diff -qr /srv/app /srv/restore-test
```

Check ownership, permissions, links, application configuration, and one real read workflow. For a database, restore into a separate database name and run integrity checks. Record elapsed time so the recovery-time target is based on evidence.

Restore files and permissions

Recover data together with ownership, modes, links, extended attributes, and required directory structure.

Application files are more than byte contents. Incorrect metadata can expose secrets or prevent a service from starting.

A restored private key with mode `644` instead of `600` is readable by every user on the machine — and OpenSSH will refuse to use it. An upload directory restored as `root:root` means the web application can no longer write to it. Both failures happen with byte-perfect file contents.

What metadata a restore must carry

- **Ownership**: user and group of every file.
- **Modes**: the permission bits, including setuid and setgid where present.
- **Symbolic links**: restored as links, not as copies of their targets.
- **Hard links**: preserved, or the restored tree silently doubles in size.
- **Extended attributes and ACLs**: some workloads depend on them.
- **Directory structure**: empty directories a service expects at startup.

Choose backup and copy tools that preserve the metadata your workload needs. For `rsync`, that's this flag set:

```
sudo rsync -aHAX /mnt/backups/etc-nginx/ /srv/staging/etc-nginx/
```

`-a` keeps owners, groups, modes, timestamps, and symlinks. `-H` preserves hard links, `-A` ACLs, `-X` extended attributes. Run it as root: an unprivileged user cannot recreate other users' ownership, so the same command without `sudo` silently restores everything as you.

Restore to staging, then inspect

Restore to a staging path first. Inspect owners, modes, symbolic links, and any ACLs before cutover:

```
ls -l /srv/staging/etc-nginx/ssl/
# -rw------- 1 root root 1704 Jul 12 03:00 flaviocopes.com.key
# -rw-r--r-- 1 root root 5834 Jul 12 03:00 flaviocopes.com.crt
```

The key is `600`, the certificate is `644`, both owned by root. That's what I expect. Then compare the staged tree against the source mechanically:

```
sudo rsync -aHAXn --itemize-changes /mnt/backups/etc-nginx/ /srv/staging/etc-nginx/
```

The `-n` makes it a dry run, and `--itemize-changes` prints one line per difference — content, permissions, owner, or timestamps. No output means the trees match, metadata included. A line like `.f....og..` flags a file whose owner or group differs: exactly the class of problem that only surfaces when the service tries to start.

Pick one configuration tree and one data tree. List the metadata each requires, restore both to staging, and compare them with the source.

Restore a database and configuration

Recover database state, roles, schema, secrets, and service configuration in a controlled order.

A database dump without its roles, extensions, version requirements, and application configuration may not produce a working service.

The dump file is the easy part. What breaks restores is everything around it: the role that owns the tables, the extension the schema depends on, the engine version the dump format expects, and the connection string the application reads from a config file you forgot to back up.

Restore in a controlled order

Create the target with a compatible engine version first. Check what the backup came from and what you're restoring onto:

```
psql --version
# psql (PostgreSQL) 16.4
pg_restore --list shopdb-2026-08-03.dump | head -3
# ; Archive created at 2026-08-03 02:00:11
# ;     dbname: shopdb
```

Restoring a dump from a newer major version into an older server is not supported. The reverse direction — old dump, newer server — is the normal upgrade path.

Restore roles and schema as required, then load data:

```
sudo -u postgres psql -f pg-globals-2026-08-03.sql          # roles first
sudo -u postgres createdb shopdb
sudo -u postgres pg_restore --dbname=shopdb shopdb-2026-08-03.dump
```

Roles come first for a reason. Load data before the owning role exists and the restore floods you with `role "shopapp" does not exist` errors, leaving objects owned by the wrong user.

Then apply only documented recovery steps — the runbook, not improvisation — and connect the application with isolated credentials: the application's own database user, not `postgres`. If the app only works when connected as a superuser, you've hidden a permissions problem you'll rediscover in production.

Validate before cutover

A restore that completes is not a restore that worked. Run validation queries with expected answers:

```
sudo -u postgres psql shopdb -c "SELECT count(*) FROM orders;"
#  count
# -------
#  48213
```

Compare against the count recorded when the backup ran. Check the newest row's timestamp against your recovery point. Then start the service against the restored database and run one real user-visible action before you send traffic at it. Only after that evidence do you make the cutover decision.

Write the exact restore order for one database-backed application. Include version checks, credentials, migrations, validation queries, and the cutover decision.

Complete a recovery drill

Rebuild a disposable server from trusted inputs and improve the plan from measured restore evidence.

A full drill reveals dependencies that file-level tests miss: DNS, certificates, packages, service units, firewall policy, and external credentials.

You can restore every file perfectly and still have no working service, because the service was never just files. It was also a DNS record, a TLS certificate, a package list, a systemd unit, a firewall rule, and an API key that lived only in someone's password manager. The only way to find those gaps is to rebuild for real, before the day you have no choice.

Run the drill

Start from a clean host. A fresh VM, not the production server — the drill must prove you can come back from nothing. Restore or rebuild each layer in runbook order, and keep a timeline as you go:

```
14:02  fresh Ubuntu 24.04 VM provisioned
14:05  restic repo credentials retrieved from password manager
14:11  packages installed from runbook list
14:19  /etc/nginx and TLS certs restored
14:24  BLOCKED: nginx unit needs /var/www/shop -- not in runbook
14:31  postgres roles + dump restored, counts match
```

```
14:58  service up, responds locally
```

Every `BLOCKED` line is the drill paying for itself. Record every undocumented dependency the moment it bites you.

Verify the application externally — from another machine, the way a user would, not with a local check that skips DNS, TLS, and the firewall:

```
curl -sS -o /dev/null -w "%{http_code} %{time_total}s\n" --resolve shop.flaviocopes.com:443:2
# 200 0.412s
```

The `--resolve` flag points the request at the drill host without touching real DNS, so you can test the full HTTPS path while production stays live.

Turn evidence into improvement

Produce a short report with actual recovery time, latest usable recovery point, missing inputs, and the next improvement owner. Compare the measured numbers against the RPO and RTO you committed to earlier in this course — a 4-hour promise and a 6-hour drill is a finding, not a failure.

Update the runbook while evidence is fresh, the same day. And when a full rebuild is genuinely too expensive this quarter, a **tabletop drill** — walking the runbook step by step on paper, checking each input exists — still catches missing credentials and stale instructions. Cheaper than a real drill, infinitely better than nothing.

Run or tabletop a complete recovery. Produce a short report with actual recovery time, latest usable recovery point, missing inputs, and the next improvement owner.

Check your understanding: restore and recover

Check restore isolation, integrity, database recovery, boot recovery, documentation, and final review.

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-storage-backups/.