

SERVER MANUAL

FREE SAMPLE · CHAPTERS 1-4

Ubuntu Server, Distilled

— *Not Brewed* —

Ubuntu Server 26.04 LTS — "Resolute Raccoon"

The first four chapters of Ubuntu Server, Distilled — complete and unabridged: install it, lock down users and access, get it on the network, and harden it. The full book continues from Chapter 5 through backups, containers, monitoring, and a full quick-reference cheat sheet.

 Step-by-step and practical Secure by default Containers made simple Monitor what matters Back up and recover Quick-reference cheat sheet**4**

CHAPTERS HERE

16

CHAPTERS TOTAL

2031

LTS SUPPORT UNTIL

Free sample · Chapters 1-4 of 16 · get the full book at SplitZrate.com

PRACTICAL RELIABLE NO FLUFF

Contents

01

Installation & First Boot [IN THIS SAMPLE](#)

03

Networking Essentials [IN THIS SAMPLE](#)

05

Package Management & Updates

07

Storage & File Systems

09

Containers & Virtualization

11

Monitoring & Logging

13

Troubleshooting & Diagnostics

15

Running LLMs on Ubuntu Server

02

Users, Permissions & Access Control

[IN THIS SAMPLE](#)

04

Security Hardening [IN THIS SAMPLE](#)

06

Ubuntu Pro & Long-Term Support
Essentials

08

systemd, Services & Processes

10

Web Server & Reverse Proxy Essentials

12

Backup & Recovery

14

Automation Basics: Ansible & Config
Management

16

Quick Reference & Cheat Sheet

This sample contains Chapters 1–4, complete and unabridged. Chapters 5–16 (grayed out above) are in the full book — get it at [SplitZrate.com](#).

1 Installation & First Boot

Everything you need to go from a blank machine to a running, updated Ubuntu Server 26.04 LTS system — choosing an install method, meeting the hardware requirements, working through the installer, and the checklist to run right after first login.

1.1 Why this chapter starts here

Ubuntu Server ships a new Long-Term Support release every two years, and 26.04 LTS (codenamed "Resolute Raccoon") is the version this manual is built around. Canonical backs each LTS with five years of standard support and the option to extend that to ten years through Ubuntu Pro, so whatever you install today is realistically what you will still be running — and patching — years from now. That makes the installation stage worth slowing down for: decisions made here, particularly around disk layout and whether the install is done by hand or scripted, are the ones that are most annoying to undo later.

This chapter treats installation as a means to an end. It does not walk through every screen of the installer pixel by pixel — Canonical's own reference already does that well. Instead, it focuses on the decisions that actually matter: which install method fits your situation, what to double-check before you start, and what a sane first hour with a fresh server looks like.

1.2 Choosing an installation path

There are three realistic ways to get Ubuntu Server onto a machine, and picking the right one up front saves time later:

- **Interactive install from the live-server ISO.** You boot from USB or virtual media and answer the installer's questions (Subiquity) one screen at a time. Best for a single server, a lab machine, or the first time you're setting something up and still want to see every option.
- **Cloud/VM images.** Pre-built `.img` files for platforms like AWS, Azure, and generic KVM/QEMU. There is no installer to run at all — the image boots directly, and first-boot configuration (user, SSH keys, hostname) is handled by `cloud-init` reading data injected by the hypervisor or cloud provider.
- **Autoinstall.** The same Subiquity installer as the interactive path, but driven entirely by a YAML answer file instead of a human. This is the right choice the moment you need to build more than a couple of identical servers, or want installs to be reproducible and reviewable in version control.

The rest of this chapter covers the interactive install in detail, since it teaches you what every field actually means, then shows how the same choices map onto an autoinstall file once you're ready to stop clicking through them by hand.

NOTE

If you only ever deploy into a cloud provider's own image catalogue, you can largely skip the interactive-install walkthrough below and jump straight to **1.7 Automating deployment with autoinstall** — the concepts of user creation, SSH access, and first-boot checks still apply.

1.3 Hardware and architecture requirements

Ubuntu Server supports four 64-bit architectures: **amd64** (Intel/AMD), **arm64**, **ppc64el** (POWER8/POWER9), and **s390x** (IBM Z / LinuxONE). For a general-purpose install, the practical minimums and realistic recommendations look like this:

Resource	Bare minimum	Realistic for production
RAM	2 GB	4 GB or more
Disk	5 GB	20 GB+ (more for containers/databases)
CPU	1 core	2 cores or more
Network	DHCP-reachable NIC	Static IP or DHCP reservation

26.04 LTS also tightens the baseline CPU generation each architecture needs, which mainly matters if you're installing onto older hardware or building your own images:

- **amd64 cloud images** now require the AMD64v3 instruction set — older CPUs such as Intel Sandy Bridge/Ivy Bridge era chips are no longer supported by the official cloud images (physical-media installs are less strict, but it's worth checking your CPU generation first).
- **RISC-V** images require the RVA23S64 profile.
- **s390x** requires at minimum an IBM Z generation z15 machine.

If you're planning an in-place upgrade from an older release rather than a fresh install, check this now rather than after the upgrade has already started.

1.4 Getting your installation media ready

Download the live-server ISO for amd64 from releases.ubuntu.com (other architectures have their own platform-specific images, several of which are covered in the reference chapter). Before writing it to media, verify the checksum — this catches both corrupted downloads and, more importantly, tampered images:

```
# download the ISO and the published checksums, then verify
sha256sum -c SHA256SUMS --ignore-missing
```

Once verified, write the image to a USB stick. On Linux or macOS, `dd` does the job directly; on Windows, tools like Rufus or balenaEtcher are the easier route.

```
# replace /dev/sdX with your USB device – double check with lsblk first
sudo dd if=ubuntu-26.04-live-server-amd64.iso of=/dev/sdX bs=4M status=progress
conv=fsync
```

WARNING

`dd` writes directly to a block device and does not ask for confirmation. Run `lsblk` immediately before this command and triple-check the device name — pointing it at the wrong disk destroys that disk's data with no prompt and no undo.

1.5 Working through the installer

Boot from the USB stick and the installer (Subiquity) will start with a language screen. The flow below covers each stage in the order you'll see it; on most hardware, the defaults are sensible and you can move quickly through the first few screens.

Language, keyboard, and network

Pick your language and keyboard layout, then the installer attempts to bring up networking automatically via DHCP on any wired interface it finds. If you need a static address instead of DHCP, this is the screen to configure it — you can also skip networking entirely and configure it after install using Netplan, which is covered in Chapter 3.

Proxy and mirror

Leave these on their defaults unless your network requires an outbound proxy or you have an internal package mirror you want to point at instead of Canonical's archive.

Storage layout

For a first server, "use an entire disk" with the guided storage layout is the simplest and safest option — it sets up LVM automatically, which makes it straightforward to resize volumes or take snapshots later. If you need custom partitioning (separate `/var`, ZFS, encrypted volumes, or multiple disks), the installer's manual mode exposes those options, but only reach for it once you actually have a reason to — the defaults serve the large majority of servers well.

Profile setup

You'll be asked for your name, a server hostname, a username, and a password. One change worth flagging in 26.04: the account is created earlier in the installer flow than in previous releases, which fixed a class of crashes that used to happen specifically in virtual machines during this step. Functionally, the fields you fill in are unchanged.

Ubuntu Pro (optional)

You'll be offered the option to attach an Ubuntu Pro subscription here. It's entirely optional for a standard install and can always be attached later with `sudo pro attach`, so feel free to skip it if you don't have a token handy.

SSH setup

This screen is easy to skip past but matters a lot in practice. Tick the box to install `openssh-server` — without it, you'll be stuck at the physical console (or your hypervisor's console) until you install it manually. If you use GitHub or Launchpad, the installer can import your public key directly by username, which saves a manual `authorized_keys` edit later:

```
ssh-import-id gh:your-github-username
```

Featured server snaps

The installer offers a short list of common snaps (Docker, Nextcloud, database servers, and similar). Unless you already know you want one, it's usually cleaner to skip this and install what you need deliberately afterwards — it keeps the base system minimal and makes it obvious later what was intentionally installed versus what came bundled by default.

Installation and reboot

From here the installer just runs — you'll see log output as packages are unpacked and configured. When it finishes, remove the installation media and select restart. Log in with the username and password from the profile step.

1.6 Should you use the minimal install?

Alongside the standard install, the installer offers a minimal variant that strips out a number of packages that are normally installed by default but rarely used on a server (things like extra language packs and some peripheral firmware). For production systems, this is generally worth choosing: fewer preinstalled packages means a smaller attack surface, fewer things to patch, and a faster first boot.

TIP

If you're not sure whether the minimal install will be missing something you need, it almost certainly won't — anything left out is just as easy to `apt install` afterwards. Treat it as the safer default for anything going into production, and reserve the full install for machines you're going to poke around on interactively.

1.7 Automating deployment with autoinstall

Autoinstall reuses the exact same Subiquity installer, but reads its answers from a YAML file instead of prompting a human. Once you've done a handful of interactive installs and know what your answers should be, this is the natural next step — it turns installation into something you can version-control, review, and repeat identically across any number of machines.

A minimal autoinstall file looks roughly like this:

```
#cloud-config
autoinstall:
  version: 1
  locale: en_US.UTF-8
  keyboard:
    layout: us
  network:
    network:
      version: 2
      ethernet:
        eth0:
          dhcp4: true
  storage:
    layout:
      name: lvm
  identity:
    hostname: srv-01
    username: ops
    password: <SHA-512 crypt hash>
  ssh:
    install-server: true
    allow-pw: false
    authorized-keys:
      - ssh-ed25519 AAAA...
  packages:
    - openssh-server
```

Serve this file over HTTP (a simple `python3 -m http.server` is enough for testing) or place it on a USB stick alongside the ISO, then point the installer at it via the kernel command line. Once the install finishes, you can confirm cloud-init actually completed successfully before treating the machine as ready:

```
sudo cloud-init status --wait
```

That single command is worth building into any provisioning script — it blocks until cloud-init finishes and returns a non-zero exit code if something failed, which is exactly what you want to gate the rest of an automated deployment on.

1.8 First boot checklist

A freshly installed server is rarely a finished one. Before handing it off — to yourself, a teammate, or an automated fleet — run through this short checklist:

```
# 1. Bring packages fully up to date
sudo apt update && sudo apt full-upgrade -y

# 2. Set the correct timezone
sudo timedatectl set-timezone Europe/Stockholm
timedatectl status

# 3. Confirm hostname and /etc/hosts agree with each other
hostnamectl
cat /etc/hosts

# 4. Check disks and mounts look as expected
lsblk
df -h

# 5. Look for anything that failed to start
systemctl --failed

# 6. Enable a baseline firewall (details in Chapter 4)
sudo ufw allow OpenSSH
sudo ufw enable
```

Three 26.04-specific behaviors are worth knowing about at this stage, since they can otherwise cause confusing surprises later:

- **/tmp is now tmpfs by default** — it lives in RAM and is capped by available memory. Workloads that write large temporary files (big compilation jobs, some backup tools) should be pointed at a disk-backed directory instead of relying on `/tmp`.
- **Removable media now mounts under `/run/media`** rather than `/media`. Any older scripts or automation that hardcode the old path will need updating.
- **cgroup v1 support has been removed entirely** (systemd 259). This only bites if you're upgrading an older system rather than installing fresh — check with `stat -fc %T /sys/fs/cgroup/` before upgrading; anything other than `cgroup2fs` means you have workloads still depending on v1 that need migrating first.

CHAPTER 1 — QUICK RECAP

- Interactive install for one-offs, cloud images for hyperscalers, `autoinstall` once you're repeating yourself.
- Verify ISO checksums before writing media: `sha256sum -c SHA256SUMS`
- Choose the **minimal install** for production systems by default.
- Always tick **install OpenSSH server** during setup, or you're stuck on console.
- Validate automated installs with `cloud-init status --wait`
- Run the first-boot checklist before treating any new server as ready: updates, timezone, hostname, disks, failed units, firewall.

2 Users, Permissions & Access Control

Who can log in, what they can touch, and how to prove it's really them. This chapter covers the account and permission model you'll be relying on for everything else in this book — get it right early and most of the rest gets simpler.

2.1 The account model on a fresh server

A default Ubuntu Server install gives you one non-root account — the one you created during installation — and membership in the `sudo` group, which lets you run commands as root by prefixing them with `sudo` and re-entering your own password. The root account itself exists but has no usable password set, so you can't log into it directly; everything privileged happens through `sudo`. This is deliberate and worth keeping: it means every privileged action is tied to a named human and logged, instead of an anonymous "root did it."

2.2 Creating and managing users

Two commands create users: `adduser` (interactive, friendly, Debian/Ubuntu-specific) and `useradd` (lower-level, part of the standard Linux toolset, needs more flags to do the same job). For day-to-day admin work, `adduser` is the one to reach for:

```
# create a new user with a home directory, prompts for password and details
sudo adduser maria

# add an existing user to the sudo group
sudo usermod -aG sudo maria

# see which groups a user belongs to
groups maria

# remove a user (keeps home directory by default)
sudo deluser maria

# remove a user AND their home directory / mail spool
sudo deluser --remove-home maria
```

WARNING

`usermod -aG` — note the `-a` (append). Leaving it off replaces the user's group list entirely with whatever you specify, silently dropping them from every other group they were in, including possibly `sudo` itself. Always use `-aG`, never just `-G`, when adding to a group.

2.3 Groups and why they matter

Every user has one primary group and can belong to any number of supplementary groups. Group membership is the standard way to grant access to a resource without editing permissions file by file: add a user to the `docker` group and they can talk to the Docker daemon; add them to `adm` and they can read system logs; create a shared group for a project directory and everyone in it gets consistent access. Prefer groups over one-off `chmod` changes wherever more than one person needs the same access — it's far easier to audit "who's in this group" than to reconstruct why a file has the permissions it has.

2.4 sudo in depth

The rules for who can run what as root live in `/etc/sudoers` and the drop-in files under `/etc/sudoers.d/`. Never edit these with a normal text editor — always use `visudo`, which validates the syntax before saving.

WARNING

A syntax error in `/etc/sudoers` saved outside `visudo` can break `sudo` entirely — including for you, mid-session. On a remote box with no other privileged access, that can mean a full reinstall to recover. `visudo` refuses to save a broken file, which is the entire reason it exists — use it every time.

A typical sudoers drop-in, added as its own file under `/etc/sudoers.d/` rather than editing the main file directly, looks like this:

```
/etc/sudoers.d/maria
maria ALL=(ALL) ALL
```

You'll sometimes see `NOPASSWD:` added to a rule, which lets a command run via sudo without a password prompt. It's occasionally justified for a single, narrowly scoped automation command — never for blanket `ALL` access on an interactive account, since it turns a compromised session or leaked SSH key into instant root with no second factor at all.

Two commands worth knowing for troubleshooting sudo itself: `sudo -l` lists what the current user is allowed to run, and every sudo invocation is logged to `/var/log/auth.log` — useful both for debugging "why can't I run this" and for after-the-fact auditing of who ran what.

2.5 SSH key-based access, per user

Each user who logs in over SSH keeps their authorized public keys in `~/.ssh/authorized_keys`. The easiest way to install a key is `ssh-copy-id` from the client side:

```
ssh-copy-id maria@server.example.com
```

NOTE

SSH is strict about permissions on this file and its parent directory — it will silently refuse a key if `~/.ssh` isn't `700` or `authorized_keys` isn't `600`. If a key that should work doesn't, check permissions before anything else: `chmod 700 ~/.ssh && chmod 600 ~/.ssh/authorized_keys`.

2.6 File permissions refresher

Every file has three permission sets — owner, group, and everyone else — each with read, write, and execute bits. `ls -l` shows them as a string like `rwxr-xr--`; the same permissions can be set numerically, where read=4, write=2, execute=1 are summed per group:

Numeric	Symbolic	Meaning
755	rwxr-xr-x	Owner: full control. Group & others: read/execute — typical for scripts and directories.
644	rw-r--r--	Owner: read/write. Group & others: read-only — typical for regular files.
700	rwx-----	Owner only, no access for anyone else — SSH keys, private data.
600	rw-----	Owner read/write only, no execute, no one else — credentials, private keys.

```
# set numerically
chmod 644 report.txt

# set symbolically – add execute for the owner
chmod u+x deploy.sh

# change ownership (user and group)
sudo chown maria:maria /srv/app
```

WARNING

`chmod -R` and `chown -R` apply recursively without asking twice. Run them against a system directory like `/`, `/etc`, or `/usr` — even by a typo in the path — and you can break every service on the box in one command. Double-check the path, and consider a dry run with `find /path -type f` first if you're unsure exactly what a recursive command will touch.

2.7 ACLs: when owner/group/other isn't enough

Standard permissions only describe one owner and one group. When you need, say, one extra user to have write access to a directory they don't otherwise own, Access Control Lists (ACLs) let you grant that without changing ownership or creating a new group:

```
# give user 'maria' read+write on a directory, without changing its owner
sudo setfacl -m u:maria:rwX /srv/shared

# check what's set
getfacl /srv/shared
```

Reach for ACLs when you have a genuinely one-off access requirement; for anything more than one or two exceptions, a dedicated group is still easier to reason about later.

2.8 Service accounts and least privilege

Applications and daemons shouldn't run as your personal user, and shouldn't run as root unless they genuinely need root. Package installs typically create a dedicated system account for this automatically (e.g. `www-data` for the web server), with no login shell:

```
# create a system account with no login shell and no home directory prompts
sudo adduser --system --no-create-home --group appsvc
```

If a service is compromised, running it under a dedicated, unprivileged account limits the blast radius to whatever that account can touch — which, done right, is very little.

2.9 Locking down root

Since root has no usable password by default, the main remaining risk is root logging in over SSH. Confirm it's disabled — and while you're in the SSH config, disable password authentication in favor of keys only, which closes off brute-force login attempts entirely:

```
/etc/ssh/sshd_config
PermitRootLogin no
PasswordAuthentication no
```

WARNING

Before restarting SSH with `PasswordAuthentication no`, make absolutely sure your own SSH key already works — test logging in with it in a second terminal *before* closing your current session. If key auth isn't actually working yet, you'll lock yourself out the moment the service restarts, with no password fallback. Chapter 4 covers SSH hardening in more depth, including fail2ban and changing the default port.

```
# Debian/Ubuntu ships the unit as "ssh", not "sshd" — "sshd" fails with "Unit
sshd.service not found"
sudo systemctl restart ssh
```

CHAPTER 2 — QUICK RECAP

- Use `adduser` for people, `adduser --system` for services.
- Always `usermod -aG`, never `-G` — the `-a` matters.
- Edit sudo rules only with `visudo`, never a plain text editor.
- SSH needs `~/.ssh` at 700 and `authorized_keys` at 600 or it silently rejects keys.
- Never run `chmod -R` / `chown -R` without triple-checking the path.
- Test SSH key login in a second session before disabling password authentication.

3 Networking Essentials

How Ubuntu Server configures its network interfaces, how to move from DHCP to a static address without cutting yourself off, and the handful of commands that answer "why can't this machine talk to the network" faster than anything else.

3.1 Netplan: the config layer you actually edit

Modern Ubuntu configures networking through **Netplan** — a YAML front-end that generates the underlying configuration for whichever backend is actually managing the interface (`systemd-networkd` on servers, NetworkManager on desktops). You don't touch the backend directly; you write a small YAML file describing what you want, and Netplan translates it. Configuration files live in `/etc/netplan/`, usually as a single file named something like `00-installer-config.yaml` left behind by the installer.

3.2 Reading and writing a Netplan file

A DHCP configuration — the default after a standard install — looks like this:

```
/etc/netplan/00-installer-config.yaml
network:
  version: 2
  ethernets:
    eth0:
      dhcp4: true
```

Switching that same interface to a static address:

```
network:
  version: 2
  ethernets:
    eth0:
      dhcp4: false
      addresses:
        - 192.168.1.50/24
      routes:
        - to: default
          via: 192.168.1.1
      nameservers:
        addresses: [1.1.1.1, 9.9.9.9]
```

YAML is whitespace-sensitive — two-space indentation, no tabs — and a single misplaced space will make Netplan reject the file outright, which is at least a safe failure mode. Apply a change with:

```
sudo netplan apply
```

WARNING

If you're connected over SSH and the new configuration is wrong — bad gateway, wrong subnet, typo'd address — `netplan apply` can cut your session instantly with no way back in except physical/console access. Use `netplan try` instead whenever you're changing network settings remotely: it applies the change, then automatically reverts after 120 seconds unless you press Enter to confirm it worked. This single habit prevents the single most common way people accidentally lock themselves out of a remote server.

```
sudo netplan try
```

3.3 Checking connectivity

A small set of commands covers the large majority of "is the network working" questions:

```
# interface addresses and state
ip addr show

# routing table — is there a default route?
ip route

# what's listening on which ports
ss -tulpn

# basic reachability
ping -c 4 1.1.1.1

# where does traffic actually go — useful when ping works but a service doesn't
traceroute example.com # or: mtr example.com
```

`ip` and `ss` have replaced the older `ifconfig` and `netstat` — those still work if installed via the `net-tools` package, but there's no reason to reach for them on a fresh system.

3.4 DNS resolution

DNS is handled by `systemd-resolved`. `/etc/resolv.conf` is a symlink managed automatically — don't hand-edit it, since the next Netplan apply or reboot will overwrite whatever you put there. Set nameservers through Netplan (as in the example above) instead. To check what's actually being used and debug lookups:

```
resolvectl status
resolvectl query example.com
```

3.5 Static IP or DHCP reservation?

Both give a server a predictable address, but they solve it in different places. A DHCP reservation (configured on the router/DHCP server, keyed to the machine's MAC address) keeps the server's own config simple and centralizes all address assignment in one place — useful if you manage several machines and want a single source of truth. A static address baked into Netplan works even if the DHCP server is down or the machine is moved to a network without one. For a home lab or small setup, DHCP reservations are usually the more maintainable choice; for anything provisioned via `autoinstall` (Chapter 1) where you want the config to travel with the machine, static addressing in Netplan is more self-contained.

3.6 Multiple interfaces: bonding and VLANs

Servers with more than one NIC can combine them into a bond for redundancy or extra throughput, or split a single NIC into tagged VLANs. Both are configured the same way — more YAML, not more tools:

```
# two NICs combined into an active-backup bond
network:
  version: 2
  ethernet:
    eth0: {}
    eth1: {}
  bonds:
    bond0:
      interfaces: [eth0, eth1]
      addresses: [192.168.1.50/24]
      parameters:
        mode: active-backup

# a tagged VLAN on top of an existing interface
vlan:
  vlan20:
    id: 20
    link: eth0
    addresses: [10.0.20.5/24]
```

Both are worth knowing exist; neither is something most single-NIC home or small-office servers will ever need.

3.7 Where the firewall fits in

Netplan controls connectivity — whether an interface has an address and a route at all. It has nothing to do with which ports are allowed through once connectivity exists; that's the firewall's job (`ufw`), covered in full in Chapter 4. If a service is unreachable, check both layers: `ip addr` and `ping` for connectivity, then `sudo ufw status` for whether the port is actually allowed.

NOTE

File permissions matter here too: Netplan will warn (and as of recent releases, refuse) if a config file under `/etc/netplan/` is world-readable, since it may contain Wi-Fi passwords or other sensitive values. Keep these files at the default `600` permissions.

CHAPTER 3 — QUICK RECAP

- Edit YAML under `/etc/netplan/`; never hand-edit `/etc/resolv.conf`.
- Always prefer `netplan try` over `netplan apply` on a remote session.
- `ip addr`, `ip route`, `ss -tulpn`, `resolvectl status` cover most troubleshooting.
- Connectivity (Netplan) and reachability-through-the-firewall (`ufw`, Ch.4) are two separate layers — check both.

4 Security Hardening

A realistic security baseline for a server exposed to the internet: firewall, SSH, automatic updates, and the handful of habits that stop the vast majority of automated attacks without turning administration into a chore.

4.1 What you're actually defending against

For the overwhelming majority of small servers, the realistic threat isn't a targeted attacker — it's automated scanning: bots that sweep the entire internet for open ports, default credentials, and unpatched software within minutes of a server coming online. The goal of this chapter isn't to make your server unbreakable — it's to get it out of the "trivially exploitable by automation" category, which covers almost everything that actually happens to small, unremarkable servers.

4.2 The firewall: ufw

`ufw` (Uncomplicated Firewall) is a friendly front-end over `iptables` / `nftables`. Default policy should be deny-everything-incoming, allow-everything-outgoing, with explicit exceptions for the services you actually run:

```
sudo ufw default deny incoming
sudo ufw default allow outgoing
sudo ufw allow OpenSSH
sudo ufw enable
```

WARNING

Order matters. If you run `ufw enable` before allowing SSH, the very next packet — including your current SSH session — gets dropped, and you're locked out with no way back in except the console. Always allow SSH (or whatever port you actually use for remote access) *before* enabling the firewall, every single time.

```
# allow a specific service by name (reads /etc/services)
sudo ufw allow http
sudo ufw allow 443/tcp

# restrict a port to a single source IP — e.g. an admin VPN
sudo ufw allow from 10.0.0.5 to any port 22

# check what's actually active
sudo ufw status verbose
```

4.3 SSH hardening

Chapter 2 already covered the two changes that matter most — key-only authentication and disabling root login. A few more are worth adding to `/etc/ssh/sshd_config`:

```
PermitRootLogin no
PasswordAuthentication no
X11Forwarding no
MaxAuthTries 3
```

Moving SSH off port 22 is common advice and worth being honest about: it is not real security, it's noise reduction. It won't stop a targeted attacker for more than a few seconds of port-scanning, but it does dramatically cut the volume of automated credential-stuffing noise hitting your logs, which makes genuine issues easier to spot. If you do change it, update the firewall rule to match *before* restarting SSH — same lockout risk as above, just with an extra step to get wrong.

WARNING

Changing `Port` in `sshd_config` without first running `sudo ufw allow <new-port>/tcp` will lock you out the moment the service restarts — the firewall will happily block the new port while the old one no longer has anything listening on it. Add the new rule, verify it, restart SSH, confirm you can connect on the new port in a second session, and only then remove the old rule.

4.4 fail2ban: banning brute-force attempts automatically

fail2ban watches log files for repeated failed login attempts and temporarily firewalls the offending IP. It pairs naturally with ufw and needs almost no configuration to be useful:

```
sudo apt install fail2ban
sudo cp /etc/fail2ban/jail.conf /etc/fail2ban/jail.local
```

Edit `jail.local` (never the original `.conf` — it gets overwritten on package updates) to enable the SSH jail and set sane thresholds:

```
/etc/fail2ban/jail.local
[sshd]
enabled = true
maxretry = 5
bantime = 1h
findtime = 10m
```

```
sudo systemctl restart fail2ban
sudo fail2ban-client status sshd
```

NOTE

If you administer the server from a small number of known locations, add your own IPs to `ignoreip` in `jail.local`. It's a rite of passage to lock yourself out with your own too-aggressive fail2ban configuration after one too many typo'd passwords — `bantime` is how long you'll be waiting it out.

4.5 Keeping the system patched automatically

Unpatched, known vulnerabilities are how most opportunistic compromises actually happen — not zero-days. The `unattended-upgrades` package applies security updates on a schedule without you needing to remember:

```
sudo apt install unattended-upgrades
sudo dpkg-reconfigure --priority=low unattended-upgrades
```

This installs security patches automatically; it's deliberately more conservative about feature updates, which you still apply by hand. Chapter 5 covers package management and update strategy in full, including how to review what unattended-upgrades has done via its log at

```
/var/log/unattended-upgrades/.
```

4.6 AppArmor: mandatory access control, already on

Ubuntu ships with AppArmor enabled by default, confining many common services (like `nginx` and `mysqld`) to only the files and capabilities their profile allows — a second line of defense if a service itself gets compromised. You rarely need to configure it, but it's worth knowing it's there when something is inexplicably denied access to a file it should be able to read:

```
sudo aa-status # see loaded profiles and their mode
sudo journalctl -xe | grep apparmor # check for recent denials
```

Profiles run in either **enforce** mode (blocks and logs violations) or **complain** mode (logs only, doesn't block) — useful for testing a new profile without breaking the service while you work out what it actually needs.

4.7 Reducing what's exposed in the first place

The best-defended port is the one that isn't open at all. Two habits keep exposure to a minimum: choosing the minimal install (Chapter 1) so fewer services exist to misconfigure, and periodically checking what's actually listening:

```
sudo ss -tulpn
```

Anything listening that you don't recognize or don't need is worth investigating and either firewalling off or disabling outright with `sudo systemctl disable --now <service>`.

WARNING

Before disabling a service you don't recognize, confirm what it actually is — some unfamiliar-looking listeners are load-bearing (e.g. `systemd-resolved` on port 53, or a database another application depends on). Check `systemctl status <name>` and search for the service before switching it off.

4.8 A baseline checklist for every new server

```
# 1. Firewall — allow SSH first, always
sudo ufw allow OpenSSH && sudo ufw enable

# 2. Key-only SSH, no root login (Chapter 2)
# 3. fail2ban for SSH
sudo apt install fail2ban

# 4. Automatic security patches
sudo apt install unattended-upgrades

# 5. Confirm nothing unexpected is listening
sudo ss -tulpn

# 6. Confirm AppArmor is active
sudo aa-status
```

CHAPTER 4 — QUICK RECAP

- Always `ufw allow OpenSSH` before `ufw enable` — no exceptions.
- Changing the SSH port needs the new firewall rule added and verified *before* restarting SSH.
- fail2ban for automated ban-on-brute-force; add your own IP to `ignoreip` first.
- `unattended-upgrades` for hands-off security patching.
- AppArmor is on by default — check `aa-status` when something is denied access unexpectedly.
- Periodically audit with `ss -tulpn`; disable what you don't recognize (after confirming what it is).

That's the first four chapters.

Twelve more chapters pick up right where this sample leaves off — from package management and Ubuntu Pro through containers, monitoring, backups, troubleshooting, automation, and a full quick-reference cheat sheet.

STILL IN THE FULL BOOK

- 5. Package Management & Updates
- 6. Ubuntu Pro & Long-Term Support Essentials
- 7. Storage & File Systems
- 8. systemd, Services & Processes
- 9. Containers & Virtualization
- 10. Web Server & Reverse Proxy Essentials
- 11. Monitoring & Logging
- 12. Backup & Recovery
- 13. Troubleshooting & Diagnostics
- 14. Automation Basics: Ansible & Config Management
- 15. Running LLMs on Ubuntu Server
- 16. Quick Reference & Cheat Sheet

GET THE FULL BOOK

Available now at SplitZrate.com — same practical, no-fluff style as the four chapters you just read, all 16 chapters, updates included as 26.04 LTS evolves.