Comparative Linux Distributions

A Complete 12-Chapter Operating Systems Course

Topics covered:

Debian & Ubuntu · Fedora & RHEL · Arch · Alpine

Package managers compared directly · release models · init systems & defaults

Governance & support models · Raspberry Pi OS

Capstone: the same service deployed on three real distributions

Exercises: 36 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 12

What Actually Makes a Distribution "Different"

Comparative Linux Distributions

Chapter 1 · What Actually Makes a Distribution "Different"

Installing and Configuring Linux already teaches how to install and configure a Linux system well — generically, without dwelling on what makes one distribution genuinely different from another. *Windows 11 Fundamentals 1*'s own OS-landscape table went further, positioning "Linux" as a single row: CLI-first by tradition, with a GUI desktop environment as one optional layer on top. This course exists specifically to open that one row up — Linux isn't one philosophy wearing different outfits, it's several genuinely different philosophies, and this chapter names exactly what actually varies between them.

Five Components, Not One "Look"

A distribution is a specific combination of five real components — and the one most beginners notice first is, genuinely, the least defining:

- **Kernel** — almost every mainstream distribution runs the same Linux kernel (with its own specific version and patch set), so this is rarely the real differentiator between them.
- **Package manager & format** — how software actually gets installed, removed, and tracked; a genuinely load-bearing difference this course's own Chapter 6 covers directly, side by side.
- **Init system** — what actually starts services at boot and supervises them afterward; overwhelmingly `systemd` today, with one real, living exception covered in Chapter 5 and Chapter 8.
- **Release model** — point releases, a continuous rolling model, or a multi-year enterprise support window; covered in full in Chapter 7.
- **Default desktop environment** — GNOME, KDE Plasma, XFCE, or none at all — the single most swappable, least defining component of the five, despite being the first thing anyone actually sees.

"IT'S BASICALLY ALL JUST LINUX UNDERNEATH" IS THE EXACT ASSUMPTION THIS COURSE EXISTS TO CORRECT

A command copied from a support forum post — `sudo apt install <package>`, say — assumes a specific package manager that simply doesn't exist on every distribution. Running it on Arch or Alpine won't produce a helpful error pointing at the right tool; it will just fail, because the assumption baked into that one command was never universal in the first place.

Why the Desktop Environment Is the Weakest Signal

Two machines running GNOME can be running genuinely different distributions underneath — different package managers, different init systems, different release philosophies — while looking nearly identical on screen. Conversely, the same distribution can run several different desktop environments, or none. Judging "how different is this distro" by its default wallpaper and icon theme is judging the single most interchangeable piece of the whole picture.

A First Look at the Four Families This Course Covers

FAMILY	PACKAGE MANAGER	INIT SYSTEM	RELEASE MODEL
Debian / Ubuntu	apt (.deb)	systemd	Point release / LTS cadence
Fedora / RHEL	dnf (.rpm)	systemd	Fast cycle / multi-year enterprise
Arch	pacman	systemd	Continuous rolling release
Alpine	apk	OpenRC	Point release, minimal-first

Notice that three of the four share the same init system — the real, still-living exception is exactly where Chapter 5 (Alpine) and Chapter 8 (init systems generally) pick the story back up.

> **IDENTIFYING A DISTRO RELIABLY, FROM THE INSIDE**
>
> `cat /etc/os-release` works consistently across virtually every modern distribution and reports the exact family and version — far more reliable than guessing from a desktop's own visual theme, or assuming a colleague's "it's just Ubuntu" description is complete enough to act on.

This Course's Own Roadmap

Debian/Ubuntu and Fedora/RHEL next, then Arch, then Alpine — one chapter each, going deep on what genuinely distinguishes it. Then package managers, release models, init systems, and governance compared directly across all four at once. Raspberry Pi OS gets its own chapter as a genuine Debian derivative worth understanding on its own terms. A practical decision framework and a real, worked capstone — the same service, deployed three different ways — close out the course.

Hands-On Exercises

EXERCISE 1

A colleague says two machines are "basically the same Linux" because both run GNOME. Using this chapter's own five-component list, explain why that observation alone doesn't actually establish much.

View solution

EXERCISE 2

A forum post's suggested fix begins with "just run sudo apt install...". Explain, using this chapter's own warn-box, why this instruction can't simply be assumed to work on any given Linux machine.

View solution

EXERCISE 3

Explain why this chapter's own compare-table pointing out that three of the four families share the same init system is still a useful observation, rather than undermining the idea that these are four genuinely different distributions.

View solution

CHAPTER 1 QUICK REFERENCE

- **Five real components:** kernel, package manager, init system, release model, default desktop environment
- The desktop environment is the most swappable and least defining of the five — despite being the most visible
- `cat /etc/os-release` reliably identifies a distro's actual family and version
- Debian/Ubuntu, Fedora/RHEL, and Arch all use systemd; Alpine's OpenRC is the real living exception (Chapters 5 & 8)
- Assuming a distro-specific command (like `apt install`) works everywhere is a real, common mistake this course exists to prevent
- **Next chapter:** Debian & Ubuntu — Stability-First

CHAPTER 2 OF 12

Debian & Ubuntu — Stability-First

Comparative Linux Distributions

Chapter 2 · Debian & Ubuntu — Stability-First

The first of this course's own four families, and the one whose stated goal is, quite literally, to be unexciting. Debian's release process, Ubuntu's own relationship to it, and the `dpkg` / `apt` tool pairing all serve one consistent underlying value — reliability, chosen deliberately over freshness.

Debian's Three Tiers — a Real Pipeline, Not Just Labels

New and updated packages enter **Unstable** (nicknamed "Sid") first. A package that survives there for a set minimum period with no new critical bugs migrates automatically into **Testing**. When Testing is judged mature enough, a snapshot of it is frozen and eventually released as the next **Stable** — after which Stable receives only security patches and critical fixes, nothing else, for the rest of its multi-year lifetime.

TIER	WHAT IT ACTUALLY IS	WHO IT'S FOR
Unstable (Sid)	Where new packages land first, genuinely can break	Debian developers, the truly adventurous
Testing	Packages that have survived Unstable long enough with no critical bugs	Those wanting newer software with somewhat more confidence
Stable	A frozen snapshot, security-patched only for years	Servers, production systems, anyone valuing predictability above all

"Boring" as an Explicit, Stated Value

Debian's own community has long described its goal, without irony, as being "boring" — a Stable release roughly every two years, package versions frozen at release time regardless of how much newer upstream software becomes during that release's own multi-year support window. This isn't neglect; it's the deliberate trade-off this whole chapter's own title names — stability chosen consciously over having the latest version of everything.

DEBIAN STABLE'S PACKAGE VERSIONS WILL LOOK "OLD," ON PURPOSE

Someone installing Debian Stable expecting the newest release of a language runtime or application will often find something noticeably behind what's available elsewhere — this is the frozen-at-release-time trade-off working exactly

as intended, not a sign of neglect. Real workarounds exist for genuinely needing something newer without abandoning Stable entirely: the official `backports` repository, or app-sandboxing formats like Flatpak, both covered further in Chapter 6.

Ubuntu — Built From Debian, Released on Its Own Terms

Ubuntu takes packages directly from Debian's own Unstable/Testing pool, applies its own patches and branding, then releases on a schedule entirely its own: an **interim release** every six months (9 months of support), and a **Long Term Support (LTS)** release every two years, supported for five years by default — extendable further through Ubuntu's own Extended Security Maintenance.

	CADENCE	SUPPORT WINDOW
Debian Stable	~Every 2 years	Roughly 3 years full support, then Long Term Support extensions via the community
Ubuntu interim	Every 6 months	9 months
Ubuntu LTS	Every 2 years	5 years (extendable via ESM)

dpkg vs. apt — Low-Level Tool, High-Level Wrapper

`dpkg` is the low-level tool that actually installs a single `.deb` file — it has no concept of fetching a package from anywhere, and no automatic dependency resolution; handing it a package whose dependencies aren't already present simply fails. **apt** is the high-level tool built on top of it — it resolves dependencies, fetches packages (and their dependencies) from configured repositories, and calls `dpkg` underneath to actually perform the install.

```
# dpkg: installs a single local .deb file, no dependency resolution
sudo dpkg -i ./somepackage.deb

# apt: resolves dependencies, fetches from repositories, calls dpkg for you
sudo apt install somepackage
```

CONFIRMING WHICH TIER AND CODENAME YOU'RE ACTUALLY ON

`cat /etc/os-release` — the same command introduced in Chapter 1 — reports the exact Debian or Ubuntu codename and version directly, useful before assuming which tier's own behavior (Stable's frozen packages vs. Testing's more frequent updates) actually applies to a given machine.

Hands-On Exercises

EXERCISE 1

A user installs Debian Stable expecting the newest version of a popular application and finds an older one instead. Using this chapter's own warn-box, explain why this isn't a bug, and what two real options exist without abandoning Stable entirely.

View solution

EXERCISE 2

A .deb file downloaded directly from a vendor's website fails to install via dpkg with a dependency error, but installs successfully via apt using the same file. Explain why, using this chapter's own tool comparison.

View solution

EXERCISE 3

Explain why Ubuntu's own interim releases and LTS releases represent a genuinely different trade-off from each other, not just a shorter and longer version of the same thing.

View solution

CHAPTER 2 QUICK REFERENCE

- **Debian's pipeline:** Unstable (Sid) → Testing → Stable, with only security patches once Stable is frozen
- "Boring" is Debian's own stated, deliberate value — stability chosen consciously over freshness
- **Ubuntu:** built from Debian's own packages, released on its own interim (6-month) and LTS (2-year, 5-year support) cadence
- **dpkg** installs a single package with no dependency resolution; **apt** wraps it with dependency resolution and repository fetching
- Debian Stable's "old" package versions are the frozen-at-release trade-off working as intended, not neglect
- **Next chapter:** Fedora & RHEL — A Fast-But-Tested Middle Ground

CHAPTER 3 OF 12

Fedora & RHEL — A Fast-But-Tested Middle Ground

Comparative Linux Distributions

Chapter 3 · Fedora & RHEL — A Fast-But-Tested Middle Ground

Chapter 2 covered Debian feeding a slower, stable identity into a faster downstream (Ubuntu). Fedora and RHEL run the relationship in reverse — and that reversal is the single most important thing to understand about this family before anything else.

The Direction Runs the Opposite Way From Debian/Ubuntu

UPSTREAM IS THE FAST ONE HERE, NOT THE STABLE ONE

In Chapter 2, Debian — the stable, conservative project — is upstream, and Ubuntu adds its own faster cadence on top of Debian's own packages. Fedora and RHEL invert that entirely: **Fedora is the fast-moving upstream**, sponsored by Red Hat as a genuine proving ground for new features, and **RHEL is the slow, enterprise-stable downstream** — taking a mature snapshot of accumulated Fedora features and supporting it for a decade. The same general "fast feeds slow" pattern, running in the opposite direction.

This isn't just organizational trivia — `systemd` itself, now the near-universal default across this entire course (Chapter 1), was first adopted as Fedora's own default init system in 2011, years before most other major distributions followed. Fedora's own explicit identity is built around shipping new upstream technology quickly, specifically so it can be battle-tested before RHEL ever touches it.

Fedora's Own Cadence

A new Fedora release roughly every six months, each version maintained for about 13 months total (giving a brief overlap window between consecutive releases) — genuinely fast by this course's own standards, though still meaningfully more conservative than Arch's continuous model, covered next in Chapter 4.

RHEL & CentOS Stream — the Enterprise Side

RHEL releases major versions roughly every three years, each supported for a full decade through a subscription-based commercial support model — Red Hat's own business, covered further in Chapter 9's governance material. **CentOS Stream** — worth knowing about specifically because its role changed in a widely discussed 2020 shift — used to be a free, downstream rebuild of finished RHEL releases; today it sits

instead as a public "midstream" between Fedora and RHEL, previewing what the next RHEL minor version will actually contain rather than rebuilding what's already shipped.

	ROLE	CADENCE	SUPPORT WINDOW
Fedora	Fast-moving upstream, community-driven	~Every 6 months	~13 months per release
CentOS Stream	Public midstream, previewing RHEL's next minor version	Rolling, tracking RHEL development	Tied to the RHEL major version it feeds
RHEL	Enterprise-stable downstream, commercially supported	~Every 3 years (major)	10 years

dnf & rpm — the Same Pairing Pattern as Chapter 2

`.rpm` (Red Hat Package Manager) is the package format; **dnf** is the modern high-level tool (successor to the older `yum`) — the exact same low-level-format/high-level-manager relationship Chapter 2 established for `.deb` and `apt`, just with different names.

```
# dnf: resolves dependencies, fetches from repositories
sudo dnf install somepackage
```

SELinux — Enabled and Enforcing by Default

SELinux (Security-Enhanced Linux) is a mandatory access control system, enabled and set to **Enforcing** by default on Fedora and RHEL — a genuinely different, more restrictive security model than Debian/Ubuntu's own default use of AppArmor. This is a real, common source of confusion: a service can have entirely correct standard Unix file permissions and still be denied access, because SELinux enforces an additional, separate policy layer on top of ordinary permissions.

"PERMISSION DENIED" ON FEDORA/RHEL IS OFTEN SELINUX, NOT UNIX PERMISSIONS

A service failing with what looks like a permissions error, despite `chmod` / `chown` already being correct, is a classic Fedora/RHEL troubleshooting trap — the real cause is very often an SELinux policy denial, checked via `sestatus` and the audit log, not a file-permission problem at all. Disabling SELinux entirely to make the error go away is a real, meaningful security regression, not a fix — the correct response is adjusting the SELinux policy (or the file's own security context) to permit the specific access actually needed.

CHECKING FOR AN SELINUX DENIAL DIRECTLY

`sestatus` confirms whether SELinux is active and in Enforcing or Permissive mode; `ausearch -m avc -ts recent` searches the audit log directly for recent denials — a faster, more targeted first step than assuming a standard permissions problem when something "should just work."

Hands-On Exercises

EXERCISE 1

Explain why describing Fedora and RHEL's relationship as "just like Debian and Ubuntu, but faster" would actually get the underlying structure backwards, using this chapter's own finding-box.

View solution

EXERCISE 2

A service on a RHEL machine fails with what looks like a permissions error, even though file ownership and mode bits are already correct. Using this chapter's own warn-box, explain the likely real cause and why disabling SELinux isn't the right fix.

View solution

EXERCISE 3

Explain what CentOS Stream's real role is today, and why calling it "just CentOS, like before 2020" would be inaccurate.

View solution

CHAPTER 3 QUICK REFERENCE

- **Fedora is the fast upstream; RHEL is the stable downstream** — the reverse direction from Debian (stable upstream) feeding Ubuntu (faster downstream)
- systemd itself debuted as Fedora's own default years before most other distros adopted it
- **CentOS Stream** — since 2020, a public midstream previewing RHEL's next minor version, not a rebuild of finished RHEL
- **rpm/dnf** mirrors Chapter 2's own dpkg/apt low-level/high-level pairing
- **SELinux** is enabled and Enforcing by default — a real, separate policy layer beyond ordinary Unix permissions, and a common troubleshooting trap
- **Next chapter:** Arch — Rolling Release & DIY

CHAPTER 4 OF 12

Arch — Rolling Release & DIY

Comparative Linux Distributions

Chapter 4 · Arch — Rolling Release & DIY

Chapters 2 and 3 both covered distributions built around discrete versions — Debian Stable's own frozen releases, Fedora's ~13-month cycles, RHEL's multi-year majors. Arch has none of that at all — no version number to even ask about — and its own installation philosophy is just as deliberately different.

Rolling Release — There Is No "Arch 15"

Arch has no discrete releases in the way every distribution covered so far does. Packages update continuously, directly from upstream, with no periodic freeze-and-snapshot cycle at all — the system genuinely *is* whatever's currently in the repositories, all the time. There's no equivalent to "Fedora 40" or "Debian 12" to name, because the entire concept of a numbered release doesn't apply here.

	WHAT "AN UPDATE" MEANS
Debian / Fedora / RHEL (point release)	Small patches within a fixed version, until the next full version replaces it entirely
Arch (rolling)	Continuous, small, frequent updates — there is no "next version" to eventually move to

pacman & the AUR

pacman is Arch's own package manager — `pacman -S` installs, `pacman -R` removes, `pacman -Syu` performs a full system update. The philosophy is frequent, small, incremental updates rather than the big periodic jumps Chapters 2 and 3 both described.

The **AUR** (Arch User Repository) is community-submitted — but critically, it doesn't hold pre-built binary packages the way the official repositories do. It holds **PKGBUILD scripts**, community-written build instructions that fetch source code and compile it locally.

THE AUR IS NOT VETTED THE WAY OFFICIAL REPOSITORIES ARE

An AUR package is a user-submitted script, not a package reviewed and signed by Arch's own developers — treating it with the same default trust as an official repository package is a real mistake. Reading a PKGBUILD before building it

(to see exactly what it downloads and runs) is standard, sensible practice within the Arch community itself, not excessive caution.

The Arch Wiki — Respected Well Beyond Arch Itself

The Arch Wiki is genuinely well regarded across the entire Linux world — detailed enough that users of completely different distributions regularly consult it for generic Linux configuration and troubleshooting information, adapting package-manager-specific steps to their own system as needed.

"You Configure Everything Yourself"

Installing and Configuring Linux covers a guided installer making sensible decisions on a user's behalf — partitioning, package selection, bootloader setup, all handled with reasonable defaults. Arch's own traditional installation process is the deliberate opposite: partitioning, mounting, base package selection, and bootloader configuration are all performed explicitly, one command at a time, with no default assumed unless the installer using it chose one on purpose. (A newer, official `archinstall` script offers a guided path today, but the manual process remains the canonical, most-documented route — and still the one the Arch Wiki's own installation guide centers on.)

CHECKING INSTALLED PACKAGE VERSIONS DIRECTLY

`pacman -Q` lists every installed package and its exact current version — genuinely useful given there's no distribution-wide version number to reference instead; on a rolling-release system, the individual package versions are the only meaningful version information that exists at all.

ROLLING RELEASE MEANS READING BEFORE UPDATING, NOT BLINDLY RUNNING PACMAN -SYU

Arch's own official policy explicitly expects users to check the Arch news feed before a major system update — occasionally, a package update requires manual intervention (a config file migration, a one-time extra step) that an automatic update alone won't handle. This isn't a flaw in the rolling model; it's the trade-off the model makes explicitly, in exchange for never needing a disruptive, all-at-once version jump.

Hands-On Exercises

EXERCISE 1

A user asks "which version of Arch are you running?" Explain why this question doesn't have the kind of answer it would have for Debian or Fedora, using this chapter's own explanation of the rolling-release model.

View solution

EXERCISE 2

A user installs an AUR package the same way they'd install one from Arch's official repositories, without reviewing it first. Using this chapter's own warn-box, explain what real risk this overlooks.

View solution

EXERCISE 3

Explain what "you configure everything yourself" actually means in Arch's traditional installation process, contrasting it directly with Installing and Configuring Linux's own guided installer.

View solution

CHAPTER 4 QUICK REFERENCE

- **Rolling release** — no version numbers at all; the system is always whatever's currently in the repos
- **pacman** — `-S` install, `-R` remove, `-Syu` full system update, favoring frequent small updates
- **The AUR** holds unvetted, user-submitted PKGBUILD scripts — not equivalent in trust to official repository packages
- The Arch Wiki is respected and consulted well beyond the Arch community itself
- Arch's traditional installer requires every decision explicitly, contrasted with Installing and Configuring Linux's own guided defaults
- Rolling release trades away disruptive version jumps for the occasional need to read release notes before updating
- **Next chapter:** Alpine — Minimalism as a Design Goal

CHAPTER 5 OF 12

Alpine — Minimalism as a Design Goal

Comparative Linux Distributions

Chapter 5 · Alpine — Minimalism as a Design Goal

Chapter 1's own compare-table already flagged Alpine as the one family in this course not using systemd. This chapter explains why — Alpine isn't Debian/Fedora/Arch with some things removed; nearly every major component is deliberately swapped for a smaller alternative, minimalism chosen as a first-class goal rather than an afterthought.

musl vs. glibc — A Different C Standard Library Entirely

Every other family in this course — Debian, Ubuntu, Fedora, RHEL, Arch — uses **glibc** (the GNU C Library), the long-standing, feature-complete standard most software is built and tested against. Alpine uses **musl** instead — a much smaller, simpler, more strictly standards-compliant C library.

> **"IT WORKS EVERYWHERE EXCEPT THIS ALPINE CONTAINER" IS OFTEN A MUSL ISSUE**
>
> Software compiled or tested only against glibc can behave subtly differently, or fail outright, under musl — a genuinely common, real cause of Docker containers that run perfectly on a glibc-based image and misbehave on an otherwise-identical Alpine one. This isn't Alpine being "broken" — it's a real, different library implementation, and the fix is either adjusting the software to be musl-compatible or choosing a glibc-based minimal image when full compatibility genuinely can't be sacrificed.

BusyBox — Dozens of Utilities, One Small Binary

Instead of the full GNU coreutils most distros ship, Alpine uses **BusyBox** — a single, compact executable providing simplified implementations of dozens of standard Unix utilities (`ls`, `grep`, `sed`, `mount`, and many more) at once. The trade-off is real: a dramatic size reduction, in exchange for a reduced feature set and occasionally different flag behavior than the full GNU version of the same command — a script written assuming GNU-specific `sed` or `grep` flags can behave differently under BusyBox's own trimmed implementation.

OpenRC — Delivering on Chapter 1's Own Named Exception

Alpine uses **OpenRC** rather than systemd, consistent with its broader minimalism-first philosophy — systemd bundles a much larger set of daemons and components alongside pure init functionality, exactly the kind of added surface area Alpine's own design goal explicitly avoids. Chapter 8 covers init systems across all four families in full; this is the "real, living exception" that chapter (and Chapter 1) already named.

apk — Alpine's Own Package Manager

`apk add`, `apk del`, and `apk update` round out Alpine's own package tooling — small, fast, and consistent with the same minimalism running through every other component covered so far.

Why Alpine Dominates Docker Base Images

A full Ubuntu base image commonly runs 70+ MB; Alpine's own base image is typically in the 5–8 MB range — a direct, concrete consequence of musl, BusyBox, and OpenRC all replacing much larger equivalents at once. Smaller images pull faster, deploy faster, and offer a genuinely smaller attack surface — exactly why so many official images on Docker Hub offer a dedicated "-alpine" variant, and why the Docker courses' own image-size and layer-optimization material is really the same underlying theme, just expressed at the individual-image level rather than the whole-OS level this chapter covers.

COMPONENT	MOST DISTROS IN THIS COURSE	ALPINE
C library	glibc	musl
Core utilities	Full GNU coreutils	BusyBox (one compact binary)
Init system	systemd	OpenRC
Typical base image size	70+ MB (e.g. Ubuntu)	~5–8 MB

CHECKING WHICH LIBC A SYSTEM IS ACTUALLY USING

`ldd --version` reports musl or glibc directly — a fast first diagnostic when a "works everywhere except this container" report comes in, before assuming the problem lies anywhere else.

Hands-On Exercises

EXERCISE 1

A CI pipeline's shell script works correctly on an Ubuntu-based runner but produces different output on an Alpine-based one, using the same sed command. Using this chapter's own BusyBox material, explain the likely cause.

View solution

EXERCISE 2

A compiled binary that runs fine on Debian and Fedora crashes immediately when run inside an Alpine-based Docker container. Using this chapter's own warn-box, explain the likely cause and the two real options for resolving it.

View solution

EXERCISE 3

Explain why so many official Docker Hub images offer a dedicated "-alpine" variant, using this chapter's own explanation of what actually makes Alpine's base image so much smaller.

View solution

CHAPTER 5 QUICK REFERENCE

- **musl** (Alpine) vs. **glibc** (everyone else in this course) — a genuinely different C standard library, a real compatibility gotcha
- **BusyBox** — dozens of utilities in one compact binary, with a reduced/different feature set than full GNU coreutils
- **OpenRC** — Alpine's own init system, delivering on Chapter 1's named systemd exception
- **apk** — small, fast, consistent with the whole minimalism theme
- Alpine's ~5–8 MB base image (vs. 70+ MB for a full distro) is why it's the default minimal Docker base
- `ldd --version` quickly confirms musl vs. glibc when troubleshooting a container-specific failure
- **Next chapter:** Package Managers, Compared Directly

CHAPTER 6 OF 12

Package Managers, Compared Directly

Comparative Linux Distributions

Chapter 6 · Package Managers, Compared Directly

Chapters 2 through 5 each introduced one package manager on its own terms. This chapter puts all four side by side, doing the exact same real tasks — and covers a real dependency-resolution policy difference worth knowing before it causes a genuinely broken system.

The Same Tasks, Four Different Commands

TASK	APT (DEBIAN/UBUNTU)	DNF (FEDORA/RHEL)	PACMAN (ARCH)	APK (ALPINE)
Install	`apt install pkg`	`dnf install pkg`	`pacman -S pkg`	`apk add pkg`
Remove	`apt remove pkg`	`dnf remove pkg`	`pacman -R pkg`	`apk del pkg`
Refresh index	`apt update`	`dnf check-update`	`pacman -Sy`	`apk update`
Upgrade all	`apt upgrade`	`dnf upgrade`	`pacman -Syu`	`apk upgrade`
Search	`apt search term`	`dnf search term`	`pacman -Ss term`	`apk search term`
List installed	`apt list --installed`	`dnf list installed`	`pacman -Q`	`apk info`

Dependency Resolution — Not All Equally Permissive

apt and dnf both resolve dependencies against repository metadata and are relatively tolerant of partial upgrades — installing one new package without a full system update generally works fine on either. pacman is meaningfully stricter, for a reason directly tied to Chapter 4's own rolling-release material.

ARCH'S OWN "PARTIAL UPGRADES ARE UNSUPPORTED" POLICY

On a rolling-release system, installed packages and their shared libraries are tightly version-coupled — installing a single new package (`pacman -S somepackage`) without first fully syncing and upgrading the entire system (`pacman -Syu`) can pull in a package built against library versions newer than what's currently installed, producing a genuinely broken partial state. The correct habit on Arch specifically is always running a full `-Syu` before installing anything new, never syncing the index just to grab one specific package in isolation — a real, well-documented policy, not overcaution.

Ansible's Package Module — Abstracting Over All Four

Ansible's `ansible.builtin.package` module detects whichever package manager is actually present on a target host and calls it automatically — the exact differences this whole chapter just catalogued, abstracted away behind one generic interface:

```
# Works identically whether the target is Debian, Fedora, Arch, or Alpine
- name: Install a package regardless of distro
  ansible.builtin.package:
    name: git
    state: present
```

This generic module only supports the lowest common denominator across all four — install/remove/ensure-latest by name. Anything more specific to one package manager (an Arch AUR helper, a Fedora module stream) still requires that manager's own dedicated Ansible module rather than the generic one.

CHECKING PACKAGE DETAILS BEFORE INSTALLING

`apt-cache policy pkg`, `dnf info pkg`, `pacman -Si pkg`, and `apk info pkg` each show detailed information about a package — version, size, dependencies — before committing to an install, the equivalent lookup across all four tools.

Hands-On Exercises

EXERCISE 1

A user runs `pacman -S somepackage` directly, without first running a full system upgrade, and ends up with a broken system afterward. Using this chapter's own warn-box, explain what happened and the correct habit going forward.

View solution

EXERCISE 2

Explain why an Ansible playbook using the generic package module can install git identically across a Debian, a Fedora, and an Alpine machine without three separate tasks, and what real limitation this convenience comes with.

View solution

EXERCISE 3

Using this chapter's own compare-table, explain why apt and dnf are described as "relatively tolerant of partial upgrades" while pacman genuinely isn't, tying your answer to Chapter 4's own rolling-release material.

View solution

CHAPTER 6 QUICK REFERENCE

- The same tasks across four tools: `apt` / `dnf` install-remove-upgrade-search-list, `pacman` / `apk` equivalents
- apt/dnf tolerate partial upgrades reasonably well; pacman genuinely does not — always full-sync before installing anything new on Arch
- Ansible's `package` module abstracts over all four for the lowest-common-denominator tasks; manager-specific modules remain necessary for anything more specialized
- **Next chapter:** Release Models — Point Release vs. Rolling vs. Enterprise

CHAPTER 7 OF 12

Release Models — Point Release vs. Rolling vs. Enterprise

Comparative Linux Distributions

Chapter 7 · Release Models — Point Release vs. Rolling vs. Enterprise

Debian's own three tiers, Ubuntu's interim/LTS split, Fedora's ~13-month cadence, RHEL's decade-long support window, and Arch's rolling model have all appeared already, one chapter at a time. This chapter puts them on a single, named spectrum — and corrects a real oversimplification worth catching before it causes a bad decision.

One Spectrum, Not Three Unrelated Categories

THE STABILITY-VS-FRESHNESS AXIS, LAID OUT END TO END

Enterprise (RHEL) → **Point release (Debian Stable)** → **Fast cycle (Fedora, Ubuntu interim)** → **Rolling (Arch)**. Moving left prioritizes predictability and a long, tested support window; moving right prioritizes access to current software, at the cost of more frequent change to manage.

Every family this course has covered sits somewhere on this one axis — not in three or four unrelated buckets, but at genuinely different points along the same underlying trade-off.

Ubuntu's Own Hybrid Position

Ubuntu is worth calling out specifically because it doesn't occupy just one point on the spectrum — it deliberately offers two, within a single project. LTS releases sit further toward the stability end (five years of support, minimal disruption); interim releases sit further toward the freshness end (six-month cycles, newer software, a much shorter nine-month window). Choosing Ubuntu doesn't commit to one position on the axis at all — it commits to choosing between two of Ubuntu's own offerings, each occupying a different point.

This hedge isn't unique to Ubuntu — openSUSE, outside this course's own four covered families, runs the same pattern explicitly with two separately named products: Tumbleweed (rolling) and Leap (point release), worth knowing exists even though it isn't covered here in depth.

Matching a Model to a Real Use Case

MODEL	STABILITY	FRESHNESS	TYPICALLY BEST FOR
Enterprise (RHEL)	Highest — a decade of support	Lowest — years behind upstream by design	Production systems needing long-term predictability above all
Point release (Debian Stable, Ubuntu LTS)	High	Low-to-moderate	Servers, most production desktops
Fast cycle (Fedora, Ubuntu interim)	Moderate	High	Developers wanting current software with still-discrete, testable releases
Rolling (Arch)	Depends on maintenance discipline	Highest — always current	Engaged users and developers genuinely willing to follow Chapter 4's own update discipline

Chapter 11 turns this same table into a concrete decision framework, applied to several real scenarios.

"ROLLING MEANS LESS STABLE" IS AN OVERSIMPLIFICATION WORTH CORRECTING

Arch's own rolling model isn't inherently less reliable than a point release — for someone genuinely following Chapter 4's own maintenance discipline (reading release notes, always running a full sync before installing anything new), it can be perfectly dependable day to day. The real trade-off isn't "stable vs. unstable" — it's "install-and-mostly-forget-for-years" (point release/enterprise) versus "requires ongoing attention to stay healthy" (rolling). Framing rolling release as simply "riskier" misses that the actual cost is maintenance discipline, not inherent unreliability.

CHECKING A DISTRO'S OWN OFFICIAL SUPPORT TIMELINE BEFORE DEPLOYING IT

Debian, Ubuntu, Fedora, and RHEL all publish official end-of-life dates for every release — worth checking directly before choosing a version for a real deployment, rather than assuming "the current version" will remain supported for as long as it happens to be needed.

Hands-On Exercises

EXERCISE 1

A colleague says "I'm choosing Ubuntu, so I know exactly where it sits on the stability-vs-freshness spectrum." Using this chapter's own material, explain why this statement is incomplete.

View solution

EXERCISE 2

A colleague argues Arch should never be used on any system that needs to be reliable, because it's a rolling release. Using this chapter's own warn-box, explain what this argument gets wrong.

View solution

EXERCISE 3

Using this chapter's own spectrum, explain why RHEL and Arch represent the two opposite ends of the same axis, rather than being two entirely unrelated approaches to releasing software.

View solution

CHAPTER 7 QUICK REFERENCE

- **One spectrum:** Enterprise → Point release → Fast cycle → Rolling, trading freshness for predictability as you move left
- Ubuntu occupies two points on this spectrum at once (LTS and interim), not a single fixed position
- openSUSE runs the same "offer both ends" pattern explicitly via Tumbleweed and Leap
- "Rolling means less stable" is a real oversimplification — the true cost is ongoing maintenance discipline, not inherent unreliability
- Always check a distro's own official EOL date before choosing a version for production
- **Next chapter:** Init Systems & Defaults Across Distros

CHAPTER 8 OF 12

Init Systems & Defaults Across Distros

Comparative Linux Distributions

Chapter 8 · Init Systems & Defaults Across Distros

Chapter 1's own table, Chapter 3's Fedora-first systemd history, and Chapter 5's OpenRC material have all pointed toward this chapter without covering it fully. This is where the init-system thread finally comes together — plus a related "defaults" story worth telling alongside it: why Fedora quietly made one of the more interesting filesystem decisions in recent Linux history.

systemd — Revisited, Not Retaught

systemd in Depth already covers PID 1, unit files, parallelized boot, and journald in full detail — this chapter doesn't repeat that. What matters here is *who uses it*: Debian and Ubuntu adopted systemd starting around Debian 8 / Ubuntu 15.04, Fedora adopted it earliest (2011, per Chapter 3), and Arch made it the default around 2012–2013. Three of this course's own four families converged on the same init system — a real, genuine point of agreement this course keeps returning to.

Ubuntu's Own Forgotten Alternative — Upstart

Before adopting systemd, Ubuntu ran its own genuinely different init system for nearly a decade: **Upstart**, an event-based init system Ubuntu itself developed and used from roughly 2006 to 2015. Ubuntu didn't simply wait for systemd to appear — it built and maintained an entirely separate alternative first, before eventually converging with the rest of this course's systemd-using families anyway.

OpenRC — The Real, Living Exception, Recapped

Chapter 5 already covered Alpine's own use of OpenRC as part of its broader minimalism theme. The core difference worth restating here: OpenRC uses simpler, dependency-based service scripts without systemd's own much larger bundle of additional components (no built-in equivalent to `journald` — Alpine uses traditional syslog instead — nor to `logind`, `timedated`, or `resolved`).

"EVERYONE USES SYSTEMD NOW" IS A REAL, COMMON ASSUMPTION THAT BREAKS ON ALPINE

Running `systemctl status` against an Alpine machine doesn't fail with a helpful redirect to the right tool — `systemctl` simply isn't present at all, since Alpine runs OpenRC. The equivalent commands are `rc-status` and `rc-`

`service`. This is the exact same category of mistake Chapter 1 already warned about for package managers — assuming a near-universal default is a true universal, and getting caught out on the one real exception.

Default Filesystems — a Related "Defaults" Story

Historically, ext4 has been the safe, boring default across most of this course's own families — and it still is for Debian, Ubuntu, and Alpine (consistent with Alpine's own simplicity-first theme from Chapter 5). Fedora broke from that pattern in a genuinely notable way.

WHY FEDORA DEFAULTS TO BTRFS — A DECISION THAT ACTUALLY FITS EVERYTHING ELSE THIS COURSE HAS COVERED

Starting with Fedora 33 (2020), Fedora's desktop variants default to **Btrfs** instead of ext4 — covered in full technical depth in *Linux Filesystems 3*. The reasoning ties directly back to Chapter 3's and Chapter 7's own material: Fedora's fast-cycle release model means more frequent updates, and Btrfs's own cheap, built-in snapshot capability makes rolling back a failed update dramatically easier than it would be on ext4. A distribution that changes more often genuinely benefits more from cheap, built-in rollback than one that rarely changes — Fedora's own default filesystem choice is a direct, sensible consequence of its own release-model identity, not an arbitrary preference.

Arch's own installer, consistent with Chapter 4's "you configure everything yourself" philosophy, doesn't default to any filesystem at all — the installer using it chooses explicitly, same as every other installation decision that chapter already described.

	INIT SYSTEM	DEFAULT FILESYSTEM
Debian / Ubuntu	systemd	ext4
Fedora / RHEL	systemd	Btrfs (Fedora desktop, since F33); ext4/XFS common on RHEL
Arch	systemd	None assumed — chosen explicitly during install
Alpine	OpenRC	ext4

CONFIRMING WHICH INIT SYSTEM A MACHINE IS ACTUALLY RUNNING

`ps -p 1 -o comm=` reports exactly what's running as PID 1 — `systemd` or `openrc-init` — a fast, direct check rather than assuming based on which distribution family a machine belongs to.

Hands-On Exercises

EXERCISE 1

Running `systemctl status` against a remote machine returns "command not found." Using this chapter's own warn-box, explain the most likely cause and the correct alternative commands.

View solution

EXERCISE 2

Explain why Fedora's own decision to default to Btrfs makes sense specifically given what Chapters 3 and 7 already established about Fedora's release model, rather than being an unrelated technical preference.

View solution

EXERCISE 3

Explain why Ubuntu's own history with Upstart is a genuinely interesting fact, rather than a minor footnote, given that Ubuntu ultimately ended up using systemd anyway like most of this course's other families.

View solution

CHAPTER 8 QUICK REFERENCE

- **systemd** — Debian/Ubuntu, Fedora/RHEL, and Arch all converged on it; *systemd in Depth* covers the mechanics in full
- Ubuntu ran its own alternative, **Upstart**, for nearly a decade before eventually adopting systemd too
- **OpenRC** (Alpine) is the real, living exception — `rc-status` / `rc-service`, not `systemctl`
- **Fedora defaults to Btrfs** (since F33) specifically because its fast-cycle release model benefits from cheap, built-in rollback — a decision that follows directly from its own release-model identity
- Arch's installer assumes no default filesystem at all, consistent with its own "configure everything yourself" philosophy
- `ps -p 1 -o comm=` confirms which init system is actually running
- **Next chapter:** Community, Governance & Support Models

CHAPTER 9 OF 12

Community, Governance & Support Models

Comparative Linux Distributions

Chapter 9 · Community, Governance & Support Models

Every prior chapter compared technical decisions — package managers, init systems, release cadences, default filesystems. This chapter steps back to a genuinely different axis: *who actually decides* what goes into each distribution, and how each project is funded and sustained.

Debian — Volunteer-Democratic

The Debian Project is a genuine volunteer nonprofit — an annually elected **Debian Project Leader (DPL)**, with major decisions formalized through the Debian Constitution and settled by **General Resolutions**, a real, structured voting mechanism open to Debian's own registered developers. It's one of the largest and longest-running examples of genuinely democratic open-source governance anywhere.

Ubuntu — Commercially Directed

Canonical Ltd., a private company founded by Mark Shuttleworth, owns and directs Ubuntu's development. Community contributions are genuinely welcomed, but final decision-making authority rests with Canonical, not a democratic vote of contributors the way Debian's own governance works.

> **UBUNTU INHERITS DEBIAN'S PACKAGES, NOT DEBIAN'S GOVERNANCE**
>
> Chapter 2 already established that Ubuntu builds from Debian's own package pool — but that relationship is purely technical. Debian's volunteer-democratic governance model has no equivalent inside Ubuntu itself; Canonical's own commercial decision-making sits entirely separately from the packages it's built on top of.

Red Hat — a Hybrid Model Behind Fedora and RHEL

Red Hat (acquired by IBM in 2019) runs a genuine commercial open-source business: **RHEL** itself is the paid product — subscriptions, certification, SLA-backed support. **Fedora**, by contrast, is governed by the **Fedora Council**, a genuine mix of Red Hat employees and independently elected community members — neither purely volunteer-democratic like Debian nor purely corporate-directed like Ubuntu, but something in between.

Arch — Minimal Governance

No foundation, no formally elected leader in Debian's own sense — a small, informal team of trusted package maintainers and developers, making decisions with deliberately low bureaucracy. This extends Chapter 4's own DIY philosophy one level further: not just "you configure your own system," but "the project itself runs with minimal formal structure too."

Alpine — Small and Volunteer-Driven

Alpine is maintained by a small nonprofit development team — closer in spirit to Debian's own volunteer model, just at a considerably smaller scale, consistent with Chapter 5's own minimalism theme extended here to organizational simplicity as well.

	GOVERNANCE	FUNDING
Debian	Elected DPL, democratic General Resolutions	Volunteer/donation-based nonprofit
Ubuntu	Canonical Ltd. — final authority, community contributions welcomed	Commercial (Canonical)
Fedora	Fedora Council — Red Hat employees + elected community members	Red Hat-sponsored
RHEL	Red Hat (IBM) — fully commercial	Subscription/support revenue
Arch	Small trusted maintainer team, low bureaucracy	Volunteer/donation-based
Alpine	Small nonprofit development team	Volunteer/donation-based

RED HAT'S 2023 SOURCE-ACCESS CHANGE — COMPLETING CHAPTER 3'S OWN CENTOS STREAM STORY

Chapter 3 covered CentOS Stream's own 2020 shift from a downstream RHEL rebuild to a public midstream. In 2023, Red Hat restricted public access to RHEL's own source code further, moving it behind a customer portal rather than fully open git repositories as before — a real, controversial governance decision. This directly caused the rise of **Rocky Linux** and **AlmaLinux** as the new free, RHEL-compatible rebuilds — the role CentOS itself used to play before 2020. Anyone today looking for "a free, RHEL-compatible OS" needs to know CentOS Stream isn't that anymore; Rocky Linux and AlmaLinux are the actual modern answer.

WHY GOVERNANCE MODEL MATTERS FOR A REAL DEPLOYMENT DECISION

A commercially backed distro (Ubuntu, RHEL) offers predictable, funded support timelines with a company standing behind them; a volunteer project (Debian, Arch, Alpine) depends on continued community engagement — a real, if less easily quantified, sustainability factor worth weighing directly alongside the technical release-model comparison from Chapter 7.

Hands-On Exercises

EXERCISE 1

A colleague assumes that because Ubuntu is built from Debian's own packages, Ubuntu must also follow Debian's own democratic governance process. Using this chapter's own finding-box, explain why this assumption is wrong.

View solution

EXERCISE 2

Someone wants a free, RHEL-compatible operating system today and plans to use CentOS Stream for that purpose. Using this chapter's own warn-box and Chapter 3's own material, explain why that plan needs updating.

View solution

EXERCISE 3

Explain why the Fedora Council is described as a genuine hybrid governance model, rather than fitting cleanly into either the Debian-style or Ubuntu-style category.

View solution

CHAPTER 9 QUICK REFERENCE

- **Debian** — elected DPL, democratic General Resolutions, volunteer nonprofit
- **Ubuntu** — Canonical Ltd. holds final authority; inherits Debian's packages, not its governance
- **Fedora** — the Fedora Council, a genuine Red Hat/community hybrid; **RHEL** — fully commercial, subscription-funded
- **Arch** — a small trusted maintainer team, minimal bureaucracy, extending Chapter 4's DIY philosophy to the project level
- **Alpine** — a small volunteer nonprofit, consistent with its own minimalism theme
- 2023's RHEL source-access restriction produced **Rocky Linux** and **AlmaLinux** as CentOS's real modern successors
- **Next chapter:** Raspberry Pi OS — A Debian Derivative for Constrained Hardware

CHAPTER 10 OF 12

Raspberry Pi OS — A Debian Derivative for Constrained Hardware

Comparative Linux Distributions

Chapter 10 · Raspberry Pi OS — A Debian Derivative for Constrained Hardware

A genuine derivative rather than a fifth independent family — but worth its own chapter precisely because it's directly relevant to *Raspberry Pi Projects* and *More Raspberry Pi Projects*, both already on this site and both quietly assuming everything this chapter actually explains.

Built Directly on Debian — Not Just "Similar To"

Raspberry Pi OS (historically named Raspbian) is literally derived from Debian's own package base — not an independent distribution inspired by Debian's style. This means Chapter 2's own `apt` / `dpkg` material transfers directly, with nothing new to learn: `apt install` works exactly the same way here as on any other Debian-family system.

What's Actually Different — ARM & Hardware Integration

Raspberry Pi hardware runs on **ARM** processors, not the x86/x86-64 architecture most desktop and server distributions primarily target. Raspberry Pi OS packages are specifically compiled for ARM (`armhf` for older 32-bit models, `arm64` for newer 64-bit-capable ones), and the distribution bundles a custom kernel with Pi-specific hardware support — the camera module, GPIO pin access, and hardware-accelerated video decoding all depend on kernel and driver integration a generic ARM build simply doesn't include.

A Genuinely Different Boot Process

Every other family in this course cares about BIOS/UEFI and a bootloader like GRUB. The Raspberry Pi's own boot process is entirely different hardware territory: the Pi's GPU itself initializes first and reads a plain text configuration file, `/boot/config.txt`, before the Linux kernel is even loaded — no BIOS/UEFI screen, no GRUB menu, no equivalent to anything covered in *GRUB & Multibooting* at all.

CONFIG.TXT AS THE PI'S OWN ROUGH EQUIVALENT TO A BIOS/UEFI SCREEN

For anyone already thinking in terms of *Build a New Development PC 11*'s own BIOS/UEFI settings, `/boot/config.txt` serves a genuinely similar purpose — hardware-level configuration read before the OS proper starts — just expressed as an edited text file rather than an interactive firmware menu.

"JUST INSTALL STANDARD DEBIAN ARM" ISN'T THE SAME THING

Running a generic Debian ARM image directly on Raspberry Pi hardware, without Raspberry Pi OS's own kernel and firmware integration, commonly either fails to boot at all or boots with broken hardware support — no camera, no GPIO, no hardware-accelerated video. This is a real, common mistake for newcomers researching "how do I put Linux on my Pi" — the Pi-specific image isn't a convenience layered on top of standard Debian; it's genuinely necessary for the hardware to work correctly at all.

Why This Chapter Matters for Raspberry Pi Projects & More Raspberry Pi Projects

Every project in those two courses assumes exactly the hardware integration this chapter describes. *Raspberry Pi Projects 6*'s own GPIO and physical computing material, in particular, depends directly on the kernel-level GPIO support Raspberry Pi OS provides — none of it would function on a generic, non-Pi-specific Linux image at all.

	ARCHITECTURE	BOOT PROCESS	HARDWARE SUPPORT
Standard Debian	Primarily x86/x86-64	BIOS/UEFI + bootloader (e.g. GRUB)	Generic PC hardware
Raspberry Pi OS	ARM (armhf/arm64)	GPU-first firmware reading config.txt	Pi-specific: camera, GPIO, hardware video decode

PREPARING AN SD CARD THE RECOMMENDED WAY

The official Raspberry Pi Imager tool writes a correctly configured Raspberry Pi OS image directly to an SD card or USB drive — the standard, recommended starting point, rather than manually writing a generic ARM image and hoping the hardware-specific pieces sort themselves out.

Hands-On Exercises

EXERCISE 1

A user writes a generic Debian ARM image to an SD card and finds their Raspberry Pi's camera module doesn't work at all. Using this chapter's own warn-box, explain the likely cause.

View solution

EXERCISE 2

Explain why someone already comfortable with apt and dpkg from Chapter 2 doesn't need to learn a new package manager to use Raspberry Pi OS.

View solution

EXERCISE 3

Explain why Raspberry Pi Projects 6's own GPIO material wouldn't work on a generic, non-Pi-specific Linux distribution, using this chapter's own hardware-integration material.

View solution

CHAPTER 10 QUICK REFERENCE

- Raspberry Pi OS is **directly derived from Debian** — the same apt/dpkg tooling from Chapter 2 applies unchanged
- ARM-compiled (armhf/arm64), with a custom kernel providing camera, GPIO, and hardware-video-decode support
- The Pi's own boot process is entirely different from BIOS/UEFI/GRUB — GPU-first firmware reading `/boot/config.txt`
- A generic Debian ARM image is not a substitute — it commonly fails to boot or loses hardware support entirely
- Raspberry Pi Projects 6's own GPIO material depends directly on this chapter's own hardware integration
- **Next chapter:** Choosing the Right Distribution for a Real Task

CHAPTER 11 OF 12

Choosing the Right Distribution for a Real Task

Comparative Linux Distributions

Chapter 11 · Choosing the Right Distribution for a Real Task

Chapter 7 promised this chapter would turn its own spectrum table into a real decision framework. Every chapter since has added another axis worth weighing — SELinux, footprint, governance, hardware needs. This chapter brings all of it together and applies it to five genuinely different real scenarios.

The Framework — Six Questions

1. **What's the actual use case?** Server/production, personal desktop, a container base image, or dedicated embedded/Pi hardware.
2. **Where does this task sit on Chapter 7's own stability-vs-freshness spectrum?** A production system tolerates far less disruption than a personal experimentation machine.
3. **Who's maintaining it, and how much ongoing attention can they realistically give it?** Chapter 7's own warn-box named this directly — rolling release demands real, sustained maintenance discipline.
4. **Does it need a specific security model?** Chapter 3's SELinux-by-default is a real, distinguishing factor for certain compliance or hardening requirements.
5. **Does minimal footprint or fast startup genuinely matter?** Chapter 5's musl/BusyBox trade-offs are worth it specifically when size and attack surface matter more than broad compatibility.
6. **Is dedicated hardware integration required?** Chapter 10's own Raspberry Pi material applies directly whenever the answer is yes.

Applying the Framework to Five Real Scenarios

SCENARIO	RECOMMENDED	WHY
Production web server, small business, limited budget	Debian Stable or Ubuntu LTS	Point-release stability, wide documentation, no subscription cost (Chapters 2, 7, 9)
Developer's personal desktop, wants current tools	Fedora (or Arch, if genuinely willing to follow its maintenance discipline)	Fast cycle balances freshness with real testing (Chapter 3); Arch only if Chapter 7's own discipline is genuinely accepted
Docker base image for a microservice	Alpine — unless a real glibc dependency exists	Smallest footprint, fastest pulls, smaller attack surface (Chapter 5); glibc-based alternative if Chapter 5's own

SCENARIO	RECOMMENDED	WHY
		compatibility warning applies
Raspberry Pi home automation project	Raspberry Pi OS specifically	The only option with the necessary hardware integration (Chapter 10) — not a generic distro chosen for other reasons
Enterprise server needing vendor SLA/compliance certification	RHEL	Commercial support, certification, and a decade-long support window are the actual requirement (Chapters 3, 9)

"I ALREADY KNOW ARCH" ISN'T A REASON TO DEPLOY IT ON A PRODUCTION SERVER

Choosing a distribution based purely on personal familiarity, without weighing the task's own actual requirements, is a real and common mistake — someone genuinely comfortable running Arch on their own desktop can still be choosing badly by reflexively deploying it to a production server that actually needs RHEL's own support SLA, or Debian Stable's own multi-year predictability. The right distribution is a function of the task's own requirements, not simply whichever one is most familiar.

WHEN GENUINELY UNSURE, DOCUMENTATION VOLUME IS ITSELF A REAL FACTOR

Ubuntu LTS for servers and desktops, Debian Stable for maximum stability, and Fedora for a fast-but-tested desktop are all reasonable defaults specifically because of how much community documentation and troubleshooting material already exists for each — a real, practical consideration worth weighing alongside the six more technical questions above, not a lesser one.

Hands-On Exercises

EXERCISE 1

A colleague wants to deploy a Docker microservice using an Alpine base image, but the application depends on a library only tested against glibc. Using this chapter's own framework and Chapter 5's own warn-box, explain what should happen instead.

View solution

EXERCISE 2

Using this chapter's own six questions, explain why the Raspberry Pi home automation scenario has effectively only one reasonable answer, unlike the other four scenarios in this chapter's own table.

View solution

EXERCISE 3

A system administrator who personally prefers Arch on their own desktop is asked to choose an OS for a new production server needing a vendor support contract. Using this chapter's own warn-box, explain why their personal preference shouldn't drive this specific decision.

View solution

CHAPTER 11 QUICK REFERENCE

- **Six questions:** use case, stability-vs-freshness, maintenance capacity, security model, footprint, hardware needs
- Production/business systems generally favor point-release or enterprise; personal/dev machines can reasonably favor freshness
- Docker base images favor Alpine unless a real glibc dependency exists
- Dedicated hardware (Raspberry Pi) has effectively one right answer, not a spectrum of options
- Personal familiarity with a distro is not, on its own, a reason to choose it for a given task
- **Next chapter:** Capstone — Setting Up the Same Service on Three Different Distributions

CHAPTER 12 OF 12

Capstone: Setting Up the Same Service on Three Different Distributions

Comparative Linux Distributions

Chapter 12 · Capstone: Setting Up the Same Service on Three Different Distributions

One real service — a small static website served by Nginx — deployed identically in intent across Debian/Ubuntu, Fedora, and Arch. The goal isn't the website itself; it's documenting precisely where eleven chapters' worth of differences actually show up in practice, and — just as tellingly — where they don't.

Step 1 — Installing Nginx (Where the Package Manager Chapter Pays Off)

```
# Debian / Ubuntu
sudo apt install nginx

# Fedora
sudo dnf install nginx

# Arch
sudo pacman -S nginx
```

This is the single most visible divergence across the whole capstone — three genuinely different commands, exactly as Chapter 6 catalogued, for the identical underlying action.

Step 2 — Enabling & Starting the Service (Where It's Genuinely Identical)

```
# Identical on all three — Chapter 8's own convergence point
sudo systemctl enable --now nginx
```

THE ONE STEP THIS CAPSTONE DIDN'T NEED TO WRITE THREE TIMES

Debian/Ubuntu, Fedora, and Arch all converged on systemd, per Chapter 8 — this is the genuine payoff of that convergence. Managing the service itself requires zero distribution-specific knowledge at all, in sharp contrast to Step 1.

Step 3 — Firewall Configuration (A New, Real Divergence)

Fedora runs **firewalld** active by default, requiring an explicit rule to permit HTTP traffic (`sudo firewall-cmd --add-service=http --permanent && sudo firewall-cmd --reload`). Debian/Ubuntu commonly use **ufw**, if installed and enabled at all (`sudo ufw allow 'Nginx HTTP'`). Arch, consistent with Chapter 4's own "you configure everything yourself" philosophy, has no firewall active by default at all — nothing to configure here unless one was deliberately set up beforehand.

Step 4 — SELinux, Fedora Only (Chapter 3's Own Payoff)

Serving files from Nginx's own default document root works immediately on Fedora. Serving files from a custom, non-standard directory instead requires setting the correct SELinux context first — exactly the extra, genuinely necessary step Chapter 3 warned about:

```
# Fedora only — required when serving from a non-default directory
sudo semanage fcontext -a -t httpd_sys_content_t "/srv/mysite(/.*)?"
sudo restorecon -Rv /srv/mysite
```

Neither Debian/Ubuntu nor Arch require this step at all, since neither runs SELinux by default — a direct, concrete instance of Chapter 3's own warning finally showing up in a real deployment.

Step 5 — Verifying the Service (Identical Again)

```
# Identical on all three
curl localhost
```

CHAPTER-ATTRIBUTION TABLE

- **Chapter 2, 3, 4** — the three package managers themselves (Step 1)
- **Chapter 6** — the direct package-manager-syntax comparison, paid off in Step 1
- **Chapter 8** — systemd's convergence across three families, paid off in Step 2
- **Chapter 4** — Arch's own "no default assumed" philosophy, showing up again in Step 3's firewall material
- **Chapter 3** — SELinux enabled by default on Fedora, paid off concretely in Step 4

HONEST SCOPE NOTE

This capstone deliberately covers only three of the five families this course discussed — Alpine (Chapter 5) and Raspberry Pi OS (Chapter 10) both already received their own dedicated, hardware/footprint-specific treatment and don't need a fourth or fifth repetition of the same basic install-a-service exercise. This also stops short of a full

production hardening pass — TLS/certificates belong to the site's own HTTPS/TLS Fundamentals course — and doesn't automate any of this via Ansible, even though Chapter 6 covered its own package-module abstraction; doing so by hand here was deliberately chosen to make each distribution's own real differences visible rather than hidden behind another layer of tooling.

Hands-On Exercises

EXERCISE 1

Explain why Step 2 (enabling and starting the service) required no distribution-specific research at all, while Step 1 required three genuinely different commands.

View solution

EXERCISE 2

A colleague follows this capstone's own steps to serve a site from a custom directory on Ubuntu and is confused why the SELinux commands in Step 4 don't apply or do anything useful there. Explain why.

View solution

EXERCISE 3

Explain why this capstone's own honest scope note says Alpine and Raspberry Pi OS didn't need to be included here, rather than treating their absence as an oversight.

View solution

CHAPTER 12 QUICK REFERENCE

- **Diverges:** package manager (Step 1), firewall tooling (Step 3), SELinux (Step 4, Fedora only)
- **Identical:** systemd service management (Step 2) and basic verification (Step 5) — the real payoff of Chapter 8's own convergence
- Real differences between distributions show up unevenly — some steps genuinely diverge, others don't at all
- This closes the full Comparative Linux Distributions course — twelve chapters spanning four families, package managers, release models, init systems, governance, Raspberry Pi OS, and this final worked capstone