

BACHELOR'S THESIS

Upgrading Linux Systems

Mechanisms, Risks, and Practices for
In-Place and Image-Based System Upgrades

Submitted in partial fulfilment of the requirements
for the degree of Bachelor of Science

Department of Computer Science

September 2026

Abstract

Linux systems in production are never static. Security errata arrive continuously, release lifecycles expire on dates fixed years in advance, and new hardware and application stacks raise the floor that the underlying platform must meet. Each of these pressures is answered by an upgrade, and every upgrade is a change to a running system and therefore a source of regression risk. Administrators consequently face a decision that is seldom stated explicitly: which upgrade mechanism to use, what that mechanism actually does to the machine, and what can still be undone once it has run. This thesis addresses that decision. It combines a structured review of upgrade mechanisms across the major distribution families—the RPM-based systems driven by `dnf` and `leapp`, the Debian-based systems driven by `apt` and `do-release-upgrade`, the SUSE systems driven by `zypper` and `transactional-update`, and rolling releases driven by `pacman`—with a set of controlled experiments performed on a virtual machine, covering ordinary package upgrades, a kernel upgrade with reboot, two distinct rollback paths, and an upgrade carried out under offline constraints from a locally captured repository. The review and the experiments support four findings. First, the available mechanisms differ less in what they install than in where they place the work and what they are able to undo: the same package set can be applied to a live root filesystem, staged into a separate boot environment, or baked into an image, and each placement buys a different form of reversibility. Second, major-version upgrades differ from ordinary patching in kind rather than in degree, because many subsystems change at once and their interactions, not the individual packages, dominate the failure modes. Third, reversibility is determined by state rather than by packages: a transaction can be reversed only to the extent that the state it touched can be restored. Fourth, offline practice succeeds or fails on the completeness of the captured content set rather than on the transfer itself. The contribution is an integrated, reproducible account of these mechanisms together with a command reference and operational checklists derived from them.

Acknowledgements

I would like to thank my supervisor for the guidance, patience, and critical reading that shaped this thesis, and in particular for the insistence that every claim be traceable to either a source or an experiment. I am grateful to the Department of Computer Science for providing the teaching, the infrastructure, and the working environment that made the practical part of this work possible. My thanks also go to the maintainers and documentation authors of the open-source projects studied here; a thesis of this kind depends entirely on the quality of publicly available manual pages, release notes, and upgrade guides, and that quality is the result of sustained and largely unrewarded effort. Finally, I thank my family for their patience during the months in which this work was written.

Contents

Abstract	i
Acknowledgements	ii
List of Figures	vii
List of Tables	viii
List of Abbreviations	ix
1 Introduction	1
1.1 Motivation	1
1.2 Problem statement	2
1.3 Objectives and research questions	2
1.4 Scope and delimitations	3
1.5 Methodology	3
1.6 Contributions	4
1.7 Thesis structure	4
1.8 Summary	5
2 Foundations	6
2.1 Anatomy of a Linux distribution	6
2.2 Release models	6
2.3 Packaging formats	7
2.4 Repositories and metadata	9
2.5 Version comparison semantics	9
2.6 API, ABI and compatibility	10
2.7 The kernel and userspace contract	10
2.8 Summary	11
3 A Taxonomy of Upgrade Operations	12
3.1 Why the Dimensions Must Be Separated	12
3.2 Dimension One: Scope of Change	12
3.3 Dimension Two: Mechanism	12
3.4 Dimension Three: System State During the Change	13
3.5 Dimension Four: Reversibility	13
3.6 Dimension Five: Blast Radius and Orchestration	13
3.7 The Combined Space	14
3.8 A Decision Procedure	14
3.9 Rolling Releases as a Distinct Regime	15

3.10	Criteria for In-Place Versus Re-Provisioning	16
3.11	Summary	16
4	Dependency Resolution and Transaction Mechanics	17
4.1	The Dependency Problem Stated Formally	17
4.2	Solvers in Practice	17
4.3	How a Transaction Is Executed	18
4.4	Configuration File Handling	20
4.5	Obsoletes, Renames and Package Splits	21
4.6	Transaction History, Undo and Its Limits	21
4.7	Control Mechanisms Before the Solver Runs	22
4.8	Summary	22
5	Major-Version In-Place Upgrades	24
5.1	Why a Major Upgrade Is Qualitatively Different	24
5.2	The Leapp Framework	24
5.3	Derivative and Cross-Distribution Migration	26
5.4	Debian and Ubuntu	26
5.5	SUSE	27
5.6	Rolling Systems and the Partial-Upgrade Hazard	28
5.7	Common Inhibitors and How to Clear Them	28
5.8	Post-Upgrade Verification	29
5.9	Summary	29
6	Kernel Upgrades, Reboots, and Live Patching	31
6.1	How Distributions Ship Kernels	31
6.2	The Boot Path and What an Upgrade Touches	31
6.3	Kernel ABI and Out-of-Tree Modules	32
6.4	Live Patching	33
6.5	Reboot Orchestration	34
6.6	Boot Failure Recovery	34
6.7	Summary	35
7	Risk Management, Testing, and Rollback	36
7.1	Framing upgrade risk	36
7.2	Change control in practice	36
7.3	The environment ladder	37
7.4	Canary and progressive rollout for fleets	38
7.5	Testing before and after	38
7.6	Rollback mechanisms ranked by strength	39
7.7	Backups as the last resort	40
7.8	A failure taxonomy with mitigations	40
7.9	Maintenance windows and scheduling	41
7.10	Summary	42
8	Automation and Fleet-Scale Upgrading	43
8.1	From one host to many	43
8.2	Unattended patching	43

8.3	Configuration management and orchestration	44
8.4	Repository control as the real control plane	45
8.5	Air-gapped and offline fleets	46
8.6	Image-based and transactional systems	46
8.7	Measuring an upgrade programme	47
8.8	Summary	48
9	Practical Evaluation	49
9.1	Experimental setup	49
9.2	Experiment 1: baseline characterisation	50
9.3	Experiment 2: an online package upgrade	50
9.4	Experiment 3: kernel upgrade and reboot	51
9.5	Experiment 4: rollback by transaction undo	51
9.6	Experiment 5: rollback by disk snapshot	52
9.7	Experiment 6: upgrading an air-gapped host	53
9.8	Threats to the validity of these measurements	54
9.9	Summary	54
10	Security, Compliance, and Lifecycle Management	56
10.1	Errata as the unit of security maintenance	56
10.2	Vulnerability identification and machine-readable state	57
10.3	Patch cadence, exposure windows, and prioritisation	57
10.4	Lifecycle management and end-of-life	58
10.5	Supply-chain integrity of the upgrade path	58
10.6	Regulated and air-gapped environments	59
10.7	Configuration management as a security control	60
10.8	Summary	60
11	Discussion	61
11.1	Answering the research questions	61
11.2	The central trade-off	62
11.3	Where the models converge	63
11.4	Practical recommendations	63
11.5	Limitations of this study	63
11.6	Threats to validity	64
11.7	Summary	64
12	Conclusion	66
12.1	Summary of the work	66
12.2	Findings against the objectives	66
12.3	Recommendations for practitioners	67
12.4	Future work	67
12.5	Closing statement	68
	Bibliography	72
A	Command Reference	73
A.1	RHEL family: dnf, leapp and system-upgrade	73

A.2	Debian family: apt and dpkg	73
A.3	SUSE: zypper and transactional-update	74
A.4	Arch Linux: pacman	74
A.5	Verification after an upgrade	75
A.6	Repository mirroring and offline capture	75
A.7	Summary	76
Appendices		73
B Operational Checklists		77
B.1	Routine security patching of a single server	77
B.2	Major-version in-place upgrade	77
B.3	Offline or air-gapped upgrade	79
B.4	Rollback rehearsal	79
B.5	Summary	80
C Experimental Data		81
C.1	Reproducing the environment	81
C.2	Baseline and upgrade measurements	81
C.3	Rollback measurements	82
C.4	Detecting an incomplete content set	83
C.5	Index of captured output	83

List of Figures

2.1	Layers of a Linux distribution and the extent of each upgrade class. Package-level upgrades replace content above the kernel while the system runs; kernel upgrades install a parallel kernel that becomes active on reboot; major-version upgrades and image replacement move the entire stack at once.	7
3.1	Decision procedure from operational constraints to an upgrade mechanism. The in-place leaf is served by <code>dnf</code> , <code>apt</code> or <code>zypper</code> , the orchestrated leaf by <code>leapp</code> , <code>do-release-upgrade</code> or <code>zypper dup</code> , the live-patch leaf by <code>kpatch</code> and its equivalents [30], and the image leaf by <code>rpm-ostree</code> , <code>bootc</code> or <code>transactional-update</code> . An air gap is a precondition on content delivery that applies to every outcome rather than a branch of the tree.	15
4.1	The solve-then-execute pipeline. The solve phase is a pure computation over metadata and may be abandoned at no cost; the execute phase writes to the filesystem and runs vendor scripts, and only the transaction test stands between the two.	19
5.1	The Leapp phase pipeline. Everything above the dashed transition runs on the live system and changes nothing; everything below runs after the reboot, with the transaction applied from the upgrade <code>initramfs</code> rather than from the system being replaced.	25
6.1	The boot chain and the artefacts a kernel upgrade writes. A kernel package adds an image, an <code>initramfs</code> and a boot entry; a failure in any one of them is invisible to the package transaction and visible only at the next boot.	32
7.1	The change pipeline for a system upgrade, from request to verified completion. A failed gate returns the system to its previous state and returns the change to the request stage rather than terminating it.	37
8.1	Staged promotion of a repository snapshot. Hosts are pinned to a channel name; promotion moves a snapshot between channels, so no host configuration changes when the content set advances.	45

List of Tables

2.1	Release models of the distributions considered in this thesis.	8
2.2	Metadata and structure of the RPM and Debian binary package formats. . . .	8
3.1	Combinations of scope, mechanism and state, with the tooling that implements them, the qualitative downtime class and the strongest rollback available. . . .	14
5.1	Major-upgrade mechanisms across the four distribution families.	28
6.1	What live patching can and cannot address.	33
7.1	Rollback mechanisms ordered by strength, with what each restores and what it leaves unchanged.	39
7.2	Recurring upgrade failure modes with their principal mitigations.	41
8.1	Package-based, image-based and transactional models for fleet upgrading. . . .	47
9.1	Experimental platform.	49
9.2	Online package upgrade of a freshly deployed host (Experiment 2).	50
9.3	State around the kernel upgrade reboot (Experiment 3).	51
9.4	Disk snapshot rollback (Experiment 5).	52
9.5	Which pre-flight check detects an incomplete captured content set.	54
10.1	Advisory classes across the distribution families.	56
A.1	Package and upgrade commands on the RHEL family.	73
A.2	Package and upgrade commands on the Debian family.	74
A.3	Package and upgrade commands on SUSE systems.	74
A.4	Package and upgrade commands on Arch Linux.	75
A.5	Commands used to verify a system after a transaction.	75
A.6	Commands for capturing and publishing repository content locally.	75
C.1	Baseline of the unmodified image (<code>e1-baseline.txt</code>).	81
C.2	Upgrade transactions, online and offline	82
C.3	Capture of the content set on a connected twin (<code>e6a.txt</code>).	82
C.4	The two rollback mechanisms as measured	82
C.5	Pre-flight checks against an incomplete content set.	83
C.6	Files in <code>experiments/</code>	83

List of Abbreviations

Abbreviation	Expansion
ABI	Application Binary Interface
ACL	Access Control List
API	Application Programming Interface
APT	Advanced Package Tool
BLS	Boot Loader Specification
Btrfs	B-tree File System
CI/CD	Continuous Integration and Continuous Delivery
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
DKMS	Dynamic Kernel Module Support
DNF	Dandified YUM (the RPM package manager of the Fedora and RHEL families)
DNS	Domain Name System
dpkg	Debian Package manager
EOL	End of Life
EPEL	Extra Packages for Enterprise Linux
ESM	Expanded Security Maintenance
EUS	Extended Update Support
FIPS	Federal Information Processing Standards
GPG	GNU Privacy Guard
GRUB	GRand Unified Bootloader
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I/O	Input/Output
IaC	Infrastructure as Code
initramfs	Initial RAM File System
kABI	Kernel Application Binary Interface
KVM	Kernel-based Virtual Machine
LTS	Long-Term Support
LVM	Logical Volume Manager
NFS	Network File System
NIST	National Institute of Standards and Technology
NVD	National Vulnerability Database
OS	Operating System
OSTree	Operating System Tree (the versioned filesystem tree store, <code>libostree</code>)
OVAL	Open Vulnerability and Assessment Language
PXE	Preboot Execution Environment
QEMU	Quick Emulator
RHEL	Red Hat Enterprise Linux
RPM	RPM Package Manager (originally Red Hat Package Manager)

continued on the next page

Abbreviation	Expansion
RPO	Recovery Point Objective
RTO	Recovery Time Objective
SAT	Boolean Satisfiability (problem)
SBOM	Software Bill of Materials
SCAP	Security Content Automation Protocol
SELinux	Security-Enhanced Linux
SHA	Secure Hash Algorithm
SLA	Service Level Agreement
SLE	SUSE Linux Enterprise
SLES	SUSE Linux Enterprise Server
SLSA	Supply-chain Levels for Software Artifacts
SSH	Secure Shell
SUSE	Software und System-Entwicklung (Software and System Development)
TLS	Transport Layer Security
UEFI	Unified Extensible Firmware Interface
URL	Uniform Resource Locator
VM	Virtual Machine
YAML	YAML Ain't Markup Language
YUM	Yellowdog Updater, Modified

1 Introduction

This chapter establishes why the upgrade of a running Linux system deserves systematic study, states the problem the thesis addresses, and defines the questions that the remaining chapters answer. It also fixes the scope of the investigation, describes the method by which evidence is gathered, lists the contributions, and outlines the structure of the document. The aim throughout is to treat upgrading not as a routine administrative chore but as an engineering activity with identifiable mechanisms, measurable properties, and characteristic failure modes.

1.1 Motivation

A Linux server is never finished. From the moment it is provisioned it begins to diverge from the state its operator intended, because the software it runs continues to change even when the machine itself does not. Vendors publish security errata on a continuous cadence, and enterprise distributions formalise this into a documented support policy that distinguishes security advisories from bug fixes and enhancements, each with its own delivery rhythm [41]. Every advisory that is not applied leaves a known, publicly documented defect in place on a machine that is frequently reachable from a network. Patch management guidance from the National Institute of Standards and Technology frames this explicitly as preventive maintenance: the decision is not whether to patch but how to schedule, test, and account for patching as an ongoing programme rather than a series of emergencies [47].

The second pressure is the end of a release’s life. Distributions do not support their releases indefinitely. Fixed-release vendors publish dated lifecycle tables that state when full support, maintenance support, and extended support each expire, and Ubuntu publishes an equivalent cycle for its interim and long-term-support releases [9]. When a release passes its end-of-life date, the stream of errata stops; the machine does not become insecure on that date, but it stops becoming more secure, and every subsequent vulnerability in its package set is permanent. This converts a gradual maintenance problem into a hard deadline, and the migration path offered in response is a major-version upgrade, whether performed in place with a tool such as `leapp` [29] or by re-provisioning the host from a newer image.

Third, upgrades deliver capability rather than only correctness. New hardware frequently requires a newer kernel, because drivers, firmware loading paths, and platform enablement code land in the kernel tree and are backported into vendor kernels only selectively [42]. Application stacks impose their own floors: a runtime that requires a recent `glibc`, a container engine that requires newer namespaces and control group features, or a database packaged only for recent releases all force the platform forward. An administrator who declines to upgrade for operational reasons is therefore also declining hardware support and, in time, the ability to run current software at all.

Against these three pressures stands the fourth force, and the reason the topic is interesting: regression risk. An upgrade is a change, and change is the dominant source of outages in production systems. Release engineering practice in large-scale environments responds by treating each release as a controlled, reproducible, and reversible artefact rather than an ad hoc action [7], and by exposing changes to a small fraction of traffic before a fleet-wide rollout [25]. The tension between the security argument for applying changes quickly and the reliability argument for applying them slowly is the practical core of upgrade management, and it cannot be resolved by policy alone; it requires knowledge of what a given upgrade mechanism actually

does to a system.

1.2 Problem statement

The folk maxim “never touch a running system” encodes a real observation: systems that are not modified do not break in new ways. Yet the maxim is unsound as a policy, because the alternative to changing a system is not stasis but silent decay of its security posture. Security-focused configuration management guidance treats the two sides as a single problem, requiring that changes be controlled, documented, and assessed for their security impact, rather than either forbidden or applied blindly [26]. Server hardening guidance likewise treats timely patching as a baseline control on which other controls depend [46]. The practitioner’s difficulty is that these documents state what must be achieved without describing the mechanisms through which an upgrade succeeds or fails on a particular distribution.

A second gap separates vendor documentation from field practice. Vendor upgrade guides are authoritative, detailed, and written for the supported path: a machine whose packages come from vendor repositories, whose configuration files are unmodified or modified in expected ways, and whose storage layout matches the installation defaults [39, 13]. Real servers deviate. They carry third-party repositories, locally built packages, hand-edited configuration under `/etc`, kernel modules built outside the distribution, and storage layouts inherited from a previous administrator. Where documentation says that unsupported content must be removed before the upgrade, it rarely says what happens if it is not, how the failure presents, or how it is diagnosed and reversed.

Third, the vocabulary is imprecise. The word “upgrade” covers applying a single security erratum, moving to the next point release of a fixed-release distribution, crossing a major version boundary, synchronising a rolling release against a moved repository, and replacing an entire root filesystem image. These operations differ in duration, in blast radius, in whether a reboot is required, and above all in whether they can be undone. Discussing them under one name obscures exactly the properties an operator needs in order to plan: the mechanisms must be named and classified before their risks can be compared.

1.3 Objectives and research questions

The objective of this thesis is to give a structured, evidence-based account of how Linux systems are upgraded, what distinguishes the available mechanisms, and which properties determine the risk of a given upgrade. The account is intended to be usable: it should allow an administrator to identify which class of upgrade is contemplated, to predict what guarantees the chosen mechanism provides, and to plan a recovery path before rather than after the change. Four research questions structure the work.

RQ1. What upgrade mechanisms exist across mainstream Linux distributions, and how do they differ? This question covers package-level transactions driven by `dnf`, `apt`, `zypper` and `pacman`, distribution-level tooling such as `leapp` and `do-release-upgrade`, and image-based or transactional approaches such as `rpm-ostree`, `bootc` and `transactional-update` [12, 8, 49]. The answer takes the form of a taxonomy and a comparative tooling analysis.

RQ2. What makes a major-version upgrade qualitatively harder than a patch-level one? Patch-level upgrades stay within a stable application binary interface and a

stable configuration syntax; major-version upgrades change interpreters, default settings, removed and replaced components, and sometimes the package manager itself. The question asks which differences are incidental and which are structural.

RQ3. What determines whether an upgrade can be rolled back? Package managers record transactions and can reverse some of them [19], but scriptlets, database migrations, and on-disk format changes are not always reversible. Filesystem or volume snapshots and image-based deployments shift the guarantee from the package layer to the storage or boot layer. The question asks what each mechanism actually promises.

RQ4. How do offline and air-gapped constraints change upgrade practice? Systems without direct access to vendor repositories must obtain content through captured mirrors, and must validate its authenticity without an online path. Package managers and their mirror infrastructure have known weaknesses when repository content or metadata is under attacker control [11], which makes the offline case a security question as well as a logistical one.

1.4 Scope and delimitations

The thesis addresses server-oriented Linux systems on the x86-64 architecture. It covers four distribution families that between them represent the dominant administrative models: the Red Hat family with `rpm` and `dnf`, the Debian family with `dpkg` and `apt`, the SUSE family with `zypper`, and Arch Linux as the reference rolling release with `pacman` [5]. Where a statement holds only for one family it is attributed to that family; where the families differ materially, the difference is described rather than averaged away.

Several adjacent areas are deliberately excluded. Desktop-specific concerns, such as upgrading a graphical session in place or migrating user profile data, are out of scope, as are embedded systems with read-only firmware update channels and Android’s partition-based update model, which solve a related problem under very different constraints. Container image build pipelines are excluded as a software-delivery topic in their own right, and are treated only where an image replaces a host upgrade: that is, where the host itself is delivered as an image and updated by deploying a new one [8]. Architecture migration, firmware updates, and application-level schema migration are likewise outside the boundary, although the thesis notes where they interact with the host upgrade.

1.5 Methodology

The work combines two methods. The first is a review of peer-reviewed literature and primary vendor documentation. The literature contributes the theory of dependency resolution [50, 1, 2], the security analysis of update distribution [11, 45], and the rebootless kernel patching work that underpins current live-patching implementations [6]. The vendor documentation contributes the authoritative description of what each tool does, and is cited in preference to secondary sources wherever the two disagree [44, 17, 14, 48, 36].

The second method is a controlled experiment, described in full in Chapter 9. The experimental platform is a single AlmaLinux 9.8 guest running under KVM, chosen because it is a current Enterprise Linux derivative whose package tooling, errata structure, and boot configuration match those documented by the upstream vendor, and because a virtual machine permits the snapshot and restore operations that the rollback experiments require. Four classes

of experiment are performed: package-level upgrades applied through `dnf`, a kernel upgrade including the reboot and boot-entry changes it entails, rollback attempted both through `dnf history undo` and through disk snapshots taken at the hypervisor layer, and an upgrade performed entirely offline from a locally captured repository. Each run is instrumented so that the resulting transaction, its duration, and its effect on the installed package set can be reported. Numerical results are presented only in Chapter 9; earlier chapters refer to that chapter rather than anticipating its figures.

1.6 Contributions

The thesis makes four contributions. The first is a taxonomy of Linux upgrade operations that separates them along the axes that matter operationally: the version distance crossed, whether the running system is modified in place or replaced, whether the operation runs against a live system or in an offline early-boot environment, and whether it is reversible. The taxonomy is developed in Chapter 3 and is used as the organising structure for the rest of the document.

The second is a comparative analysis of the tooling across the four distribution families, covering the transaction model of each package manager, the pre-upgrade analysis offered by major-version tools, and the differing guarantees of the image-based and transactional systems. The third is a body of measured evidence from the virtual-machine experiments, which grounds otherwise qualitative claims about duration and reversibility in observations the reader can reproduce. The fourth is a practitioner checklist, given in Appendix B, that translates the analysis into an ordered sequence of pre-upgrade, upgrade, and post-upgrade actions with an explicit rollback decision point.

1.7 Thesis structure

Chapter 2 establishes the foundations: the anatomy of a distribution, the release models that govern when upgrades become available, the packaging formats and their metadata, repository structure and signing, version comparison semantics, and the ABI stability promises on which the whole arrangement rests. Chapter 3 develops the taxonomy of upgrade classes and the criteria that assign a given operation to a class.

Chapter 4 examines the transaction layer: dependency resolution as a constraint problem, the role of `libsolv` and comparable solvers, and the mechanics of scriptlets, file conflicts, configuration file handling, and transaction history. Chapter 5 treats major-version in-place upgrades, including `leapp`, `ELevate`, `do-release-upgrade`, and `zypper dup`, together with their pre-upgrade analysis models. Chapter 6 covers kernel upgrades specifically: `kABI`, the `initramfs`, boot loader configuration, Secure Boot, live patching, and reboot orchestration.

Chapter 7 addresses risk management, testing strategy, snapshots, backups, and rollback planning, and Chapter 8 scales the discussion to fleets, covering configuration management, automated update daemons, staged mirrors, air-gapped operation, and image-based hosts. Chapter 9 presents the experimental evaluation. Chapter 10 relates upgrading to security and compliance obligations, from errata classification and vulnerability assessment tooling to supply-chain integrity. Chapter 11 synthesises the findings and states the threats to validity, and Chapter 12 answers the research questions. Three appendices provide a command reference, the practitioner checklists, and the raw experimental data.

1.8 Summary

Upgrading is a recurring obligation rather than an occasional project, driven by the continuous publication of security errata, by hard end-of-life dates, and by hardware and software that require newer platform components. It is resisted because change is the principal source of regressions, and the resulting tension is not resolvable by policy statements alone. Vendor documentation describes the supported path precisely but says little about systems that have drifted from it, and the informal vocabulary of “upgrading” conflates operations with very different risk profiles.

This thesis therefore asks what mechanisms exist and how they differ, what makes major-version upgrades structurally harder than patch-level ones, what determines reversibility, and how offline constraints change practice. It answers these through a review of literature and primary vendor documentation combined with controlled experiments on an AlmaLinux 9.8 virtual machine, and it delivers a taxonomy, a comparative tooling analysis, measured evidence, and a practitioner checklist. The next chapter supplies the foundations on which those answers depend.

2 Foundations

This chapter supplies the technical vocabulary the rest of the thesis depends on: the composition of a distribution, the release models that govern when upgrades become available, the two dominant packaging formats, repository publication and authentication, version comparison, and the stability promises the resulting system makes. Each topic reappears as a source of upgrade risk, so the treatment is deliberately mechanical.

2.1 Anatomy of a Linux distribution

A distribution is not a program but an integration: a curated set of upstream projects compiled against one another, packaged with metadata, and shipped with a policy stating how they interact. At the base sits the kernel with its loadable modules, firmware blobs and the initial RAM filesystem that bridges the boot loader and the real root filesystem [21]. Above it sits the C library, almost always `glibc` on servers, which mediates every system call ordinary programs make and so forms the second, and arguably stricter, compatibility boundary in the system.

Above the C library sit the shared libraries and toolchain against which everything else is linked, then the service manager, `systemd` on all four families studied here. Unit metadata rather than the package manager decides whether a service restarts after its binaries are replaced and how that restart is ordered against other services. Above it sit the userland proper — daemons, utilities, language runtimes and applications — and the configuration and policy layer under `/etc`, including the mandatory access control policy and the defaults the distribution chooses on the administrator’s behalf.

The policy is the distribution’s constitution. Debian’s policy manual specifies filesystem layout, dependency semantics, maintainer script behaviour, and the obligations a package assumes towards the rest of the system; conformance is what allows independently maintained packages to be installed together [15]. Enterprise distributions express comparable rules through the support policy defining which interfaces are stable for the life of a release [41].

Figure 2.1 shows these layers and the extent of each upgrade class. They differ in replaceability: the C library can be replaced in place, but processes already linked against the old copy keep using it until restarted, whereas the kernel can only be installed alongside and activated by a reboot or, for a restricted class of changes, patched in memory [30].

2.2 Release models

A release model determines when new content appears, how long old content is maintained, and what an administrator must do to move between versions. Three families dominate, with hybrids between them; Table 2.1 gives the cadence and upgrade path of each distribution studied here.

The fixed or point-release model freezes a package set and then maintains it, backporting security and critical bug fixes into the frozen versions rather than moving to newer upstream releases. Red Hat Enterprise Linux is the canonical case: a major version receives minor releases during its supported life, with the errata policy distinguishing security, bug fix and enhancement advisories and support divided into phases that progressively narrow what will be fixed [41]. Debian follows the same shape with different mechanics: a stable release is published when judged ready and receives point releases that roll up accumulated fixes before the staged

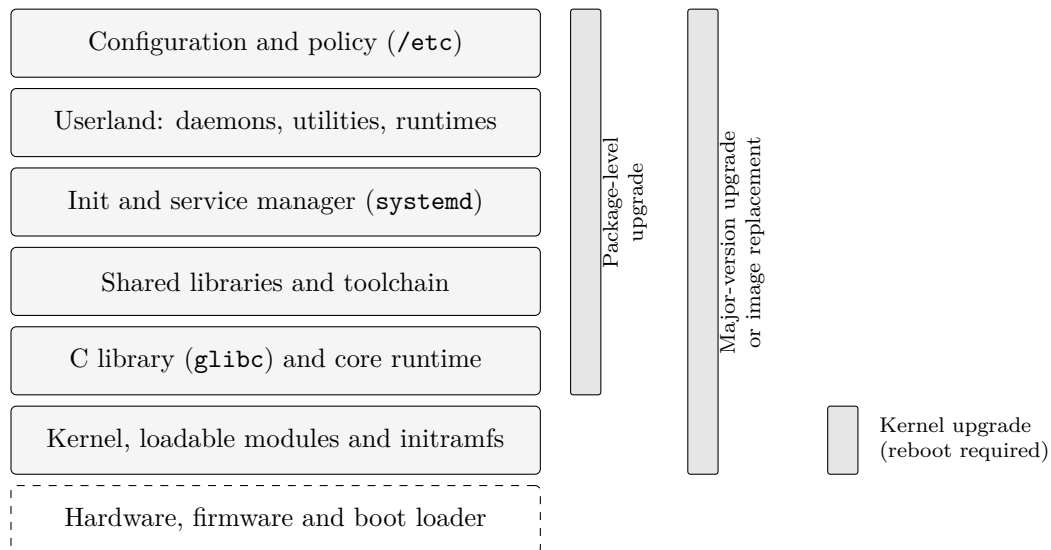


Figure 2.1: Layers of a Linux distribution and the extent of each upgrade class. Package-level upgrades replace content above the kernel while the system runs; kernel upgrades install a parallel kernel that becomes active on reboot; major-version upgrades and image replacement move the entire stack at once.

upgrade documented in the release notes [13].

The time-based model fixes the calendar rather than the content. Ubuntu publishes interim releases every six months and long-term-support releases every two years, with different support durations attached to each, and moves between them with `do-release-upgrade` [9, 10]. The deadline is known years ahead, but a release contains whatever was ready on the date.

The rolling model abolishes the release entirely. Arch Linux ships packages continuously and expects the system to be synchronised against the repositories as a whole with `pacman -Syu`; partial upgrades are explicitly unsupported because the repository is only ever tested as a coherent set [5, 4]. The burden shifts from rare, large transitions to frequent, small ones preceded by reading announcements.

Hybrids combine these properties. openSUSE Leap is built from the same sources as SUSE Linux Enterprise and is upgraded with `zypper dup`, a distribution upgrade permitting package downgrades and vendor changes rather than only forward moves [36, 48]. Tumbleweed is fully rolling and uses the same command.

2.3 Packaging formats

Table 2.2 sets the two formats side by side. An RPM package is a lead, a signature section, a header and a compressed cpio payload; the header is a typed key-value metadata store carrying name, version, release, epoch, architecture, dependency relations, the file list with permissions and digests, and the maintainer scriptlets [44]. Because it is self-describing and separable from the payload, repository indexes can be generated from headers alone. A Debian binary package is an `ar` archive whose control member holds the `control` file, the `md5sums` manifest, the `conffiles` list marking which files are configuration and therefore protected from silent replacement, and the maintainer scripts. Debian policy specifies the meaning of each control field and the states in which each maintainer script may be invoked, which is what makes unattended upgrades tractable at all [15].

Two entries govern upgrades in particular. A *provides* may be a virtual capability such

Table 2.1: Release models of the distributions considered in this thesis.

Distribution	Model	Cadence	Upgrade path
RHEL and derivatives	Fixed, point release	Minor releases during a major version's life; phased support [41]	<code>dnf upgrade</code> within a major version; <code>leapp</code> across one [39]
Debian	Fixed, point release	Stable released when ready; point releases roll up fixes	Edit <code>sources.list</code> , then a staged <code>apt</code> sequence [13]
Ubuntu	Time-based	Interim releases every six months; LTS every two years [9]	<code>do-release-upgrade</code> [10]
SLES / openSUSE Leap	Hybrid, service packs	Leap built from enterprise sources; service pack migrations	<code>zypper dup</code> after repository switch [48, 36]
openSUSE Tumbleweed	Rolling	Continuously published tested snapshots	<code>zypper dup</code> [36]
Arch Linux	Rolling	Continuous; no release boundaries	<code>pacman -Syu</code> for the whole system [5]

as a soname or an abstract service name, decoupling consumers from producers. RPM *obsoletes* tells the resolver that a package supersedes another and should replace it during an upgrade, which is how a distribution renames, merges or splits packages across releases without administrator intervention, and correspondingly a common source of surprise when a package silently disappears; Debian achieves similar effects with *Replaces*, *Breaks* and transitional dummy packages.

Scriptlets are the escape hatch and the principal hazard. RPM packages may carry `%pre`, `%post`, `%preun` and `%postun` scriptlets, each told whether the operation is an install, an upgrade or an erase, plus triggers that fire in one package when another changes state [44]. Triggers handle cross-package migrations, but because they execute arbitrary code during a transaction they fall outside the guarantees the package database otherwise provides: a failed scriptlet may leave the filesystem modified even when the package operation is reported as complete.

Table 2.2: Metadata and structure of the RPM and Debian binary package formats.

Aspect	RPM	Debian binary package
Container	Lead, signature, header, cpio payload	<code>ar</code> archive: version marker, control archive, data archive
Metadata store	Typed header tags [44]	<code>control</code> file in the control archive [15]
Dependencies	<i>Requires</i> , with version ranges	<i>Depends</i> , <i>Pre-Depends</i> , <i>Recommends</i> , <i>Suggests</i>
Virtual capabilities	<i>Provides</i> , including sonames and file paths	<i>Provides</i> , usually abstract names
Supersession	<i>Obsoletes</i> replaces a package on upgrade	<i>Replaces</i> plus transitional dummy packages
Incompatibility	<i>Conflicts</i>	<i>Conflicts</i> and <i>Breaks</i>
File list	In the header, with mode, owner and digest	In the data archive, digests in <code>md5sums</code>
Configuration files	<code>%config</code> or <code>%config(noreplace)</code>	Listed in <code>conffiles</code> , arbitrated by <code>dpkg</code>
Maintainer scripts	<code>%pre</code> , <code>%post</code> , <code>%preun</code> , <code>%postun</code>	<code>preinst</code> , <code>postinst</code> , <code>prerm</code> , <code>postrm</code>
Cross-package hooks	Triggers fire on another package's state change	Triggers in <code>triggers</code> , deferred to end of run

2.4 Repositories and metadata

A repository is a directory tree of packages plus an index that allows a client to plan a transaction without downloading the packages themselves. In the RPM world the index lives under `repodata/`, anchored by `repomd.xml`, which names and checksums a primary index of package headers and dependency relations, a file list index, changelogs and, where published, update records mapping advisories to the packages that satisfy them [40]. Clients configure repositories under `/etc/yum.repos.d/` and cache metadata locally, with expiry governed by `/etc/dnf/dnf.conf` [19, 17]. The Debian equivalent is a `Release` file checksumming per-suite `Packages` indexes, configured in `/etc/apt/sources.list` or under `sources.list.d` [14].

Authenticity is established by signature rather than by transport. The Debian `Release` file is signed, and because it checksums the `Packages` indexes, which in turn checksum the packages, one verified signature covers the whole tree. The RPM world signs both the metadata and the individual packages, so a package remains verifiable after being copied out of its repository — a property that matters for the air-gapped case examined in Chapter 8.

Signing is necessary but not sufficient. Mirror networks serve content from hosts the distribution does not control, and analysis of package manager architectures has shown that an attacker controlling a mirror can mount attacks that signature verification alone does not prevent, including serving stale but correctly signed metadata to freeze a client at a vulnerable version [11]. Later work addressed these weaknesses through explicit freshness guarantees and a role-separated key hierarchy that limits the damage a single compromised key can do [45]. These concerns return in Chapter 10.

Two refinements complicate upgrades. Delta content ships the difference between an installed package and its successor, paying for the bandwidth saving in CPU time and in extra failure modes when the local copy does not match expectations. Modular content, introduced in the RPM ecosystem to let one release offer parallel versions of a runtime, adds a second resolution layer in which streams are enabled and profiles select package subsets [40]. Both introduce state that must be reconciled when the release moves.

2.5 Version comparison semantics

Upgrade decisions reduce to a comparison of two version strings, and the rules are neither obvious nor shared between ecosystems. An RPM version is a triple written `epoch:version-release`. The epoch is an integer that dominates the comparison outright and lets a packager declare a package newer despite an upstream string that sorts lower; an absent epoch is treated as zero. Version and release are each compared by the `rpmvercmp` algorithm, which splits the string into alternating runs of digits and letters, compares numeric runs numerically and alphabetic runs lexically, and treats a numeric segment as newer than an alphabetic one [44].

Debian’s scheme is structurally similar and different in detail. A Debian version is written `epoch:upstream-revision` and is compared in the same alternating way, with the notable rule that the tilde character sorts before everything, including the empty string, which gives packagers a clean way to express pre-releases. The comparison is exposed as a predicate rather than as a number, so scripts ask a relational question and read the exit status, as Listing 2.1 shows [15, 14].

Listing 2.1: Version comparison in the Debian and RPM toolchains; both examples turn on the epoch, which dominates the rest of the string.

```
# Debian: dpkg answers a relational question via its exit status
```

```
$ dpkg --compare-versions 1:1.2.3-1 gt 1.2.3-9 && echo "left is newer"
left is newer

# RPM: rpmvercmp is reachable through the embedded Lua interpreter
$ rpm --eval '%{lua:print(rpm.vercmp("1:1.2.3-1", "1.2.3-9"))}'
1
```

Neither scheme is semantic versioning, which assigns meaning to the position of a number: a major increment signals an incompatible API change, a minor increment backward-compatible functionality, and a patch increment backward-compatible fixes [37]. Distribution version comparison is purely an ordering relation, saying which package is newer and nothing about compatibility, and this gap is the most consequential misunderstanding in upgrade practice. A backported security fix changes only the release field, leaving the upstream version unchanged, so a scanner that reads version numbers as semantic statements reports a patched package as vulnerable. Compatibility is asserted by the ABI discipline described next, not by the version string.

2.6 API, ABI and compatibility

An application programming interface is a source-level contract determining whether code compiles; an application binary interface is a machine-level contract determining whether an already compiled binary keeps running against a replaced library. Upgrades care almost exclusively about the latter, because the installed base is binaries, not sources.

The primary tool of ABI discipline in the userland is the soname, an internal name such as `libssl.so.3` that binaries record in their dynamic section and the dynamic linker resolves at run time through a symbolic link to the concrete file. A library that changes its ABI incompatibly must change its soname, letting the old and new libraries coexist on disk so that a distribution can carry both through a transition. The `glibc` project adds symbol versioning on top of this: individual symbols carry version tags, so one library file can export both an old and a new implementation of the same function and serve binaries compiled against either. This is why binaries built on a newer system frequently fail on an older one while the reverse succeeds.

Enterprise kernels apply the same idea internally. A kernel ABI, commonly abbreviated kABI, is a vendor commitment that a defined subset of in-kernel symbols and structure layouts remains stable for the life of a release, so that out-of-tree modules compiled against one kernel continue to load against its successors [42]. The commitment is vendor policy, not a property of the kernel project, and it is bought by careful backporting.

Verifying such claims by inspection is impractical, so the work is automated. The `libabigail` framework reads the debug information of two builds of the same library and reports the differences in their exported interfaces: added, removed or changed functions and variables, changed types, and changes in symbol visibility [24]. It makes precise what a distribution means when it promises that a package is “the same version” after a security update: the interface a compiled consumer sees is unchanged, not the source. A fix that alters internal behaviour satisfies that promise; one that removes an exported function must wait for the next major release.

2.7 The kernel and userspace contract

The final foundation is the asymmetry between the kernel’s two interfaces. Facing userspace, the kernel maintains a stable system call ABI: a program that ran on an older kernel is expected

to keep running on a newer one, and regressions are treated as defects. This is what makes it possible to install a new kernel under an unchanged userland and reboot into it, and why kernel upgrades can be evaluated largely independently of the rest of the package set.

Facing in-kernel code, the kernel deliberately maintains no stable interface at all. Internal APIs and structure layouts change whenever a better design is found, on the argument that a frozen internal interface would prevent the kernel from improving and would shift the maintenance burden onto the wrong parties [28]. Consequently every out-of-tree module must be rebuilt or re-supplied for each kernel it loads into, and a kernel upgrade that succeeds perfectly at the package level can still leave a machine without its storage controller. Enterprise kABI promises soften this within a release and expire at its boundary; Chapter 6 treats the mechanisms.

2.8 Summary

A distribution is a layered integration of kernel, C library, shared libraries, service manager, userland and policy, and each layer differs in how it can be replaced while the system runs. Release models determine when replacement becomes necessary: fixed releases concentrate change into rare, large transitions, rolling releases distribute it into frequent, small ones, and hybrids sit between. The packaging formats encode the constraints that make automated replacement possible — dependencies, provides, obsoletes, conflicts and file lists — and also the scriptlets and triggers that sit outside those guarantees.

Repositories publish this metadata as signed indexes, establishing authenticity but not freshness and leaving a class of attacks that later chapters revisit. Version strings impose an ordering and, unlike semantic versioning, make no compatibility claim; compatibility is asserted instead through soname discipline, symbol versioning and vendor kABI commitments, and verified with tools such as `libabigail`. The kernel's stable syscall ABI and deliberately unstable internal API explain why kernel upgrades are separable from userland upgrades and why out-of-tree modules make them dangerous. Chapter 3 builds on these foundations to classify upgrade operations.

3 A Taxonomy of Upgrade Operations

The word “upgrade” covers operations that share almost nothing beyond the fact that software on a running machine ends up newer. Applying a single security erratum to `openssl`, moving a fleet to the next major enterprise release, and replacing a read-only root filesystem with a freshly built image differ in duration, in failure mode and in what can be done when they go wrong. This chapter develops a taxonomy along five independent dimensions — scope of change, mechanism, system state during the change, reversibility, and blast radius — and derives from them a decision procedure to apply before choosing a tool. Rolling releases are treated separately at the end, because they occupy this space differently from periodic-release distributions.

3.1 Why the Dimensions Must Be Separated

A flat classification such as “patch, minor, major, rolling” conflates a property of the content with a property of the procedure. A major release upgrade may be performed in place by an orchestrator, by booting into an offline transaction, or by discarding the machine and provisioning a new one: the content is identical, the risk profile is not. Conversely, `dnf upgrade` applies a single erratum and a full minor release transition through the same code path. Treating the dimensions as orthogonal makes it possible to say which property of a planned change is responsible for a given risk.

3.2 Dimension One: Scope of Change

The narrowest scope is the *single package transaction*, in which one package and its unavoidable dependencies change version. The next is the *security erratum set*, a curated group of updates that a vendor publishes together and consumers apply selectively through `dnf upgrade --security` against the errata metadata described by Red Hat, Inc. [41]. A *minor release* upgrade moves the system to the next point release of the same major version; an enterprise distribution is expected to preserve application binary interfaces and configuration semantics across it, so the package manager performs it unaided. A *major release* upgrade crosses a compatibility boundary: interpreters, cryptographic defaults, unit semantics and package names all change, and the vendor documentation for Red Hat Enterprise Linux [39, 38] and Debian [13] treats it as a distinct procedure with its own preparation phase.

The widest scope is a *distribution change*, in which the system moves not only forward in version but sideways to a different vendor. The canonical cases are conversions among the Red Hat Enterprise Linux derivatives, by the ELevate project, which extends the Leapp framework with community-maintained migration data [3], or by conversion scripts such as `migrate2rocky` [43]. Such a change adds to the hazards of a major upgrade the problem that package identities are being reassigned, and must be planned as the most invasive scope of all.

3.3 Dimension Two: Mechanism

Four mechanisms cover essentially all practice. The *in-place package transaction* computes a transaction against the installed package database and applies it to the live root filesystem; this is `dnf upgrade`, `apt full-upgrade`, `zypper dup` and `pacman -Syu`. The *in-place orchestrated upgrade* adds a programme that inspects the system, refuses to proceed on known-bad configura-

tions and rewrites state that a plain transaction cannot; Leapp is the reference implementation, a pipeline of actors running before, during and after the transaction [29]. *Image replacement* abandons per-package mutation of the running root: the system boots a complete filesystem tree composed elsewhere, as with `rpm-ostree upgrade` [12] or `bootc upgrade`, which sources that tree from an OCI container image [8]. Finally, *re-provisioning* does not upgrade the machine at all: a new instance is built from a golden image or from configuration management, traffic is moved to it, and the old instance is destroyed. It alone produces a result independent of the machine's history, which is why it suits long-lived hosts carrying undocumented local changes.

3.4 Dimension Three: System State During the Change

An *online* or live upgrade replaces files while services are running. It is the default for package transactions and the source of a characteristic hazard: a process holding an open descriptor to a deleted library keeps running the old code until it restarts, so what is installed and what is executing diverge. An *offline* upgrade applies the transaction when almost nothing is running; Fedora's system upgrade downloads it during normal operation, reboots into a minimal target in which the package manager runs without a login session, a display server or application services, and reboots again on completion [22]. A *transactional* update goes further: the change is applied to a snapshot of the root filesystem, and that snapshot becomes the running system only at the next boot. SUSE's `transactional-update` implements this on a read-only Btrfs root [49], and image-based systems generalise it by staging a whole deployment. The three states differ in when the window of inconsistency exists: online upgrades distribute a small risk across the whole service lifetime, whereas offline and transactional upgrades concentrate it into a schedulable reboot.

3.5 Dimension Four: Reversibility

Reversibility is the dimension most often assumed rather than verified. The weakest level is *none*, the normal condition of a major in-place upgrade once the transaction has begun. *Package-level undo* reinstalls the previous versions, as `dnf history undo` attempts; its limits are severe enough to be examined in detail in chapter 4. A *filesystem snapshot* taken immediately before the transaction restores the whole root filesystem, reversing scriptlet side effects and configuration rewrites that package-level undo cannot, but also discarding data written afterwards. *Image rollback* selects the previous deployment at boot and is the strongest form available to an in-place system, because the previous tree was never modified; it is bounded only by the number of deployments retained. *Redeploy* is rollback by re-provisioning from the last known-good artefact, the only option whose success does not depend on the state of the failed machine.

3.6 Dimension Five: Blast Radius and Orchestration

The final dimension is how many systems a single decision affects. On a *single host* the upgrade is its own change unit and the only orchestration needed is a maintenance window. In a *cluster with quorum* — an etcd ensemble, a database replica set, a distributed storage system — the upgrade must respect an availability invariant: only a bounded number of members may be down at once, members may need a defined order, and mixed-version operation must be supported throughout the rollout. Across a *fleet* the unit of decision is a cohort rather than a host, and the dominant concerns are staged exposure and the ability to stop; the release engineering literature describes this as progressive rollout with an explicit abort criterion [7, 25], and the first cohort exists precisely to discover the failure that testing missed. Because this

dimension is independent of the other four, a low-risk mechanism applied at once to an entire fleet still produces the worst outcome in the taxonomy.

3.7 The Combined Space

Table 3.1 maps the combinations that occur in practice onto tooling, the system state each implies, a qualitative downtime class and the strongest rollback available. Downtime is given in classes rather than durations, because measured timings belong to the practical evaluation reported later in this thesis.

Table 3.1: Combinations of scope, mechanism and state, with the tooling that implements them, the qualitative downtime class and the strongest rollback available.

Scope	Mechanism and tooling	State	Downtime class	Rollback
Single package	<code>dnf</code> , <code>apt</code> , <code>zypper</code> , <code>pacman</code> transaction	Online	Service restart	Package undo
Erratum set	<code>dnf upgrade --security; unattended-upgrades</code>	Online	Service restart	Package undo
Erratum set	Kernel live patch (<code>kpatch</code>) [27]	Online	None	Patch unload
Minor release	<code>dnf upgrade; apt full-upgrade</code>	Online	Restart plus reboot	Snapshot
Minor release	<code>dnf system-upgrade</code> offline transaction [22]	Offline	Two reboots	Snapshot
Minor release	<code>transactional-update</code> [49]	Transactional	One reboot	Snapshot boot
Major release	<code>leapp upgrade</code> [29]; <code>do-release-upgrade</code> [10]; <code>zypper dup</code> [48]	Offline phase	Extended, two reboots	None in practice
Major release	<code>rpm-ostree upgrade</code> [12]; <code>bootc upgrade</code> [8]	Transactional	One reboot	Image rollback
Major release	Re-provision from image or configuration	New host	Cutover only	Redeploy
Distribution change	<code>ELevate</code> [3]; <code>migrate2rocky</code> [43]	Offline phase	Extended, two reboots	None in practice
Rolling	<code>pacman -Syu</code> [5]	Online	Restart plus reboot	Package cache

Two observations follow. First, the strongest rollback available is a property of the mechanism rather than of the scope: a major upgrade by image replacement is more reversible than a single-package transaction performed in place. Second, the mechanisms offering the best rollback are precisely those that forbid the accumulated local modification which makes long-lived hosts hard to upgrade at all, so the choice concerns the operating model as much as the single change.

3.8 A Decision Procedure

Figure 3.1 turns the dimensions into a procedure driven by constraints an operator can establish before touching the system: the downtime budget, whether a bounded-time rollback is required, how many hosts are affected, and whether the site can reach vendor content at all. An air gap appears as a condition on content delivery rather than as a branch, because it changes not the mechanism but what must be staged locally before any mechanism can run.

The procedure is conservative at two points. It asks about rollback before effort, because an

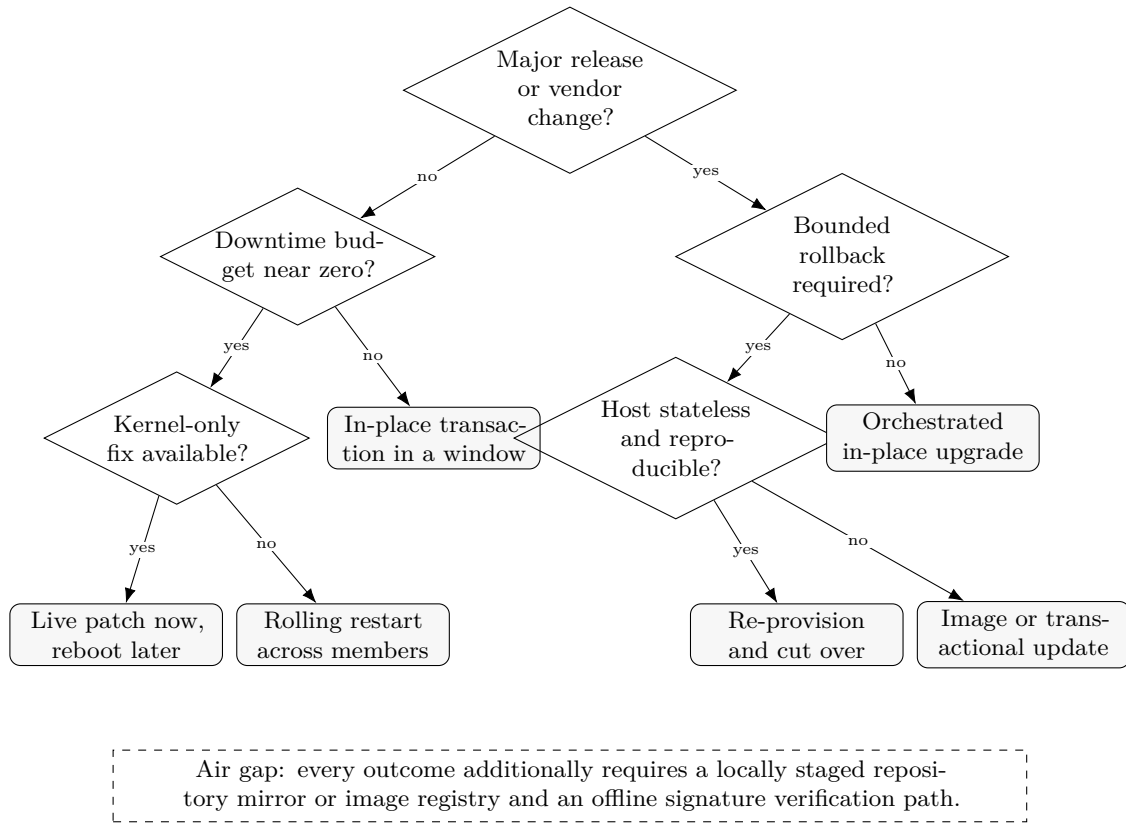


Figure 3.1: Decision procedure from operational constraints to an upgrade mechanism. The in-place leaf is served by `dnf`, `apt` or `zypper`, the orchestrated leaf by `leapp`, `do-release-upgrade` or `zypper dup`, the live-patch leaf by `kpatch` and its equivalents [30], and the image leaf by `rpm-ostree`, `bootc` or `transactional-update`. An air gap is a precondition on content delivery that applies to every outcome rather than a branch of the tree.

upgrade whose failure cannot be undone within the maintenance window is a different kind of change; and it asks whether the host is reproducible before proposing re-provisioning, because rebuilding from a configuration that exists only on the host’s own disk replaces an upgrade problem with a reconstruction problem.

3.9 Rolling Releases as a Distinct Regime

Rolling distributions do not occupy a point in the scope dimension; they remove it. There is no minor or major release event, because every upgrade is a continuous, unversioned move of the whole system toward the current state of the repositories. Arch Linux is the reference case: `pacman -Syu` is a full system upgrade and the only supported way to install a package, since synchronising the package databases without upgrading the installed set produces a *partial upgrade* in which newly installed binaries link against library versions the system does not have [5]. The maintenance guidance adds the operator’s obligations: read the announcements before upgrading, resolve `.pacnew` configuration files as they appear, and never upgrade only what appears to need it [4].

This is a trade rather than an improvement. A periodic release concentrates compatibility breakage into a rare, documented, rehearsable event; a rolling release removes the event and distributes small, unannounced breakages across the whole year. Peak risk falls and the standing requirement for operator attention rises, because no period exists during which skipping upgrades is safe: deferral enlarges the eventual transaction and the number of manual

interventions that may have been missed. For fleets the asymmetry is sharper, since without a version number two hosts upgraded a week apart run materially different systems with no label saying so — precisely the property that staged rollouts exist to prevent.

3.10 Criteria for In-Place Versus Re-Provisioning

The final choice is between changing a machine and replacing it, and five criteria decide it. The first is whether the host's configuration is reproducible from a source of truth outside the host; if it is not, re-provisioning is unavailable whatever its merits. The second is the location of persistent state: a host whose data lives in a database replica or on network storage can be replaced, whereas one whose data lives on its own root filesystem requires a migration step that is itself the risky part of the change. The third is the rollback requirement, since only re-provisioning and image rollback give a reversal of known duration. The fourth is capacity cost, because re-provisioning runs the old and new instances concurrently, which a cloud fleet absorbs easily and a single physical appliance cannot. The fifth is licensing, hardware identity and anything else bound to the specific installation.

Where the criteria conflict, the mechanisms are not exclusive: a common resolution re-provisions the stateless tier and upgrades the stateful tier in place under snapshot protection, so that the fleet gains the reversibility of image replacement while the few irreplaceable hosts receive the more careful treatment. Chapter 5 examines the orchestrated in-place path in detail, and Chapter 8 returns to image-based and transactional systems at fleet scale.

3.11 Summary

This chapter replaced the usual flat vocabulary of patch, minor and major upgrades with five independent dimensions: scope, mechanism, system state, reversibility and blast radius. Separating them shows that risk is not a property of scope alone: the strongest rollback belongs to the mechanism, the most dangerous window to the system state, and the largest possible loss to the orchestration. Table 3.1 and Figure 3.1 translate these observations into a sequence driven by constraints that can be established before any command is run.

Rolling releases were treated as a separate regime rather than a fourth point on a scale, because eliminating the upgrade event exchanges a rare, prepared risk for a continuous, unannounced one, while the criteria separating in-place upgrades from re-provisioning turn chiefly on reproducibility and the location of persistent state. The mechanics on which all of these choices rest — how a transaction is computed and what happens while it is applied — are the subject of chapter 4.

4 Dependency Resolution and Transaction Mechanics

Every upgrade described in chapter 3 reduces to two operations: deciding which set of package versions the system should hold, and changing the filesystem so that it does. The first is a combinatorial search problem of well-studied complexity; the second is an ordered sequence of file replacements and vendor-supplied scripts that can fail in ways the search cannot anticipate. This chapter states the dependency problem formally, describes the solvers distributions ship, follows a transaction through the RPM and dpkg execution models, and examines the parts that most often produce a technically successful upgrade with an operationally wrong outcome: configuration handling, package renames, the limits of undo, and the controls that constrain the solver beforehand.

4.1 The Dependency Problem Stated Formally

A package repository can be modelled as a set of *units*, each identified by name, epoch, version, release and architecture. A unit carries *provides*, capability names it satisfies, including its own name, virtual capabilities such as `webserver`, sonames and, in RPM, absolute file paths; *requires*, each a capability name with an optional version relation; *conflicts*, which forbid the simultaneous installation of a matching unit; and *obsoletes*, which declare that the unit supersedes another.

Given an installed set and a request, the question whether a valid installation exists is the *installability* problem, and it is NP-complete by reduction from satisfiability: each variable of a formula in conjunctive normal form becomes a pair of mutually conflicting packages and each clause a package requiring the corresponding literals. Modern tools therefore abandon hand-written backtracking heuristics and encode the problem for a satisfiability solver. Tucker et al. [50] demonstrated the encoding for Debian, translating dependencies and conflicts into clauses and adding a pseudo-Boolean objective function so that the solver returns not merely a valid installation but an optimal one under a stated criterion. Abate et al. [1] argued that dependency solving should be a concern separate from the package manager itself, with a documented input format and an interchangeable solver, and Abate et al. [2] surveyed the following decade and found complete, SAT-based solvers largely displacing incomplete heuristics across ecosystems.

The distinction between *a* solution and *the right* solution is the practically important one. A repository state usually admits many valid installations, including degenerate ones that satisfy the request by removing much of the system, and a SAT solver has no preference among them. Every real tool therefore layers an optimisation criterion on top: prefer the newest version of the requested packages, then minimise the packages changed, then minimise removals. Differences between package managers are more often differences in this criterion, or in the weight given to a soft rule, than differences in solving power.

4.2 Solvers in Practice

`libsolv` dominates the RPM world, combining a compact dictionary-based representation of package and dependency data with a satisfiability algorithm [35]. Repository metadata and the installed database are read into a *pool* of solvables, the user's request becomes a set of *jobs*, and the library generates and solves requires, conflicts, update and job rules. Hard rules must hold; weak rules the solver may violate if no other assignment exists, which is how a tool reports

keeping a package back rather than failing outright. Both `dnf` and `zypper` are built on this library [17], which is why their behaviour on a difficult repository state is more similar than their user interfaces suggest. The administrator’s controls map onto the rule model directly: `--best` demands the newest candidate, `--nobest` relaxes that demand, `--allowerase` permits the solver to satisfy a job by removing installed packages, and `--skip-broken` drops jobs that cannot be satisfied.

APT’s resolver has a different history: the classic implementation is a heuristic problem resolver rather than a complete search, scoring packages and repeatedly attempting to repair broken dependencies. The manual page documents the resulting asymmetry: `apt upgrade` installs newer versions but never removes an installed package, so packages whose upgrade would require a removal are reported as kept back, whereas `apt full-upgrade` may remove packages in order to complete the upgrade [14]. Removals are therefore not a malfunction but the resolver reporting the only assignment it found, typically because a package has been dropped from the archive or a library transition has made an installed package’s dependencies unsatisfiable. The danger is that the proposal is accepted unread, because the removal list is exactly where a monitoring agent or an in-house package quietly disappears.

Pacman occupies the opposite end of the design space: an Arch repository carries a single version of each package, so there is no version selection to optimise and no meaningful notion of upgrading one package without the rest. The documentation accordingly makes `pacman -Syu` the supported operation and treats the partial upgrade that follows a database synchronisation without an upgrade as unsupported, precisely because the resolver was never asked to consider the resulting mixture [5]. The simplicity is bought by removing the freedom that makes the problem hard elsewhere.

Figure 4.1 shows the resulting pipeline, whose decisive property is the boundary in the middle: to the left nothing has been written and the work is reversible, to the right the system changes.

4.3 How a Transaction Is Executed

Once a set has been chosen, RPM assembles a *transaction set* with one element per package to install, upgrade or erase. The library first checks it for unresolved dependencies and file conflicts, then orders the elements so that a package’s dependencies are installed before it, breaking the dependency loops that are unavoidable among mutually dependent core packages [44]. File conflicts arise when two packages in the same set own the same path with differing content; identical content, or files of a different architecture colour, are permitted. Because the check runs over the whole set, a conflict is reported before anything is written, which gives the transaction its all-or-nothing appearance.

That appearance is only partly justified, because packages also carry scriptlets. `%pre` and `%post` run before and after the new package’s files are installed, `%preun` and `%postun` before and after an old package’s files are removed, and each receives the number of instances that will remain, which is how a scriptlet distinguishes a first installation from an upgrade and from a final removal. `%pretrans` and `%posttrans` bracket the whole transaction, and triggers let one package run code when another changes state [44]. Scriptlets execute arbitrary shell code as root — creating users, migrating databases, restarting services — and their effects lie outside the transaction model, which is why an RPM transaction is not a transaction in the database sense.

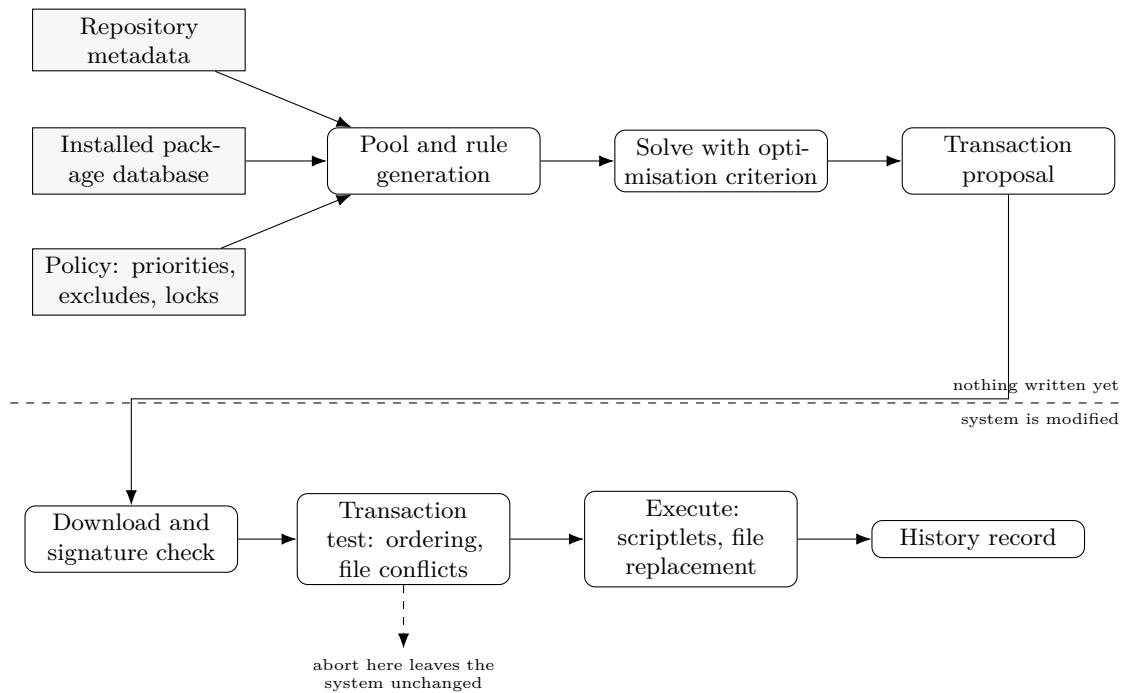


Figure 4.1: The solve-then-execute pipeline. The solve phase is a pure computation over metadata and may be abandoned at no cost; the execute phase writes to the filesystem and runs vendor scripts, and only the transaction test stands between the two.

Debian’s model separates the work into two phases. `dpkg --unpack` extracts a package’s files and runs its `preinst`, leaving the package unpacked but unconfigured; `dpkg --configure` then runs `postinst` and marks it installed. Policy requires that a package be configured only after the packages it depends on are configured, but permits it to be unpacked earlier, which is what allows `dpkg` to break circular dependencies and process a large upgrade in a sensible order [15]. A package may therefore be legitimately present in the half-installed state that `dpkg -l` reports as `iU`, completed only by `dpkg --configure -a`; an interrupted upgrade commonly leaves a system in exactly this condition. Pre-dependencies cover the cases where even unpacking requires another package configured first.

Listing 4.1 shows what the solver reports before execution begins, annotated with the class of decision behind each block.

Listing 4.1: Illustrative `dnf` transaction summary with annotations. Each block corresponds to a different class of solver decision.

```

Dependencies resolved.
=====
Package      Arch  Version      Repo      Size
=====
Upgrading:
glibc        x86_64  2.34-100.el9  baseos    1.9 M
glibc-common x86_64  2.34-100.el9  baseos    24 M
openssl-libs x86_64  1:3.0.7-28.el9 baseos    2.1 M
# Version-locked and protected packages never appear here.
Installing dependencies:
libxcrypt-compat x86_64  4.4.18-3.el9  appstream 92 k
# Pulled in by a requires rule, not asked for by the operator.
Removing:
python3-legacy noarch  1.15.0-9.el9  @System   115 k
# Removal implies --allow-erase or an Obsoletes; verify why.
Skipping packages with broken dependencies:
vendor-agent  x86_64  3.2.1-1.el9   vendor    4.4 M
  
```

```
# A weak rule was violated: the job could not be satisfied and
# the package stays at its old version. This is the line that
# most often explains a post-upgrade outage.
```

Transaction Summary

```
=====
Install      1 Package
Upgrade      3 Packages
Remove       1 Package
Skip         1 Package
```

Listing 4.2 shows the corresponding Debian case, where the only assignment the resolver found deletes installed packages.

Listing 4.2: Illustrative `apt full-upgrade` output in which the resolver proposes removals.

The `REMOVED` block must be read before the prompt is answered.

```
$ sudo apt full-upgrade
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Calculating upgrade... Done
The following packages will be REMOVED:
  libfoo2 python3-legacy-plugin site-monitoring-agent
The following NEW packages will be installed:
  libfoo3 python3-newlib
The following packages will be upgraded:
  base-files libc-bin libc6 openssl
4 upgraded, 2 newly installed, 3 to remove, 0 not upgraded.
After this operation, additional disk space will be used.
Do you want to continue? [Y/n]
```

4.4 Configuration File Handling

Configuration files are the one class of package content the administrator is expected to modify, so both packaging systems treat them specially. RPM marks them `%config`, optionally with the `noreplace` attribute, and compares the file as shipped, as it exists now, and as it appears in the new package. An unmodified file is simply replaced. If it was modified and the package uses plain `%config`, the local file is renamed with an `.rpmsave` suffix and the new version installed; under `%config(noreplace)` the local file is kept and the new version written beside it with an `.rpmnew` suffix [44]. Debian's `conffile` mechanism instead prompts interactively when a locally modified `conffile`'s shipped version has changed, offering to keep the current file, install the maintainer's version, show a difference or open a shell [15]. Non-interactive runs select a policy through `--force-confold`, `--force-confnew` or `--force-confdef` under `/etc/apt/apt.conf.d/`, and the `ucf` helper extends the same three-way logic to files a maintainer script generates rather than ships. Listing 4.3 contrasts the two reports.

Listing 4.3: Configuration divergence as the two packaging systems report it.

```
# RPM, %config(noreplace): the local file wins, silently
warning: /etc/ssh/sshd_config created as
  /etc/ssh/sshd_config.rpmnew
# RPM, plain %config: the local file is moved aside
warning: /etc/chrony.conf saved as /etc/chrony.conf.rpmsave

# dpkg: the same situation, but the operator is asked
Configuration file '/etc/ssh/sshd_config'
==> Modified (by you or by a script) since installation.
==> Package distributor has shipped an updated version.
What would you like to do about it ? Your options are:
```

```

Y or I : install the package maintainer's version
N or O : keep your currently-installed version
D      : show the differences between the versions
Z      : start a shell to examine the situation
The default action is to keep your current version.
*** sshd_config (Y/I/N/O/D/Z) [default=N] ?

```

The operational consequence is easy to miss. Under `%config(noreplace)` and under an automated `--force-confold`, the upgrade exits zero while the service keeps running its old configuration, and the vendor's new defaults — possibly the hardened cryptographic settings that were the point of the upgrade — sit unread in an `.rpmnew` file. The divergence stays silent until the next restart, when the service may fail on a directive the new binary no longer accepts. A review step therefore belongs in every upgrade procedure: enumerate the leftovers with `find /etc -name '*.rpmnew' -o -name '*.rpmsave'`, merge them deliberately, and use `needs-restarting` to identify services still running against replaced binaries [20].

4.5 Obsoletes, Renames and Package Splits

A substantial share of upgrade failures originates not in the solver but in the metadata telling it how package identities have changed. In RPM, a package declares **Obsoletes** to state that it supersedes another, and `dnf` honours these during upgrades, removing the old package and installing the new one in a single step. When the declaration is missing or incorrect, the old package survives, still owning its files, and may shadow the replacement. Debian expresses the same intent with **Provides**, **Replaces** and **Conflicts**, usually with a transitional dummy package depending on the renamed one [15].

Package splits are the more insidious variant. When a package is divided into a smaller base and one or more subpackages, an upgrade installs the base, satisfies every dependency, reports success, and leaves the system without files that used to be present, because nothing requires the subpackage. Where the relationship is a weak dependency, behaviour depends on configuration: `dnf` installs recommended packages only when `install_weak_deps` is enabled in `/etc/dnf/dnf.conf`, so a minimised image that disables it diverges from a default installation over successive upgrades. Across a major release the same problem appears as renames, precisely the class of change an orchestrated upgrade framework encodes as explicit mappings rather than leaving to the solver, as chapter 5 describes.

4.6 Transaction History, Undo and Its Limits

DNF records every transaction in a history database and exposes it through `dnf history list`, `dnf history info`, `dnf history undo` and `dnf history rollback`, the last of which reverts every transaction performed after a given one [19]. Red Hat's documentation presents these as the normal way to reverse an unwanted package change [40], and for a recent, self-contained transaction they work as advertised.

The limits follow from the mechanics described above and are severe. First, undo is a package operation, not a state operation: a database schema migration performed by a scriptlet or by the service on first start survives it and is then presented to the older binary. Second, scriptlet side effects are not reversed, because RPM has no inverse for arbitrary shell code; users created, caches rebuilt, units enabled and configuration migrated by `%post` remain. Third, undo requires the old packages to be obtainable, and once a repository has moved on and the local cache has been cleaned it fails unless the administrator accepts partial reversal with

`--skip-unavailable`. Fourth, the manual states that if some transactions cannot be undone given the current state of the RPM database, none of them are undone [19]. `dnf history undo` is therefore a convenience for package mistakes and not a rollback mechanism; genuine rollback requires the snapshot or image mechanisms classified in chapter 3 and planned in chapter 7.

Transaction replay is more useful in practice. DNF can store a completed transaction to a file and replay it elsewhere, turning a transaction verified on a staging host into a reproducible artefact for the rest of a cohort, subject to the same constraint that the stored packages remain obtainable on the target [19]. Replay expresses intent explicitly rather than trusting two hosts to resolve identically against a repository that may have moved between the runs.

The Debian side offers less. APT keeps human-readable records under `/var/log/apt/`, in `history.log` and `term.log`, and retains downloaded archives in `/var/cache/apt/`, so a reversal means reinstalling explicit versions with `apt install pkg=version`; no single command reverts a transaction. Zypper keeps its history under `/var/log/zypp/` and, on a Btrfs root managed by Snapper, pairs each transaction with a filesystem snapshot, a materially stronger guarantee because it restores the whole root rather than the package set.

4.7 Control Mechanisms Before the Solver Runs

Because the solver optimises within the space it is given, the most effective administrative controls shape that space in advance. Repository priorities come first: the `priority` and `cost` options in a `/etc/yum.repos.d/*.repo` file, the equivalent zypper priorities and APT's pinning through `/etc/apt/preferences.d/` decide which source wins when several offer the same package, and so keep a third-party repository from overriding a vendor's base packages. Exclusions belong to the same family, expressed as `excludepkgs` in `/etc/dnf/dnf.conf` or as `IgnorePkg` in `/etc/pacman.conf` [5].

Protection mechanisms prevent catastrophic removals. DNF reads `/etc/dnf/protected.d/` and refuses to remove the packages listed there, including itself and the running kernel, while `installonly_limit` bounds how many kernels are retained; Debian's equivalent is the `Essential` flag. Version locking is the narrower instrument: the `dnf versionlock` plugin pins packages listed in `/etc/dnf/plugins/versionlock.list`, zypper `addlock` does the same on SUSE, and `apt-mark hold` on Debian. Module streams in the RHEL 8 generation are a coarse form of the same idea: an enabled stream fixes which application version the solver may consider, and streams must be reset before a major upgrade because the target release does not define them [29].

All of these controls share a failure mode: each is invisible in the final transaction summary, which reports what the solver decided but not what it was forbidden to consider, and each tends to be applied during an incident and never removed. A lock placed to defer one awkward update becomes, two years later, the reason a major upgrade cannot be computed at all. Auditing locks, holds, exclusions and enabled streams therefore belongs at the start of any upgrade procedure.

4.8 Summary

This chapter followed an upgrade from the resolver to the filesystem. Dependency resolution was stated as the installability problem, shown to be NP-complete, and connected to the SAT-based approach the shipped tools now implement: `libsolv` beneath `dnf` and `zypper`, a heuristic resolver in APT whose proposed removals are reports rather than errors, and `pacman`'s

restricted model in which only a full system upgrade is meaningful.

Execution is weaker than the word “transaction” suggests. RPM orders its transaction set and detects file conflicts before writing, but scriptlets run arbitrary code with effects outside the model, and dpkg’s two-phase design admits states in which packages are present but unconfigured. Configuration handling compounds the problem, since both `%config(noreplace)` and `--force-confold` produce successful upgrades in which a service still runs its previous configuration. Obsoletes, renames and splits move failures into the metadata, and history provides an undo that cannot reverse data changes, scriptlet effects or unavailable packages. These mechanics explain why the major-version procedures examined in chapter 5 devote most of their effort to the phase before the transaction begins.

5 Major-Version In-Place Upgrades

The preceding chapters treated an upgrade as a transaction over a package set: a solver computes a consistent state, and the package manager applies it. A major-version upgrade breaks that framing, moving a system not along a maintained release line but onto a different line whose toolchain, system libraries, service definitions and security defaults were developed independently of the ones installed. This chapter examines what makes that transition qualitatively different from the routine updates of chapter 4, and how the four distribution families mechanise it: the Leapp framework in detail, then derivative migration tooling, the Debian and Ubuntu sequence, the SUSE model and rolling releases, closing with the recurring inhibitors and the verification that must follow the first boot.

5.1 Why a Major Upgrade Is Qualitatively Different

The defining property of a major upgrade is simultaneity. Within a maintained release a vendor holds the compiler, C library, unit semantics and default cryptographic policy constant, backporting fixes into that frozen environment. A major upgrade replaces all of them at once: the C library gains new symbol versions and behavioural corrections that previously tolerant code fails against, the compiler and its default flags change, and whole subsystems may be swapped for different implementations. Their concurrence defeats the intuition, drawn from routine patching, that a failure can be attributed to one cause and reverted alone.

Three categories are invisible to dependency resolution. The first is init unit semantics: a service unit may be replaced by one with a different name, default ordering, sandbox or notion of a successful start, so the transaction succeeds and the dependency graph is satisfied, yet the service fails at boot because a directory it assumed writable is now covered by a hardening directive. The second is the default cryptographic policy: enterprise releases progressively retire weak algorithms, key sizes and protocol versions through a system-wide policy consumed by the TLS libraries and the SSH daemon [39, 38], so connections that worked before fail against an unmodernised peer, often surfacing at the far end of the link rather than on the upgraded host.

The third category is the package set. Between major releases packages are removed, renamed, merged, split or moved into an optional repository. A rename is usually handled by an obsoletes relation, but a removal is not: the functionality ceases to exist, and any local integration built on it must be rewritten. Changed defaults compound this, since a configuration file whose default changed silently adopts the new value if never edited, whereas an edited one is preserved and the host diverges from every freshly installed peer. This is how a uniform fleet becomes heterogeneous.

5.2 The Leapp Framework

Leapp is the framework Red Hat Enterprise Linux and its derivatives use for major-version in-place upgrades. Its design decision of interest is that the upgrade logic is not a monolithic script but is decomposed into independent units called actors, each declaring the message types it consumes and produces [29]. Actors communicate only through typed messages, so the framework, not the author, determines execution order. The consequence is extensibility: support for a new risk or target release arrives as actors and data rather than edits to a central procedure, which makes the framework reusable by the projects of section 5.3.

5.2.1 Phases and the Two-Stage Execution Model

Actors are grouped into phases executed in the fixed workflow of Figure 5.1. Facts collection gathers the state of the running system into messages, the checks phase produces findings from those facts, and the remaining pre-reboot phases resolve the target transaction, download its content and build a special initramfs. The system reboots into that initramfs, the RPM transaction is applied there, the application, third-party, finalisation and first-boot phases complete the migration [29].

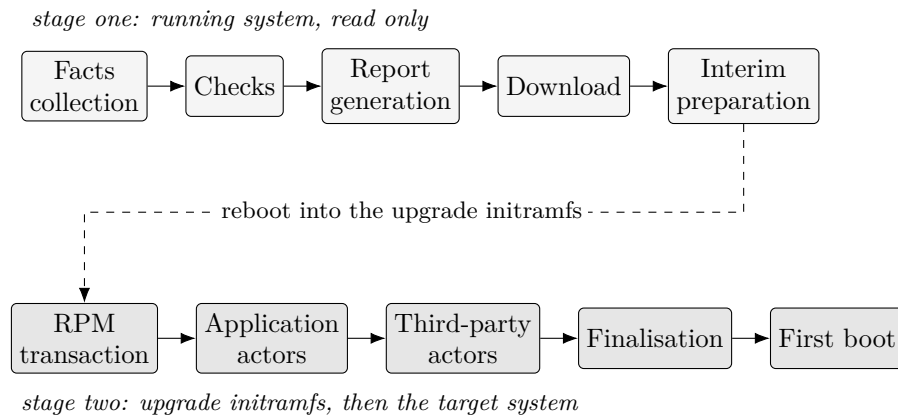


Figure 5.1: The Leapp phase pipeline. Everything above the dashed transition runs on the live system and changes nothing; everything below runs after the reboot, with the transaction applied from the upgrade initramfs rather than from the system being replaced.

The split into a read-only stage and a mutating stage is the framework’s central safety property. `leapp preupgrade` executes only the phases above the reboot boundary: it collects facts, runs checks and writes a report, performs no package operation, and leaves the system bootable in its original state. The analysis is repeatable and usable as a fleet-wide survey long before a change window is scheduled.

5.2.2 The Pre-Upgrade Report

The report is written to `/var/log/leapp/leapp-report.txt` and to a JSON document for machine consumption. Findings carry a risk factor and a distinction between inhibitors and warnings: an inhibitor is a condition the framework judges incompatible with a successful upgrade, and while any is present `leapp upgrade` refuses to proceed, whereas a warning records a condition the operator must understand but which does not block execution [29]. Other findings are questions the framework cannot answer on the operator’s behalf, recorded in an answer file completed with `leapp answer`. Listing 5.1 shows an entry in abbreviated form.

Listing 5.1: Structure of an entry in a Leapp pre-upgrade report. Wording varies by finding; the fields shown are those an operator triages on.

```

Risk Factor: high (inhibitor)
Title: Packages not signed by the distribution vendor were found
Summary: The listed packages come from a repository the upgrade
        tooling does not recognise. They have no counterpart in the
        target release and their behaviour after the upgrade is
        undefined.
Remediation: [hint] Remove the packages, or provide a target
        repository carrying a compatible version, then rerun the
        pre-upgrade check.
Key: <opaque identifier used to correlate findings>

```

Triage proceeds by risk factor rather than report order: inhibitors first, because nothing else can be tested until they are cleared, then warnings. The report is regenerated on every run, so the workflow is iterative: clear a finding, rerun, and confirm that no new one has appeared behind it.

5.2.3 Why the Transaction Runs Outside the Running System

The most consequential decision in Leapp is that the RPM transaction is not applied to the running system. Replacing the C library, the package manager's own interpreter and the init system while those components execute is inherently fragile: a scriptlet may invoke a binary already replaced by an incompatible version, and an interrupted transaction cannot be finished because the tools to do so are half-replaced. Leapp instead builds a self-contained upgrade initramfs, reboots into it, mounts the real root filesystem and runs the transaction against that mounted tree from outside [29, 39], so nothing on the target filesystem executes while it is rewritten.

This is the offline update principle [22] at much larger scale, and it buys atomicity at the cost of observability: during the transaction the system has no network, no logging daemon and no SSH access, so a remote operator is blind until the machine either returns or fails to. Out-of-band console access is therefore a precondition, a point developed in chapter 6; the offline stage is logged under `/var/log/leapp/`.

5.3 Derivative and Cross-Distribution Migration

Because Leapp's logic lives in actors and data rather than a fixed procedure, it can be retargeted. The ELevate project maintains a community-developed actor set and the metadata describing migration paths among the enterprise Linux derivatives, so a host running one derivative can be upgraded to a newer major version of the same or another [3]. The workflow mirrors the vendor one; what differs is the provenance of the metadata mapping old packages to new ones.

Where ELevate moves a system forward a major version, `migrate2rocky` and its equivalents move a system sideways between derivatives of the *same* major version [43]. No release boundary is crossed, so the toolchain and the ABI do not change: the script replaces the release and repository packages that identify the distribution and then issues a distribution synchronisation, reconciling every installed package against the new vendor's build of the same version. Because the two sets are built to be binary compatible, the operation is far narrower than a major upgrade and should not be presented as equivalent.

Both paths carry an organisational rather than technical consideration. A vendor-supported upgrade is one the vendor has tested, documents and will assist with, whereas a community path is tested against configurations that may not resemble a given estate: such paths do not therefore fail, but an unexpected result has no support contract behind its recovery.

5.4 Debian and Ubuntu

Debian ships no dedicated upgrade framework. Instead the release notes for each stable release document a sequence the administrator executes, normative rather than advisory [13]. It begins on the current release: the system is fully updated, held packages are reviewed with `apt-mark showhold`, an interrupted `dpkg` state is repaired with `dpkg --audit`, and free space is confirmed, since the transaction holds the archives and both versions of many files. Third-party repositories and pinning rules are then disabled, because packages from outside the distribution

have no defined relationship to the target release’s dependency graph.

The upgrade itself runs in two movements, whose distinction is the semantic difference at the heart of `apt`: a plain `apt upgrade` installs newer versions of already-installed packages but never removes one, whereas `apt full-upgrade` may remove installed packages when that is necessary to resolve the new release’s dependency graph [14]. A minimal upgrade adding no new packages therefore moves the core of the system first, and only then does the full upgrade complete the transition [13], as Listing 5.2 shows.

Listing 5.2: The documented shape of a Debian release upgrade. Release names and release-specific interventions come from the target release’s notes.

```
# 1. Reach a clean, fully updated state on the current release.
apt update && apt full-upgrade
dpkg -audit
apt-mark showhold

# 2. Point the sources at the target release, and disable
#   third-party sources under /etc/apt/sources.list.d/.
#   (Either sources.list or the deb822 .sources format.)
apt update

# 3. Minimal upgrade: no new packages, no removals.
apt upgrade -without-new-pkgs

# 4. Full upgrade: removals permitted where required.
apt full-upgrade

# 5. Reboot, then review obsolete packages and .dpkg-dist files.
```

Ubuntu wraps an equivalent sequence in `do-release-upgrade`, which fetches an upgrade tool published for the target release, so the logic executed is what the target release’s maintainers shipped rather than whatever was present on the old system [10]. The tool disables third-party repositories for the run, reporting them so they can be restored deliberately, and takes precautions for remote sessions, starting a second SSH daemon on an alternative port so that a dropped connection does not abandon the upgrade. Which upgrades are offered is governed by the `Prompt` setting in `/etc/update-manager/release-upgrades`, and availability is staged: a path opens once the maintainers judge it ready, which for the long-term-support-to-long-term-support path is not when the new release appears [10, 9]. In both distributions the release notes carry manual interventions that no tool performs, and skipping them is a common cause of failure.

5.5 SUSE

SUSE draws an explicit line between updating within a release and moving to a new one, and expresses it in two subcommands. `zypper update` installs newer versions of installed packages while preserving the installed set and the package vendor, and will not remove a package or change its vendor to resolve a conflict. `zypper dup`, the distribution upgrade, synchronises the whole installed set against the configured repositories: it may downgrade a package if that is what the target repository carries, removes packages the target release no longer provides, and is the mechanism by which a vendor change is negotiated [36]. It is therefore correct exactly when those repositories describe the release one intends to run, which is why the documented procedure changes them first and verifies the resulting list before anything is applied.

On openSUSE Leap that change is commonly expressed through the `-releasever` variable, which substitutes the target version into the repository URLs so the whole set moves together [36].

On SUSE Linux Enterprise the equivalent is service pack migration through the subscription tooling: the system registers, the available targets are enumerated, and the selected target’s repositories are activated before the distribution upgrade runs [48]. Such a migration must start from a supported service pack; migrating from one that has left support is not a tested path.

5.6 Rolling Systems and the Partial-Upgrade Hazard

A rolling distribution has no major upgrade because it has no releases to move between, but it has a hazard the versioned distributions largely avoid: the partial upgrade. Arch Linux builds its packages against the current versions of their dependencies and does not, as policy, support a package set drawn from two points in time. Refreshing the databases and then installing a single package from them — `pacman -Sy` followed by an install — produces exactly that state: a new binary linked against a library version the system does not yet have [5, 4]. The supported operation is the full system upgrade, `pacman -Syu`, and the failure mode of ignoring it is a system that cannot run the tools needed to repair it.

The second obligation is that upgrades are not unattended. Manual interventions are announced before they are required, and an administrator who upgrades without reading them may find that a change in a core package’s layout demands action the package manager cannot take alone [4]. The rolling model therefore distributes, as continuous small effort, the risk the versioned model concentrates into a single window. Table 5.1 compares the four families.

Table 5.1: Major-upgrade mechanisms across the four distribution families.

Mechanism	Direction supported	Report first	Rollback and support
leapp	One major version forward, on the vendor’s release line	Yes	No built-in revert; external snapshot or image; vendor supported [29, 39]
ELevate, migrate2rocky	Forward across derivatives, or sideways within a major version	Partly; ELevate inherits the Leapp report	As above, but community maintained [3, 43]
apt full-upgrade, do-release-upgrade	One release forward; skipping a release is undocumented	No analysis stage; release notes fill the role	Restore from backup or snapshot [13, 10]
zypper dup, service pack migration	Forward to the release the configured repositories describe	Dry-run and transaction summary	Snapshot rollback where the root filesystem supports it; vendor supported on the enterprise product [48, 36]
pacman -Syu	Not applicable; continuous	No	Package cache only; community supported [5, 4]

5.7 Common Inhibitors and How to Clear Them

Although only the Leapp family names them, the blocking conditions recur across all four families, so an operator who knows them can anticipate a report. Third-party packages are the most frequent: the tooling has no mapping from such a package to a counterpart on the target release, and a third-party kernel module is worse still, since it must be rebuilt against a kernel

that did not exist when it was written. The remedy is to establish beforehand whether the supplier has a build for the target release, then remove and reinstall the package around the upgrade.

Storage is the second category: filesystems and volume managers the target release no longer supports block an upgrade because the resulting system might not mount its own data. Encrypted volumes need separate attention: the key material and the unlocking mechanism must survive into the new initramfs, and a network-bound scheme must still reach its server from there. Out-of-support hardware drivers are related: a device whose driver was removed from the target kernel disappears at the next boot rather than during the transaction, and if it is a storage controller or a network interface the host does not come back.

The remaining categories are configuration rather than hardware. Network configuration migration inhibits an upgrade where the target release changes the configuration back end or the interface naming scheme, since a host whose interface is renamed loses its address. Deprecated authentication back ends are similar: a directory client, a name service module or a mechanism the target release has retired leaves a host that boots but at which nobody can log in. Both fail only after the reboot, which is why a console and a locally valid emergency account are verified before the change window opens.

5.8 Post-Upgrade Verification

A successful boot begins the verification rather than ending the upgrade. The first check is the service inventory: `systemctl --failed` lists units that did not start, and `systemctl list-unit-files` is compared against a pre-upgrade capture to find units disabled, masked or renamed. The first boot's journal is read in full rather than grepped, since the most informative messages come from units that started but complained.

Configuration reconciliation follows. On RPM systems the transaction leaves the `.rpmnew` and `.rpmsave` files whose opposite meanings section 4.4 sets out, so both must be located and each decided individually [44]. Debian presents the equivalent as `.dpkg-dist` and `.dpkg-old` files [15], SUSE writes `.rpmnew`, and Arch writes `.pacnew`, for which a merge helper exists [4].

Three further checks close the procedure. The effective cryptographic policy is confirmed to be the one intended, and any peer the host must still reach with a weaker protocol is identified explicitly rather than during an outage. Loaded kernel modules are compared against the pre-upgrade set, since a module that failed to rebuild is silently absent rather than loudly broken. Finally, the package database is queried for packages still carrying the old release's identifiers: on RPM a query on the release field, on Debian a listing of packages no longer available from any source. Each such leftover receives no further security updates, which connects this procedure to the lifecycle obligations of chapter 10.

5.9 Summary

A major-version upgrade replaces the toolchain, the system libraries, the service semantics and the security defaults of a host in one operation, so the resulting failures are not attributable to a single package. Leapp answers this with the most mechanisation: actors and phases, a read-only analysis stage separated from a mutating one, and a transaction applied from an upgrade initramfs, with a pre-upgrade report distinguishing inhibitors from warnings. The Debian family substitutes a normative documented sequence, SUSE makes the same distinction explicit in its command set, and rolling distributions replace the problem with the obligation

to upgrade fully rather than partially.

Across all four the inhibiting conditions are predictable: third-party content and out-of-tree kernel modules, unsupported storage, encrypted volumes, migrated network configuration, retired authentication back ends and removed hardware drivers. All fail after the reboot, when the usual diagnostic channel may itself be gone, which makes out-of-band access a precondition rather than a contingency. None of these tools offers a genuine revert; rollback comes from the snapshot or image taken beforehand, a dependency that motivates chapter 7, while the kernel-specific half of the problem is the subject of chapter 6.

6 Kernel Upgrades, Reboots, and Live Patching

The kernel occupies a position no other package holds: it cannot be replaced in the running system by overwriting files and restarting a service, because the code being replaced is the code executing the replacement. Every distribution therefore treats it as a special case in its package manager, its boot loader integration and its support policy. This chapter examines how kernels are shipped and retained, what a kernel package changes on disk, why out-of-tree modules dominate kernel-upgrade failures, how live patching narrows but does not eliminate the need to reboot, how reboots are orchestrated at scale, and what must be in place before a remote kernel upgrade is attempted.

6.1 How Distributions Ship Kernels

The first design decision every distribution makes is that kernel packages install in parallel rather than replacing one another. Where an ordinary upgrade removes the old version's files, a kernel upgrade adds a second kernel alongside the first, with its own image, initramfs, module directory under `/lib/modules/` and boot entry. The reason is recovery: if the new kernel does not boot, the only tool available is the boot loader menu, and that menu is useful only if a known-good kernel is still installed with its modules and initramfs on disk.

The mechanism differs by family. On RPM systems the package manager marks kernel packages as install-only and keeps a bounded number of them, governed by `installonly_limit` in `/etc/dnf/dnf.conf`; once the limit is reached, installing a new kernel removes the oldest, though the running kernel is protected [40, 42]. Debian and Ubuntu achieve parallel installation through naming instead: each kernel is a distinct package whose name embeds the version and ABI number, so `linux-image-...` packages coexist and a metapackage depends on whichever is current [14]. Old kernels then accumulate until removed, a recurring problem on a small `/boot`. SUSE is configuration-driven, with a multiversion setting in `/etc/zypp/zypp.conf` declaring kernel packages multiversion-capable and a retention policy. Arch replaces its single kernel package on upgrade, and administrators wanting a fallback install a separately named long-term-support kernel alongside it [5].

Retention policy is therefore a decision, not a default to be ignored. Keeping too few kernels removes the recovery path; keeping too many exhausts `/boot`, and a `/boot` that fills mid-transaction produces a truncated initramfs and an unbootable system.

6.2 The Boot Path and What an Upgrade Touches

Installing a kernel package writes considerably more than a kernel image. Its scriptlets generate an initramfs, register a boot entry and, on systems using a boot loader configuration file, regenerate that file. Each is a place where the transaction reports success while the system has become unbootable. Figure 6.1 shows the chain and the artefacts each stage consumes.

The initramfs is generated by `dracut` on RPM systems and by equivalent tooling on Debian systems. It must contain exactly the drivers and tools needed to find and mount the real root filesystem: the storage controller driver, the volume manager, the filesystem module and anything needed to unlock an encrypted or network-backed volume. The generated image is by default host-specific, which keeps it small but valid only for the configuration that generated it [21]. Configuration lives in `/etc/dracut.conf.d/`, and an administrator who adds a driver or unlocking mechanism there must regenerate the images of the kernels already installed.

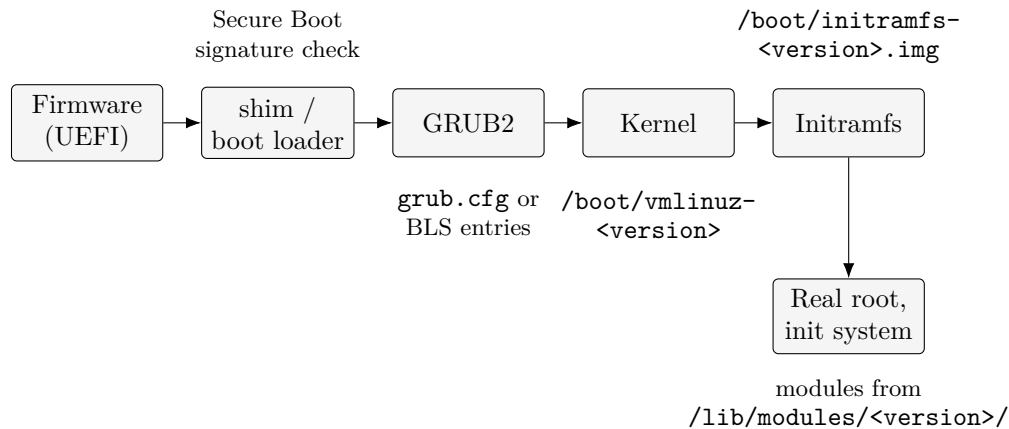


Figure 6.1: The boot chain and the artefacts a kernel upgrade writes. A kernel package adds an image, an initramfs and a boot entry; a failure in any one of them is invisible to the package transaction and visible only at the next boot.

Boot entries are managed in one of two styles. The traditional style generates a single configuration file, `/boot/grub2/grub.cfg` or its Debian equivalent, from `/etc/default/grub` and `/etc/grub.d/` [23]. The Boot Loader Specification replaces that monolith with one small drop-in file per entry, conventionally under `/boot/loader/entries/`, each naming a kernel, an initramfs and a command line [51]. Installing or removing a kernel then adds or deletes a file rather than rewriting a generated script, removing the class of failure in which regeneration silently loses a locally added entry. The corresponding trap is that an edit to `/etc/default/grub` may apply only to entries created afterwards.

Secure Boot adds a signature requirement to the chain. The firmware verifies a first-stage loader, which verifies the boot loader, which verifies the kernel, and the kernel refuses to load unsigned modules while the feature is active. A distribution kernel arrives signed by the vendor; an out-of-tree module does not, and must be signed with a key enrolled in the firmware’s machine-owner key list before it will load. This interacts badly with automatic module rebuilds, since a rebuild produces a new unsigned binary and enrolling a key is an interactive operation at the console. A kernel upgrade on a Secure Boot host with third-party modules is therefore not purely remote unless the signing step was automated in advance.

6.3 Kernel ABI and Out-of-Tree Modules

The kernel deliberately provides no stable internal API or ABI. Interfaces between subsystems change whenever that improves the kernel, on the reasoning that every in-tree user can be updated in the same commit and that freezing internal interfaces for out-of-tree code would impose a permanent cost on the kernel’s own development [28]. The consequence is direct: a module compiled against one kernel version has no guarantee of loading into another, and the failure is a refusal to load rather than a graceful degradation.

Enterprise distributions soften this within a release. They designate a subset of exported symbols as a kernel ABI whitelist, preserve those symbols’ signatures across the release’s kernel updates, and expect partner drivers to stay within that subset. A module that respects the whitelist continues to load across minor updates; one that reaches outside it does not, and the distinction is invisible until the module meets a newer kernel. `libabigail` makes the comparison explicit, reporting the differences between the ABI surfaces of two binaries so that a change can be classified as compatible without waiting for a runtime failure [24]. Enterprise kernels also

link modules from a previous kernel into a newly installed one when their symbol requirements are still satisfied, so a compatible module survives without a rebuild.

Where a module cannot be supplied as a pre-built binary for every kernel, the usual answer is to rebuild it on the host. The dynamic kernel module support framework keeps module sources under `/usr/src/` and rebuilds them against each newly installed kernel from the kernel package’s post-install hooks. When it fails it produces the characteristic failure of kernel upgrades: the transaction succeeds, the build fails in a log nobody reads, and the module is absent after the reboot. If that module is a storage driver the system does not boot; if it is a security agent, the system boots into a state that looks healthy and is not compliant. The mitigation is to treat the build as part of the upgrade: confirm the matching kernel headers are installed first, and verify the module after the transaction and again after the reboot.

6.4 Live Patching

Live patching addresses the narrow question of whether a specific kernel fix can be applied without rebooting. The mechanism rests on the function tracing infrastructure already in the kernel: compiled functions carry a patchable entry point, and the live patching subsystem uses it to redirect calls to a replacement supplied by a patch module, so that once the patch is enabled every subsequent call to the old function lands in the new one [30]. Because the patch is a module rather than a modification of the on-disk image, a live-patched system reverts to its unpatched kernel at the next boot unless the patch is reapplied.

The difficult part is not the redirection but the transition. When a patch is enabled, tasks may be executing inside the functions being replaced or have them on their call stacks, and a patch that changes the meaning of data shared between an old and a new function would be inconsistent if both ran at once. The kernel therefore applies a consistency model: it migrates tasks to the patched state individually, checking that no task is inside a patched function before switching it, and leaves the patch in a transitional state until every task has been converted [30]. Long-blocked tasks can stall that transition indefinitely. The design descends from Ksplice, which demonstrated that many security fixes could be expressed as function-level replacements derived largely automatically from the source change [6], and from kpatch and kGraft, reconciled into the in-tree subsystem [27].

Table 6.1: What live patching can and cannot address.

Amenable to a live patch	Not amenable
A self-contained fix inside one or a few functions, such as a corrected bounds check or a missing permission test	A change to the layout or meaning of a persistent data structure, since existing instances would have to be converted
A fix whose effect is entirely captured by replacing the function body	A change in initialisation code that has already run, including code discarded after boot
A fix to code reachable through an ordinary function call	A change that must alter how the kernel was built or configured, or its command-line parameters
A fix that leaves the kernel’s interfaces to modules unchanged	A version upgrade, which by definition replaces the whole image

These constraints, summarised in Table 6.1, determine what vendors offer. Live patching programmes are scoped: they cover selected kernels rather than every kernel a distribution ships, they carry high-severity fixes rather than the full stream of errata, and each patch stream

is maintained for a bounded period after which the host must move to a newer kernel [30, 27]. A system kept current by live patching alone therefore accumulates every fix that could not be expressed as a function replacement, and those are the deeper structural ones.

Live patching thus defers the reboot rather than removing it. That is a substantial gain, decoupling the urgency of a security fix from the availability of a maintenance window, the coupling that causes organisations to leave known-vulnerable kernels running. It does not change the fact that the host runs an older kernel with a patch module layered over it, that the layered state is lost at the next boot, and that a full kernel upgrade remains the mechanism by which the system is brought up to date.

6.5 Reboot Orchestration

Two different conditions are commonly conflated under the phrase “needs a reboot”. The first is that a process is running with a library replaced on disk: the old file persists as an unlinked inode held open by the process, so the running process keeps the vulnerable code while a freshly started one gets the fixed version; that is repaired by restarting the affected services. The second is that the kernel on disk is newer than the kernel running, which nothing short of a reboot resolves. Distinguishing them separates a brief service restart from a full outage of the host.

The tooling reflects the distinction. On RPM systems `needs-restarting` reports the processes whose mapped files have been replaced, and a separate option reports whether a core component such as the kernel requires a full reboot [20]. Debian systems record the same fact differently: package hooks create a marker file under `/var/run/` listing the packages responsible, while companion tools enumerate the processes still using deleted files. In both families the useful discipline is to query the condition explicitly after every transaction rather than reboot on a fixed schedule, which both reboots hosts that did not need it and leaves hosts that did.

At fleet scale the reboot itself becomes the operation requiring design. A stateless service behind a load balancer needs only that the host be removed from rotation, drained, rebooted, health-checked and returned. A clustered stateful service adds a quorum constraint: rebooting members faster than they can rejoin costs the cluster its availability, so the reboot rate must be bounded by the rejoin time. The general pattern is that of any risky change: proceed in waves, begin with a small canary population, and make continuation conditional on an explicit health signal rather than on the absence of complaints [7, 25]. What is distinctive about kernel reboots is that the health check must verify the intended kernel is running and every expected module loaded, because a host that boots on the *old* kernel, or without a third-party module, will otherwise pass a naive check.

6.6 Boot Failure Recovery

The recovery plan for a failed kernel upgrade is the previous kernel, which is why the retention policy of section 6.1 is a safety control rather than housekeeping. Selecting it is a boot-loader operation: the menu lists the installed kernels, and GRUB additionally supports a one-shot selection that applies to the next boot only and then reverts [23]. That is the correct tool for testing a kernel remotely, because an unsuccessful attempt returns the host to the entry that was working. A rescue image, where the distribution installs one, pairs a kernel with a deliberately generic initramfs, the right target when a host-specific initramfs lacks the driver for the root device.

When no entry boots to a usable system, the next layer is the boot environment itself. Parameters appended to the kernel command line from the boot menu interrupt the sequence: the `initramfs` can be told to stop and present a shell before handing over to the real root, and the `init` system to start a minimal rescue or emergency target. These give access to a partially assembled system, enough to inspect the storage stack, regenerate an `initramfs` or correct a boot entry [21].

Every one of these techniques requires a console, since each involves editing a boot entry, reading messages that appear before any network interface is configured, or typing at a shell that exists before the SSH daemon does. Remote kernel work therefore has a precondition as much organisational as technical: working out-of-band access, through a management controller, a virtual machine console or a serial console on the kernel command line, verified before the change rather than during it. An administrator without it is not performing a kernel upgrade but wagering that it will succeed.

6.7 Summary

The kernel is handled as a special case throughout the software management stack. It is installed in parallel rather than replaced, so the previous version remains bootable; its installation writes an `initramfs` and a boot entry and, under Secure Boot, depends on signatures third-party modules do not automatically carry; and its internal interfaces are deliberately unstable, which is why out-of-tree modules dominate kernel-upgrade failures. The characteristic failure is a successful transaction followed by an absent driver, detected only after a reboot.

Live patching decouples the delivery of a high-severity fix from the availability of a maintenance window, but within real limits: it replaces functions, not data structures or initialisation code, it is offered for selected kernels and for a bounded period, and its effect disappears at the next boot. Reboots should be driven by an explicit query of whether one is required and executed in waves with drain, quorum and health checks confirming that the intended kernel and its modules are actually running. Underlying all of it is out-of-band console access, without which the previous-kernel entry, the rescue image and the emergency shell are unreachable. The broader framework of testing, staging and rollback into which these practices fit is the subject of chapter 7.

7 Risk Management, Testing, and Rollback

Earlier chapters described what an upgrade does to a system. This chapter changes the question from *how does it work* to *how is it made survivable*, treating the upgrade as an engineering change that must be proposed, reviewed, rehearsed, observed and, if necessary, undone. It covers the honest framing of upgrade risk, the staging of a change through environments and canaries, the verification available around the transaction, and what rollback restores — including the boundary beyond which nothing helps. A failure taxonomy and maintenance windows close the chapter; fleet-scale automation is deferred to chapter 8.

7.1 Framing upgrade risk

Risk is conventionally expressed as the product of the probability that an adverse event occurs and the magnitude of its impact. Applied to upgrades it is useful less for producing a number than for forcing both terms to be named: an administrator who declines an update has not avoided risk but exchanged one profile for another. NIST Special Publication 800-40 Revision 4 frames patching as preventive maintenance and directs the planning effort at making routine remediation ordinary and low-friction rather than at deciding each patch on its merits [47]; the corollary is that an organisation treating every upgrade as an exceptional event will, in aggregate, apply fewer of them.

The two risks have different shapes, and this asymmetry is the central point of the section. The risk of remaining unpatched is slow and diffuse: it accumulates quietly, is hard to attribute after the fact, and summons nobody to a bridge call, yet its impact is open-ended, because an exploited vulnerability is not bounded by the service that carried it. The risk of the change window is sharp and concentrated: confined to a known interval, owned by a named person, and immediately visible if it goes wrong. Because the second is legible and the first is not, judgement systematically overweights the change window, and risk management here consists largely of correcting that bias.

Correcting it is not the same as ignoring the change window. The concentrated risk is tractable: it can be rehearsed in staging, limited by exposing a subset of hosts first, and bounded in time and in consequence by a defined window and a tested rollback path. The diffuse risk admits none of these controls and can be reduced only by applying the change. Configuration management supplies the connective tissue, since a controlled change presupposes a known prior state; NIST Special Publication 800-128 treats security-focused configuration management as establishing a baseline, controlling deviations through a change process, and monitoring for drift [26]. Without a baseline, neither the impact of an upgrade nor the success of a rollback can be assessed.

7.2 Change control in practice

Change control is often caricatured as bureaucracy, and when implemented badly it deserves the caricature; its defensible core is the small pipeline of Figure 7.1: a change request recording what will be altered on which hosts and why, a review by someone other than the author checking the blast radius, the timing and the dependencies on other systems, an approval by a person with authority to accept the consequences, a back-out plan naming how the system returns to its previous state and who invokes it, and communication to the people who will notice the outage.

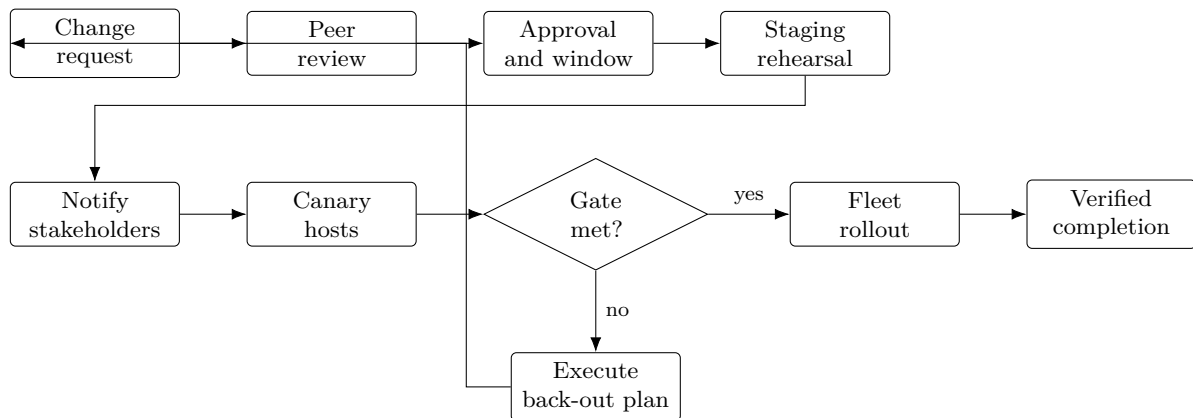


Figure 7.1: The change pipeline for a system upgrade, from request to verified completion. A failed gate returns the system to its previous state and returns the change to the request stage rather than terminating it.

The most commonly neglected element is the back-out plan, and the neglect takes a specific form: the plan is written but never executed. A plan that reads “restore the virtual machine snapshot” has not established that a snapshot exists, that it is recent, that it restores within the window, or that the service starts afterwards; one that reads “run `dnf history undo`” has not established that the transaction is reversible, which section 7.6 argues is frequently false. The staging rehearsal should therefore include the rollback and record its duration, since that duration determines how late in the window the decision to abort can still be made. Announcing the window is cheap and disproportionately effective: stakeholders who know a change is coming volunteer dependencies the author did not know about.

7.3 The environment ladder

The environment ladder discovers the concentrated risk somewhere cheap. Development allows the change to be constructed and its obvious errors removed; staging is where it is rehearsed against something resembling production; a pre-production canary exposes it to genuine traffic on a few hosts; production is the last rung, by which point the change should hold no surprises the earlier rungs could have caught.

What makes staging representative is worth stating precisely, because “mirror” is usually taken to mean only that the operating system version matches. A representative staging host shares the distribution and release, the enabled repositories including third-party ones, the kernel variant, the installed package set including locally built and vendor-supplied packages, the storage layout and filesystem types, and the service configuration. It also shares the *shape* of the data: not the volume, but the schema version, the index structure and legacy rows predating an earlier migration, because a schema migration triggered by a package upgrade behaves differently against an empty test database than against one holding years of production artefacts.

Staging environments fail to be representative in recurring ways. They drift, because they are upgraded ad hoc during testing and never returned to the baseline. They are smaller, so resource exhaustion during the transaction — particularly disk space in `/var` and `/boot` — does not reproduce. They lack the third-party monitoring, backup and endpoint-security agents production carries, precisely the packages most likely to conflict with a distribution upgrade. They are not under load, so races in service restarts do not appear. Each is a known limitation rather than a defect provided it is recorded; the danger is the unrecorded divergence

that silently invalidates the rehearsal.

7.4 Canary and progressive rollout for fleets

Where staging approximates production, a canary is production: the change is applied to a small subset of hosts, observed against defined signals for a defined period, and only then carried further. It converts “will this change break production” into “has this change broken a small, recoverable part of production”, a question answerable by observation instead of prediction. The site reliability engineering literature accordingly treats progressive rollout as a standard property of a release process, on the grounds that no pre-production environment reproduces every property of the production system [7, 25].

Choosing canary hosts trades representativeness against exposure: a canary that serves no traffic proves nothing, and one that serves the most important customer is not a canary. Suitable hosts are ordinary members of their role, receive a genuine share of traffic, are instrumented at least as well as the rest of the fleet, and can be drained without user-visible effect. Where the fleet is heterogeneous a single canary is insufficient, since each combination of hardware generation, role and configuration profile carries its own failure modes.

The observation window must be long enough to include slow failure modes, since many upgrade defects appear not at restart but at the first periodic job, certificate renewal or traffic peak; it is better defined by the events it must cover than in raw time. Halting criteria should be fixed before the rollout begins and, where possible, automated — not because automation is more accurate, but because it is not subject to the sunk cost of a rollout already in progress. Useful criteria compare the canary against the unchanged remainder of the fleet rather than an absolute threshold, since that controls for changes in the workload itself; error rates, latency, saturation and new journal message signatures all serve. The rollout then proceeds in widening batches with the gate evaluated between them, so that a defect escaping the canary is still bounded.

7.5 Testing before and after

Verification falls into two groups: checks that the packages on disk are what the distribution intends, and checks that the system built from them behaves. The first is mechanical. The RPM database records the expected size, mode, ownership, checksum and timestamp of every installed file, and `rpm -Va` compares the filesystem against those records [44]; `debsums` verifies Debian-family files against the checksums shipped in the package metadata, which not every package ships, and `pacman -Qkk` performs the equivalent check on Arch [5]. All of these report configuration files as modified wherever a local edit exists, so their output must be read against knowledge of which files are locally managed; the signal of interest is an unexpected difference in a binary or library. Listing 7.1 shows the capture step.

Listing 7.1: Package verification and inventory capture before a transaction, on RPM-based and Debian-based systems.

```
# RPM family: record verification output and inventory before the change
rpm -Va > /var/tmp/verify-before.txt
rpm -qa -qf '%{NAME} %{EVR} %{ARCH}\n' | sort > /var/tmp/pkg-before.txt

# Debian family: the same idea with debsums and dpkg-query
debsums -c > /var/tmp/verify-before.txt 2>&1
dpkg-query -W -f='${Package} ${Version} ${Architecture}\n' \
  | sort > /var/tmp/pkg-before.txt
```

```
# After the transaction, compare the two package inventories
diff -u /var/tmp/pkgs-before.txt /var/tmp/pkgs-after.txt
```

The second group is behavioural. Service health checks confirm that the units the host is should run are active and not in a restart loop: `systemctl is-system-running` and `systemctl -failed` give the coarse answer, a per-unit check of the enabled set the precise one. Application-level smoke tests exercise the service through its real interface, such as an authenticated request returning the expected status, and are the only checks demonstrating that the system does its job rather than that its processes exist. They should be written before the upgrade, kept under version control alongside the change, and run both before and after, so that a pre-existing failure is not attributed to the change.

Two further checks are routinely omitted. The first is a comparison of configuration files before and after, since an upgrade may install a new default file, rename the administrator’s version or leave a merge artefact, the mechanisms differing by family [44, 14]; diffing `/etc` against a copy taken beforehand surfaces a class of failure that would otherwise appear days later. The second is verifying that services actually restarted, because an upgrade replaces the binary on disk while a running process keeps executing the old code from its already-mapped image, so a host may pass every functional test while still running the vulnerable version. The `needs-restarting` utility and its `-r` option report affected processes and reboot requirements on RPM systems [20], `needrestart` plays the analogous role on Debian, and comparing unit start timestamps against the transaction timestamp confirms the result independently.

7.6 Rollback mechanisms ranked by strength

Rollback mechanisms are not interchangeable and can be ordered by how much of the system they restore. Table 7.1 gives that ordering and, more importantly, what each mechanism leaves untouched.

Table 7.1: Rollback mechanisms ordered by strength, with what each restores and what it leaves unchanged.

Mechanism	Restores	Does not restore
Transaction undo	Package versions installed or removed by one transaction	Configuration altered by scriptlets, migrated data, files deleted with a removed package
Filesystem snapshot (Btrfs, LVM)	The subvolume or logical volume at snapshot time, including <code>/etc</code>	Data on volumes outside the snapshot; anything committed after it
Virtual machine or volume snapshot	Full guest disk state, including the boot loader and kernel	External state: databases on other hosts, queues, third-party side effects
Image rollback (rpm-ostree, bootc)	The whole operating system tree, atomically, to a prior deployment	<code>/var</code> and other externalised state, by design
Redeploy from configuration	A host equivalent to the intended baseline	Anything not expressed in the configuration; data must be restored separately

The weakest mechanism is transaction undo, which `dnf history undo` and the rollback facilities of `zypper` implement on the package layer alone by computing an inverse transaction [40, 19]; it fails outright if the older packages are no longer in the enabled repositories. Filesystem

snapshots are stronger, and openSUSE integrates them with the package manager so that pre- and post-transaction snapshots are created automatically and can be booted [36, 49]. Virtual machine and storage-volume snapshots are stronger again because they include the boot loader and kernel, which makes them the mechanism of choice when the change touches the boot path, and image-based systems, discussed in chapter 8, make rollback a property of the design rather than a procedure. Redeploying from configuration restores a host rather than a state.

The limit that matters most is the data problem. Every mechanism above restores *code and configuration*; none restores *state that has already been transformed*. When a stateful service — a relational database, a message broker — starts under a new version, it frequently migrates its on-disk format irreversibly, rewriting system catalogues, upgrading index structures or converting a storage format the previous version cannot read. Reverting the packages then produces the worst available state: an older binary pointed at a newer data directory, which typically refuses to start and may, if it does start, damage the data further. The rollback has succeeded at the package layer and failed at the system layer.

For such hosts the rollback plan must therefore be expressed in terms of the data rather than the packages: a verified backup or a snapshot of the data directory taken while the service was stopped, with a documented restore procedure, is the only real back-out. The point of no return is not the package transaction but the first start of the new service version, and the plan should name that moment explicitly. A dump-and-reload upgrade path is preferable to an in-place format migration precisely because it leaves the original data directory intact, and where a service cannot be rolled back at all that fact belongs in the change request rather than in the incident report.

7.7 Backups as the last resort

When every rollback mechanism has failed or does not apply, what remains is the backup. For upgrade purposes the set captured is narrower than a disaster-recovery backup but must be complete within its scope: the data directories of every stateful service in a consistent state, produced by the service's own dump facility or by a snapshot taken while it was stopped; the whole of `/etc`; the package inventory and enabled repository configuration; local or vendor-supplied packages unavailable from a public repository; and the credentials and keys required to bring the service back into its environment. A backup that restores the data but not the configuration which makes it reachable has solved half the problem.

What distinguishes a backup from an archive is that the restore has been rehearsed. A rehearsal on a separate host, ending with the service running and a smoke test passing, is the only evidence that the backup is usable, and it produces the figure planning requires: how long a restore takes at realistic volumes. The recovery point objective, the age of the most recent usable backup, and the recovery time objective, the tolerable unavailability, then constrain scheduling: the window must contain the upgrade, the verification and a full restore, and the backup must be recent enough at its start.

7.8 A failure taxonomy with mitigations

The failures that occur during upgrades are drawn from a small catalogue, and each has a mitigation known in advance. Table 7.2 lists the recurring modes with the control that prevents or contains each.

Three deserve emphasis because they turn a recoverable change into an outage. An

Table 7.2: Recurring upgrade failure modes with their principal mitigations.

Failure mode	Manifestation	Principal mitigation
Dependency conflict	Solver refuses the transaction or proposes removing needed packages	Resolve in staging; audit third-party repositories; use a snapshotted repository so the solver sees a fixed content set
Disk exhaustion mid-transaction	Transaction aborts while writing; <code>/var</code> or <code>/boot</code> full	Check free space against download and install size beforehand; prune caches and old kernels; alarm on thresholds
Interrupted transaction, power loss	Partially installed package set; locked package database	Run under a detached session; prefer offline or transactional update modes; repair with the database recovery tooling
Configuration file divergence	New defaults not applied, or local edits displaced	Diff <code>/etc</code> before and after; search for <code>.rpmnew</code> , <code>.rpmsave</code> and <code>.dpkg-dist</code> ; manage <code>/etc</code> under configuration control
Service fails to start	A unit enters a failed state after the transaction	Post-upgrade unit health check and smoke test; restore from snapshot while the pre-change state is still available
Kernel does not boot	Boot stops in the <code>initramfs</code> , or the boot entry is missing	Retain the previous kernel and its boot entry; verify <code>initramfs</code> generation; arrange console access before the window
Network configuration lost	Remote host unreachable; no interface configured after the change	Out-of-band console; scheduled automatic revert unless the session is reconfirmed; never change networking without a second path
Subscription or licence expiry	Repository refuses authentication; the transaction cannot start	Verify entitlement and repository reachability as a pre-flight check; track expiry dates in the inventory

interrupted transaction is most often caused not by power loss but by a dropped administrative session, prevented trivially by a detached terminal multiplexer and more thoroughly by the offline update mode, in which the transaction runs from a minimal environment at boot [22]. A kernel that does not boot is recoverable in seconds if the previous kernel is still installed with a boot loader entry, and is an on-site visit otherwise. Losing network configuration on a remote host has the worst ratio of probability to consequence, because the operator loses the channel they would use to repair it.

7.9 Maintenance windows and scheduling

A maintenance window converts an unbounded risk into a scheduled one. Its length should be derived rather than assumed: the rehearsed durations of the transaction, the verification and the rollback, plus a margin, with the abort decision placed early enough that the rollback still fits inside the window. Windows placed at night reduce user impact but degrade the alertness of the people performing the change and the support available if it goes wrong.

The recurring scheduling question is whether errata should be batched or streamed. Batching

collects updates into periodic windows, concentrating testing effort and making the fleet's state easy to describe, at the cost of leaving known vulnerabilities unpatched for the interval; streaming applies updates continuously, minimising exposure time but making each change individually less rehearsed. The usual resolution is a hybrid: a batched cadence for routine updates with a separate expedited path for the few that meet a severity threshold. That path must itself be rehearsed, since an emergency procedure used once a year is an untested procedure.

Freeze periods — intervals during which no change is permitted, typically around peak trading or financial close — are a legitimate control on the concentrated risk of the change window and a direct increase in the diffuse risk of remaining unpatched. A freeze is defensible when bounded, published in advance and paired with a working exception process for critical security updates, and indefensible when indefinite or when that exception exists only on paper. The framing of section 7.1 applies: a freeze moves risk from a visible category to an invisible one rather than removing it, and the exposure accumulated during it should be recorded so that the trade-off is deliberate [47].

7.10 Summary

Upgrade risk is a choice between two differently shaped risks rather than between risk and safety: the diffuse, open-ended risk of remaining unpatched against the sharp, bounded risk of the change window. The controls described here — rehearsed back-out plans, a ladder of environments whose representativeness is stated rather than assumed, canary hosts with automated halting criteria, verification around the transaction, and a rollback mechanism chosen for what it actually restores — exist to make the second risk small enough that it stops dominating the decision.

The limit on all of this is state. The mechanisms of Table 7.1 restore code and configuration with increasing strength, but none reverses an on-disk format migration a stateful service has already performed, which makes a rehearsed backup of the data directory the only genuine back-out for such hosts and the first start of the new service version the true point of no return. chapter 8 takes the same principles to a fleet, where the gates become automated and the repository itself becomes the primary instrument of control.

8 Automation and Fleet-Scale Upgrading

chapter 7 treated the upgrade as a controlled change to a system, assuming a person is present to run the commands and read the output. This chapter removes that assumption and examines what changes when the same upgrade must reach thousands of hosts, what may safely be delegated to an unattended mechanism, and how the repository becomes the principal instrument of control. It then turns to fleets with no route to the public network, to the image-based and transactional models that replace the package transaction with an atomic switch between whole system states, and finally to the indicators by which an upgrade programme can be judged. The accompanying measurements are reported in chapter 9.

8.1 From one host to many

Scaling an upgrade from one host to many is not merely repetition, because three properties change qualitatively. The first is heterogeneity. A fleet described as “the web tier” is rarely uniform: hosts were built at different times from different base images, carry different hardware generations, have accumulated different third-party repositories and may sit at different minor releases. A procedure correct for the modal host is not necessarily correct for the tail, and the tail is where the failures occur.

The second is drift. Configuration management establishes a baseline, but every manual intervention, every locally installed package and every hand-edited file in `/etc` moves a host away from it, and the divergence is invisible until an upgrade touches the file in question [26]. Drift is what makes a fleet-wide transaction unpredictable: the solver runs against a different installed set on each host, so the transaction rehearsed in staging is not the transaction that will execute.

The third, and the most consequential for automation design, is that partial failure becomes the normal case. A fleet-wide run typically reports a large majority of successes together with a handful of hosts that were unreachable, out of disk space, holding a stale package lock, or already failed before the run began. Automation must be designed around this outcome: it needs an inventory enumerating the intended targets, a reporting channel that distinguishes “succeeded” from “not attempted” from “failed”, and a means of retrying the remainder without repeating work already done. A tool that only prints progress to a terminal does not satisfy these requirements, because the result must remain queryable after the terminal has closed. Both the desired and the observed state must therefore be recorded centrally; the host is a poor place to store the record of what was supposed to happen to it.

8.2 Unattended patching

The simplest form of automation applies updates on each host without an operator. On RPM-based systems this is the role of `dnf-automatic`, driven by `/etc/dnf/automatic.conf` and triggered by a `systemd` timer. Two settings together define three useful modes: report only, download but do not apply, or download and apply. The `upgrade_type` setting narrows the candidate set to security errata, and the emitter configuration determines whether results are sent by electronic mail, to standard output, or to the system log [18]. Separating download from application removes network transfer time from the maintenance window and makes the later transaction fast and predictable.

On Debian-based systems the equivalent is `unattended-upgrades`, configured through

`/etc/apt/apt.conf.d/50unattended-upgrades` and enabled through the periodic settings in `/etc/apt/apt.conf.d/20auto-upgrades`, with the `apt-daily` and `apt-daily-upgrade` timers performing the metadata refresh and the upgrade respectively. Its central concept is the list of allowed origins: candidates are matched against origin and archive patterns, so a policy can admit the security archive while excluding the general updates archive. A blacklist excludes individual packages by name, and separate settings control whether and when the host may reboot itself [16]. Listing 8.1 shows the essential fragments of both configurations.

Listing 8.1: Essential settings for unattended patching on RPM-based and Debian-based systems.

```
# /etc/dnf/automatic.conf - security errata, downloaded and applied
[commands]
upgrade_type = security
download_updates = yes
apply_updates = yes
reboot = never
[emitters]
emit_via = stdio,motd
# enable with: systemctl enable --now dnf-automatic.timer

# /etc/apt/apt.conf.d/50unattended-upgrades
Unattended-Upgrade::Origins-Pattern {
    "origin=Debian,codename=${distro_codename}-security";
};
Unattended-Upgrade::Package-Blacklist { "postgresql-"; };
Unattended-Upgrade::Automatic-Reboot "false";
Unattended-Upgrade::Automatic-Reboot-Time "03:00";
```

SUSE systems reach the same end through a timer invoking `zypper patch`, or, on transactional variants, through `transactional-update` on a timer [49]. Rolling distributions are the exception: because a partial upgrade is unsupported and manual intervention is occasionally required between releases of core packages, Arch documentation advises against unattended upgrading altogether [4].

The interesting question is not how to configure these tools but which classes of update may apply without a human. The defensible general policy is that security errata for leaf packages — those on which no locally significant service depends for its on-disk state — may apply unattended, because the diffuse risk of delay outweighs the concentrated risk of an unobserved change. Updates that restart a stateful service, cross a minor release boundary, touch the kernel or the boot path, or come from a third-party repository should not. The reboot setting deserves its own policy: applying updates while deferring the reboot to a scheduled window is usually the right compromise, provided the deferred reboot is tracked rather than postponed indefinitely.

8.3 Configuration management and orchestration

Configuration management systems express the desired state of a host declaratively and converge the host towards it. For upgrades the relevant primitive is idempotent package state: a declaration that a package is present, present at a specified version, or at the latest available version, which produces a transaction only when the host does not already satisfy it. The distinction between “present” and “latest” is a policy decision disguised as a syntax choice. Declaring “latest” delegates the upgrade decision to whatever the enabled repositories happen to offer when the run occurs, making the outcome a function of timing rather than of intent; declaring an explicit version keeps the decision in the inventory, where it is reviewable, diffable

and identical for every host in the group. This is the argument of section 7.2 applied to code: the change should be visible in a repository before it is visible on a host.

Orchestration is the layer above, concerned with the order and pace of execution across the fleet. Serial or batched execution applies the change to a bounded number of hosts at a time, implementing the progressive rollout of section 7.4. A health gate between batches queries the hosts just changed — service state, a smoke test, an error rate from the monitoring system — and stops the run if the result is unsatisfactory, so that a defect is contained at the batch in which it appeared. A maximum failure threshold halts a run in which hosts are failing at a rate suggesting a systematic cause rather than local accidents. Restarts are best expressed as handlers triggered by the change rather than as unconditional steps, so that a run which changes nothing restarts nothing; combined with the restart detection of section 7.5, this restarts services exactly when their code has changed.

8.4 Repository control as the real control plane

Automation that decides *when* to upgrade is far less powerful than control over *what* is available to upgrade to. If every host queries an upstream repository whose contents change continuously, two hosts patched an hour apart converge on different package sets, and no orchestration will make the fleet uniform. The remedy is to place a repository under local control. Mirroring copies the upstream content locally; the important step is snapshotting, in which the mirror is frozen at a point in time and published under a stable name, so that every host resolving against that snapshot sees an identical content set. A local mirror is also a trust boundary, since the integrity of the metadata a client consumes determines which packages it will install, and research on package manager security has shown that a compromised or stale mirror can be used to withhold updates or replay old metadata [11, 45]; signature verification and freshness checks are therefore part of the mirror’s configuration rather than an optional extra.

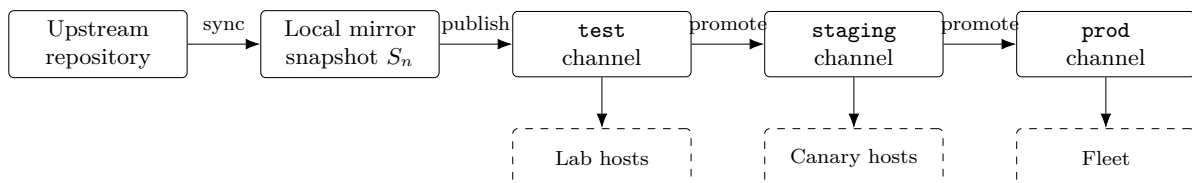


Figure 8.1: Staged promotion of a repository snapshot. Hosts are pinned to a channel name; promotion moves a snapshot between channels, so no host configuration changes when the content set advances.

Figure 8.1 shows the resulting arrangement. A snapshot is published first to a test channel, exercised there, then promoted to a staging channel consumed by canary hosts, and finally to the production channel. Because hosts are configured with a channel name rather than a snapshot identifier, promotion changes what the fleet sees without touching any host’s configuration, and demotion rolls the fleet’s content set back just as cheaply. The pattern is expressed with `reposync` and `createrepo_c` on RPM systems and with a mirroring tool that supports named snapshots on Debian systems.

Within a channel, finer control is available. The `python3-dnf-plugin-versionlock` plugin pins selected packages to a version so that a general upgrade does not move them, and the `excludepkgs` option in `/etc/dnf/dnf.conf` or in an individual repository file removes packages from consideration entirely [40, 19]. Debian systems achieve the equivalent through pin priorities in `/etc/apt/preferences.d` and through `apt-mark hold`, and SUSE through

`zypper addlock`. These suit a package that must remain at a vendor-certified version, but each is a permanent exception that will eventually block a security update, so locks should carry a recorded justification and a review date.

8.5 Air-gapped and offline fleets

Fleets with no route to a public repository invert the problem: instead of controlling what the hosts may reach, the administrator must first construct what they will reach. The unit of transport is a dependency closure — the packages required to satisfy the intended transaction, everything they depend upon, and, critically, the repository metadata that describes them. A directory of package files without generated metadata is not a repository, and a closure whose metadata came from a different package set will resolve to packages that are not present.

The capture is performed with the same solver the target will use. On RPM systems `dnf download -resolve` or a `reposync` of the relevant repositories produces the files, and `createrepo_c` generates the metadata; on Debian systems the equivalent is a download-only `apt-get` run followed by generation of package indices. Transport is usually removable media, and verification on arrival is not optional: the repository signing keys must be present on the target and signatures checked rather than disabled for convenience, since the transport medium is where the supply chain is least observable.

The specific failure mode worth naming is that a closure captured on a connected host is silently incomplete. The solver computes what is needed *in addition to what is already installed*, so any dependency present on the capture host is omitted. The result passes every test on the capture host and fails on the target, in the worst place — inside the maintenance window, on an air-gapped system, with no means of fetching the missing file. The discipline that prevents it is to build the closure against an empty root: an installation root created solely for the purpose, or a minimal container image matching the target release, so that the solver has no pre-existing packages to assume. The closure should then be verified in a clean environment at the target's release, with no network and resolving only against the captured repository, before the window rather than during it. An offline upgrade performed under these constraints is among the procedures measured in chapter 9.

8.6 Image-based and transactional systems

A different response to fleet-scale upgrading is to stop performing package transactions on running systems. In the image-based model the unit of change is an entire operating system tree, built once and deployed unchanged. With `rpm-ostree` the host holds a few complete deployments; an upgrade writes a new one alongside the current and switches the default at the next boot, and `rpm-ostree rollback` restores the previous deployment. Because the switch is a pointer change rather than a sequence of file replacements, there is no state in which the system is half-upgraded [12]. The `bootc` project expresses the same idea using container images as the transport and build format, so the host is updated by pulling a new image reference [8]. SUSE's `transactional-update` applies the transaction inside a new read-only Btrfs snapshot rather than to the running filesystem, and that snapshot becomes the default root at the next boot, with `rollback` selecting the previous one [49].

Table 8.1 compares the three models. Atomicity removes the partially upgraded system from the space of possible outcomes, eliminating a whole class of failures from Table 7.2. Identical fleet state follows from deploying a single artefact rather than converging each host

Table 8.1: Package-based, image-based and transactional models for fleet upgrading.

Property	Package-based	Image-based	Transactional
Unit of change	Individual packages	A complete system tree or container image	A package transaction inside a new snapshot
Atomicity	Per transaction, not per fleet	Whole tree; switch at boot	Whole snapshot; switch at boot
Rollback	Inverse transaction or snapshot	Select the previous deployment	Select the previous snapshot
Fleet uniformity	Depends on repository control and drift	Identical by construction	Identical if built from the same content set
Local modification	Direct; the normal way of working	Layering or rebuilding the image	Applied inside a new snapshot
State handling	In place; state and code share the root	State must be externalised	Persistent subvolumes outside the snapshot
Prerequisite	A controlled repository	Build and registry infrastructure	Btrfs layout and a reboot discipline

independently, so drift ceases to divide the rehearsal from the run. Rollback becomes trivial and, more importantly, uniform: the same operation reverses any upgrade, so the back-out plan is the same for every change and is exercised often enough to be trustworthy.

The costs are equally concrete. State must be externalised, because the tree is replaced wholesale and only designated directories persist across the switch; a service that keeps its data inside the image will lose it, and one that keeps data on a persistent volume is subject to the same irreversible-migration limit described in section 7.6, which atomic rollback does not repeal. Layering local changes becomes awkward: a host carrying local layers is no longer identical to its peers, which forfeits the main advantage. Finally, build infrastructure becomes mandatory: every change, however small, must pass through a build and a publication step, at once a genuine improvement in discipline and a genuine increase in the minimum viable toolchain.

8.7 Measuring an upgrade programme

An upgrade programme that is not measured cannot be improved, and the natural instinct — to count the updates applied — measures activity rather than outcome. Four indicators are more informative. Patch latency is the elapsed time between the publication of an update and its application on a host, reported as a distribution across the fleet rather than an average, because the tail is the exposure that matters. Coverage is the proportion of the intended inventory the mechanism actually reaches, and it is meaningful only against an independent inventory: hosts no tool knows about are precisely the ones never patched. Rollback rate is the proportion of changes that had to be reversed, and failed-host rate the proportion of hosts on which a run did not complete successfully.

The last two are leading indicators, predicting the future of the programme rather than describing its past. A rising failed-host rate signals accumulating drift, expiring entitlements or an inventory that no longer matches reality, and it appears before patch latency degrades. A very high rollback rate indicates insufficient testing, but one that is zero over a long period is equally suspicious, since it usually means the rollback path is never exercised and would not work if invoked. These indicators serve the purpose that NIST Special Publication 800-40 Revision 4 assigns to metrics in a patch management programme: to show whether routine maintenance is functioning, so that attention can be directed at the exceptions [47]. Reporting

them requires the central record described in section 8.1, which is why an inventory is a prerequisite for automation rather than a by-product of it.

8.8 Summary

Upgrading a fleet differs from upgrading a host in kind rather than degree, because heterogeneity, drift and routine partial failure make the outcome of any single run uncertain. Unattended mechanisms such as `dnf-automatic` and `unattended-upgrades` handle the routine case well when policy restricts their scope, and configuration management supplies idempotent package state, batched execution and health gates. The more powerful lever is the repository: mirroring and snapshotting content, then promoting a snapshot from a test channel through staging to production, gives the fleet a single known content set and makes both rollout and rollback a property of the channel rather than of each host. Air-gapped fleets apply the same idea to a transported dependency closure, whose principal hazard is silent incompleteness when captured against an already-populated host.

Image-based and transactional systems — `rpm-ostree`, `bootc` and `transactional-update` — take the argument to its conclusion by replacing the in-place transaction with an atomic switch between whole system states, buying atomicity, uniformity and a uniform rollback at the price of externalised state, awkward local modification and mandatory build infrastructure. None of these models removes the limit established in chapter 7: an irreversible on-disk migration survives every form of system rollback. Patch latency, coverage, rollback rate and failed-host rate provide the evidence that the programme is working, and chapter 9 reports the measurements taken for this thesis.

9 Practical Evaluation

The preceding chapters describe upgrade mechanisms as their designers and maintainers document them. This chapter reports what those mechanisms actually did on a controlled system. Six experiments were carried out on virtual machines created for this thesis: a baseline characterisation, an online package upgrade, a kernel upgrade with the reboot that validates it, two rollback attempts by different mechanisms, and an upgrade performed entirely from a locally captured content set on a host with no network access. The purpose is not to benchmark distributions against one another but to obtain evidence for the research questions, and in particular for RQ3, which concerns what determines whether an upgrade can be undone.

9.1 Experimental setup

All experiments ran on virtual machines under QEMU/KVM on an x86-64 host. Each guest was created from the same unmodified AlmaLinux 9.8 generic cloud image, so every run started from an identical, reproducible state. AlmaLinux is a rebuild of Red Hat Enterprise Linux 9.8 from the same sources, which makes the observed `dnf` and `rpm` behaviour representative of the wider RHEL family discussed in Chapters 4 and 5.

Table 9.1: Experimental platform.

Property	Value
Guest operating system	AlmaLinux 9.8 (Olive Jaguar), generic cloud image
Guest kernel at baseline	5.14.0-687.36.1.el9_8.x86_64
Virtualisation	QEMU/KVM, q35 machine type, host CPU passthrough
Guest resources	8 virtual CPUs, 8 GiB memory, 40 GiB virtual disk
Guest disk format	qcow2 copy-on-write overlay on a shared read-only base
Package manager	<code>dnf</code> 4 with the distribution’s default repositories
Network (online runs)	user-mode networking with outbound access
Network (offline runs)	user-mode networking with <code>restrict=on</code> ; no route out

Two network configurations were used. Online guests reached the distribution mirrors normally. Offline guests were started with QEMU’s `restrict=on` option, which gives the guest an address but drops every packet leaving it; only an inbound forwarded port for administrative access remained. Each offline guest was verified to be isolated before any measurement was taken: name resolution timed out and `dnf` could not reach its mirrorlist. This arrangement reproduces an air-gapped host while keeping the experiment scriptable.

Timings were taken with `/usr/bin/time` inside the guest and are reported as wall-clock seconds for a single run. They indicate magnitude rather than performance: a virtual machine on a shared host, with mirror selection and caching outside the experiment’s control, is not a benchmarking environment. What matters here is qualitative — whether an operation succeeded, what it changed, and what it could restore — and those results are not sensitive to timing noise.

The integrity of a system state was captured by hashing the sorted package list, `rpm -qa | sort | sha256sum`, which yields a short fingerprint that changes if any package is added, removed or changed in version. Fingerprints are quoted by their first sixteen hexadecimal digits.

9.2 Experiment 1: baseline characterisation

The unmodified image was booted and measured before any change. The installed system comprised 388 packages occupying approximately 823 MB as reported by the RPM database, with 1.20 GB used on the root filesystem. Exactly one kernel was installed, and the system reached its start-up target in 6.09 seconds. Immediately after boot, `dnf check-update` reported 55 packages with newer versions available, and took 0.31 seconds to do so against freshly downloaded metadata.

Two observations follow. An image published five weeks before the experiment already required 55 updates, which illustrates the exposure argument of Chapter 7: a freshly deployed host is not a patched host. The minimal image also lacked `needs-restarting`, which had to be installed from `yum-utils` before the restart analysis in Section 9.4 — tooling an upgrade procedure assumes is not necessarily present on a minimal installation.

9.3 Experiment 2: an online package upgrade

The full set of pending updates was applied with `dnf -y upgrade`. The transaction upgraded 51 packages and installed 4 — the four packages of the new kernel, which are installed alongside the running one rather than replacing it, as described in Chapter 6 — for a download of 145 MB.

Table 9.2: Online package upgrade of a freshly deployed host (Experiment 2).

Measure	Before	After
Packages installed	388	392
Size reported by the RPM database	822.8 MB	965.6 MB
Root filesystem used	1201.1 MB	1393.6 MB
Kernels installed	1	2
<i>Transaction</i>		
Packages upgraded / installed		51 / 4
Download size		145 MB
Wall-clock time		85.5 s
Peak resident memory of <code>dnf</code>		133.9 MB

The operation took 85.5 seconds and the host remained reachable throughout, which is why routine patching is operationally cheap and easy to underestimate. The transaction left the system internally consistent but not yet correct: the running processes were still the old ones.

`needs-restarting -r` made that explicit, exiting with status 1 and reporting that core libraries and services had been updated since boot:

Listing 9.1: Restart analysis after the upgrade transaction (Experiment 2).

```
# needs-restarting -r
Core libraries or services have been updated since boot-up:
* dbus-broker
* kernel
* microcode_ctl
* systemd
Reboot is required to fully utilize these updates.
```

In addition, eleven running processes — including PID 1 — still mapped files belonging to the replaced packages. The distinction drawn in Chapter 6 between applying an update and

activating it is therefore not academic: at this point the host reported itself as fully upgraded to any inventory that asks `rpm`, while continuing to execute the code the upgrade was meant to remove.

9.4 Experiment 3: kernel upgrade and reboot

Before the reboot, both kernels were present and the boot loader had already been updated to prefer the new one. The Boot Loader Specification drop-in directory contained one entry per kernel, and `grubby --default-kernel` pointed at 5.14.0-687.48.1, confirming the mechanism described in Chapter 6: the old kernel remains installed and bootable, which is the cheapest rollback available for a kernel change.

Table 9.3: State around the kernel upgrade reboot (Experiment 3).

Measure	Before reboot	After reboot
Running kernel	5.14.0-687.36.1	5.14.0-687.48.1
Boot entries present	2	2
<code>needs-restarting -r</code>	reboot required (exit 1)	not required (exit 0)
Processes with replaced files	11	0
Failed systemd units	—	0
Start-up time	6.09 s	5.17 s

The reboot completed and the host became reachable again within roughly half a minute. Afterwards the new kernel was running, no process mapped replaced files, no systemd unit had failed, and `rpm -Va` reported only two modified files, both of them configuration files legitimately changed during the experiment. The upgrade was, at this point, both applied and activated.

The observation is one of asymmetry. Installing the kernel cost a few seconds within an 85-second transaction and carried almost no risk, because nothing changed for the running system; validating it required the reboot, the only operation in the experiment that could have left the host unreachable and whose outcome the transaction’s exit status cannot predict. Chapter 6 argues that live patching defers this moment rather than removing it, and the measurement supports that framing.

9.5 Experiment 4: rollback by transaction undo

With the system upgraded and rebooted, an attempt was made to reverse the upgrade with `dnf history undo`, the mechanism most often named as the package manager’s rollback facility.

Listing 9.2: Attempting to undo the upgrade transaction (Experiment 4).

```
# dnf -y history undo 2
Error:
Problem: The operation would result in removing the following protected
packages: kernel-core
```

The attempt was refused in 0.45 seconds and the system was left untouched: the package fingerprint was identical before and after, `e76f386b83c30793`. The reason is a safeguard rather than a defect. Undoing the transaction would remove the kernel packages it installed, and the system had by then booted into that kernel, so `dnf` declined to remove a protected package that was in use. The practical consequence deserves emphasis: the single transaction that a practitioner is most likely to want to reverse — a routine upgrade that included a kernel — is the one that transaction undo refuses outright once the host has rebooted.

Undo works well where its preconditions hold. Installing a package and then undoing that transaction removed it cleanly in 0.52 seconds. Its second precondition, availability, was tested by removing a package, then undoing the removal with every repository disabled:

Listing 9.3: Transaction undo depends on the old packages still being obtainable.

```
# dnf -y history undo 9 --disablerepo="*"
Error: The following problems occurred while running a transaction:
  Cannot find rpm nevra "unzip-6.0-60.el9.x86_64".
  Cannot find rpm nevra "zip-3.0-35.el9.x86_64".
```

The same undo succeeded in 1.19 seconds once the repositories were enabled again. An undo is thus not a stored copy of the previous state; it is a recipe that must be re-executed against content that is still published. On an air-gapped host, or after a distribution has retired an old build, the recipe may no longer be satisfiable — which is precisely when rollback is most likely to be needed.

A third limit was visible throughout. A file written by an application after the upgrade, `/var/lib/demo/data.txt`, survived every undo attempt unchanged. Transaction undo operates on packaged files and does not touch service state; it neither restores data nor reverses what a service has already migrated. This is the mechanism’s most important boundary, and it is invisible in its exit status.

9.6 Experiment 5: rollback by disk snapshot

The second rollback mechanism operated one layer lower. With the guest powered off, a qcow2 snapshot of the upgraded state was created, the guest was booted again, and further changes were made: an additional package was installed and a second data file, `/var/lib/demo/after-snapshot.txt`, was written. The snapshot was then restored.

Table 9.4: Disk snapshot rollback (Experiment 5).

Measure	Before rollback	After rollback
Packages installed	396	393
Package fingerprint	a8e67b2b56420e59	e76f386b83c30793
Package installed later	present	absent
Data written before snapshot	present	present
Data written after snapshot	present	absent
dnf transaction history	6 entries	3 entries
Failed systemd units	0	0
Snapshot creation time		0.01 s
Snapshot restore time		0.02 s
Overlay growth from snapshot		65.6 kB of metadata

The restored fingerprint, `e76f386b83c30793`, is exactly the fingerprint recorded immediately after the upgrade and reboot. The snapshot restored the system to a byte-identical prior state, including the package manager’s own history, in twenty milliseconds, and the operation itself cost almost nothing in space because a copy-on-write snapshot records metadata rather than data.

It also destroyed everything written after the snapshot was taken. The file created after the snapshot was gone, as were the guest-side logs of Experiment 4, which had to be recovered from the console transcript. This is the defining property of the mechanism, and it is the

mirror image of transaction undo: undo preserves data and cannot reliably restore the system, while a snapshot restores the system perfectly and cannot preserve data. Neither is a general rollback for a stateful service, and no combination of the two is either, because the question they cannot answer — what to do about state that has already been migrated to a new format — is a property of the application, not of the upgrade tooling.

9.7 Experiment 6: upgrading an air-gapped host

The final experiment reproduced the workflow that Chapter 8 describes for disconnected environments: capture the content on a connected twin, transport it, and apply it on a host that cannot reach any repository.

A connected guest, identical to the target and at the same baseline, downloaded the pending updates without installing them and generated repository metadata over them:

Listing 9.4: Capturing the content set on a connected twin (Experiment 6).

```
# dnf -y upgrade --downloadonly --destdir /root/updates
# createrepo_c /root/updates
```

The capture produced 55 packages totalling 152.4 MB in 5.5 seconds, and metadata generation added 1.75 MB in 0.38 seconds; compressed for transport the set was 146.0 MB. On the isolated target — verified unreachable beforehand — the upgrade was performed against that directory alone:

Listing 9.5: Applying the captured content set on the isolated host.

```
# dnf -y --disablerepo="*" \
    --repofrompath=cap,file:///root/updates --enablerepo=cap \
    --nogpgcheck upgrade
```

The transaction completed in 90.1 seconds and produced the same package and kernel set as the online run of Experiment 2. Allowing for the differences between the two runs, the offline upgrade was neither faster nor slower in any meaningful sense: the download that the online run performed was replaced by reads from a local directory, and the remainder — dependency solving, script execution, file installation — dominated both.

9.7.1 What an incomplete capture does

The failure mode that matters in disconnected environments was then reproduced deliberately. On a second isolated guest, one package was deleted from the captured directory before the upgrade, simulating a truncated transfer or an omission during capture. The result is shown in Listing 9.6.

Listing 9.6: An incomplete content set fails late, after the transaction has been resolved.

```
Transaction Summary
=====
Install    4 Packages
Upgrade   51 Packages

Total size: 145 M
Downloading Packages:
Error opening /root/updates/systemd-libs-252-67.el9_8.6.alma.1.x86_64.rpm:
  No such file or directory
Package "systemd-libs-..." from local repository "cap" has incorrect checksum
Error: Some packages from local repository have incorrect checksum
```

Three details are worth drawing out. First, the transaction was resolved, displayed and approved before the problem surfaced: dependency solving uses the metadata, which still described a complete set, so the gap was found only when the file was opened. Second, the diagnostic is misleading — a missing file is reported as an incorrect checksum — which can send an operator looking for corruption rather than for an absent package. Third, nothing was changed: the host remained at 387 packages with the old `systemd`, so the failure was safe but the maintenance window was consumed.

Whether the problem can be found before the window was then tested directly. A resolution-only dry run did not detect it, reporting a complete transaction and stopping at the confirmation prompt. Adding `--downloadonly`, which forces every package to be opened and verified, reproduced the error exactly.

Table 9.5: Which pre-flight check detects an incomplete captured content set.

Check	Work performed	Detects the gap
<code>dnf upgrade --assumeno</code>	metadata, solving only	no
<code>dnf upgrade --downloadonly</code>	solving plus reading every file	yes

The operational recommendation follows directly, and is cheap: an offline upgrade should be rehearsed with `--downloadonly` on a representative host before the change window, because a dry run that only resolves dependencies certifies the metadata rather than the content.

9.8 Threats to the validity of these measurements

The experiments are deliberately small. Each measurement is a single run on one virtualised host of one distribution family, so the timings carry no statistical weight. The guest ran no real workload — no database to migrate, no connection to drain, no cluster peer to coordinate with — which is precisely the situation in which rollback is hardest, and the data-loss observation in Section 9.6 used a placeholder file rather than a live service. The upgrade spanned one minor-release increment; a major-version upgrade, with the orchestration of Chapter 5, would differ in kind. Finally, the content set was captured from a twin at the identical baseline, the favourable case; drift between capture host and target is a common source of the omissions discussed in Chapter 8.

What the experiments do support is narrower and more durable than timings: that applying an upgrade and activating it are separate events with different risk profiles; that the two available rollback mechanisms are complementary in what they preserve and neither covers service state; and that an offline upgrade’s viability is decided by the completeness of the captured content, which can be verified in advance by one specific check.

9.9 Summary

A routine upgrade of a five-week-old image replaced 51 packages and added a kernel in 85.5 seconds without interrupting service, then left eleven processes, including PID 1, running replaced code until a reboot five minutes later activated the change. Reversing that upgrade with `dnf history undo` was refused outright because the host had booted the new kernel, while undo of an ordinary transaction succeeded in under a second and undo without repository access failed for lack of the old packages. A disk snapshot restored a byte-identical system in 0.02 seconds and destroyed every change made after it was taken. An air-gapped host was upgraded successfully from a 152 MB captured content set in 90.1 seconds, and an artificially

incomplete set failed only after the transaction had been resolved and approved — a failure that a resolution-only dry run missed and a `--downloadonly` rehearsal caught. These results are the evidential basis for the discussion in Chapter 11.

10 Security, Compliance, and Lifecycle Management

Previous chapters treated upgrading as an engineering operation. This chapter treats it as an obligation, expressed in a vocabulary of advisories, vulnerability identifiers, scan results, support phases and audit evidence. It connects that vocabulary to the mechanisms already described, so that an administrator can decide which upgrades must be applied, by when, and with what record, and it examines the integrity of the upgrade path itself, because a patching programme that trusts unverified content substitutes one risk for another.

10.1 Errata as the unit of security maintenance

Enterprise distributions do not ship individual patches; they ship advisories. An erratum is a curated bundle of rebuilt packages, a description of what changed, a severity assessment and a list of the issues the rebuild resolves. Vendors classify errata as security, bug fix or enhancement advisories, and the classification carries policy weight: lifecycle documents state which classes are published in which phase of a release's life [41]. The erratum, not the package, is therefore the unit in which security maintenance is planned and reported, and Table 10.1 summarises how the families name these classes.

Table 10.1: Advisory classes across the distribution families.

Family	Advisory naming and classes
Red Hat	Security, bug fix, and enhancement advisories, exposed as <code>dnf updateinfo</code> metadata [40, 41]
Debian	Security advisories, published separately from the stable point release stream [13]
Ubuntu	Security notices, aligned with each release's lifecycle phases [9]
SUSE	Security, recommended, and optional updates, applied through <code>zypper patch</code> [48]
Arch Linux	Advisories against a rolling set, with no backporting: the fix is the newer upstream version [4, 5]

What makes errata operationally useful is that the classification is machine-readable. Repository metadata carries an update information document associating each advisory with its severity, its type, the vulnerability identifiers it addresses and the package builds carrying the fix, so a package manager can report which pending changes are security-relevant and which may be deferred. In the Red Hat family these queries are served by `dnf updateinfo` and the `-security` and `-advisory` selectors on `dnf upgrade` [40, 19]; SUSE offers an equivalent through `zypper list-patches`. A fleet can thus be brought to a defined security state rather than to the newest available one, which matters when that state has not yet been tested.

Listing 10.1: Restricting a transaction to security content.

```
# list pending advisories by type and severity
dnf updateinfo list -available

# apply only the packages carried by security advisories
dnf upgrade -security

# apply one named advisory and its dependencies
dnf upgrade -advisory=<advisory-id>
```

Rolling distributions occupy the opposite position. Because they do not backport fixes into a frozen version, an advisory is satisfied by synchronising against the current repository state,

and the granularity of `dnf upgrade -security` does not exist [4]. Fixes arrive sooner, bundled with every other change, so the auditable security-only transaction is lost.

10.2 Vulnerability identification and machine-readable state

An advisory is a vendor's statement about its own packages; a vulnerability identifier is the cross-vendor name for the underlying defect, and it is what appears in scanner output, supplier questionnaires and audit findings. The link is many-to-many: one advisory typically resolves several identifiers, and one identifier is addressed by different advisories in different releases, at different package versions. Administrators reasoning only in identifiers will call a system vulnerable when the fix has been backported into an unchanged upstream version number, a mismatch that recurs whenever a scanner compares version strings alone.

The machine-readable bridge over this gap is a formal definition language for system state. OVAL expresses what must be true of a host for a vulnerability to apply: which platform it is, which packages are installed, and at which evaluated versions [31]. Vendors publish OVAL definitions alongside their errata, so a scanner evaluates the assertion locally rather than inferring exposure from a version string, which is how backported fixes are recognised.

SCAP assembles OVAL and the related enumeration and scoring formats into a single assessment framework, implemented by OpenSCAP as a scanner that consumes vendor definitions and configuration baselines, reports findings and can generate remediation content [34]. Two uses follow: vulnerability assessment against the installed package set, and compliance assessment of a hardening profile. An upgrade usually improves the first, because it installs the fixed builds, and may worsen the second, because a new version can reintroduce a vendor default the baseline prohibits. Re-scanning afterwards belongs in the post-upgrade verification of Chapter 7, not in a yearly audit.

10.3 Patch cadence, exposure windows, and prioritisation

The interval between the publication of a fix and its installation on a given host is the exposure window, the quantity a patching programme exists to control. NIST guidance frames patch management as preventive maintenance, planned and resourced as a continuing programme with defined responsibilities and measurable timeliness rather than improvised per incident [47]. That guidance is explicit that the window cannot be driven to zero: testing and approval consume time, and pretending otherwise merely moves risk from unpatched software to untested change.

Prioritisation is what makes a non-zero window defensible, and it combines three inputs. Severity assesses the worst plausible impact; exploitability describes whether working attack code exists and how much access it presupposes; exposure is whether the affected code is reachable on the specific host. The third is most often omitted and most changes the answer, since a flaw in an enabled, listening network daemon differs from the same flaw in a library never loaded. General server security guidance makes the point from the other direction: reducing the installed and running surface reduces both the volume of applicable advisories and the urgency of each [46].

This separates two defensible policies that are frequently confused. Patching everything on a fixed cadence is simple, auditable and uniform, at the cost that every advisory consumes a maintenance window and carries regression risk. Patching what is reachable is cheaper for the changes that matter, but depends on a model of what each host runs that decays silently.

Most organisations adopt a hybrid: frequent security-only transactions using the advisory metadata of section 10.1, with a slower full-upgrade cadence on the minor release stream. The experiments in Chapter 9 support the first half and show why the second needs a window, since reversibility degraded once a service had migrated on-disk state.

10.4 Lifecycle management and end-of-life

Cadence governs the life of a release; lifecycle governs the life of the platform. Vendors divide a supported release into phases of narrowing content: an early phase receiving security fixes, bug fixes and enhancements, a later phase restricted to security and selected critical fixes, and sometimes an extended phase covering only the most severe issues [41]. Ubuntu publishes an equivalent structure, separating a long-term-support release’s standard support from the extended maintenance that follows and from the shorter life of an interim release [9]. These documents are authoritative for planning, because they state when the errata stream stops.

The critical distinction, most often left unmade, is between a supported minor release and an end-of-life major release. A host on an older minor release of a supported major version is behind, not unsupported: the fixes exist, and applying them is an ordinary `dnf upgrade` within a stable ABI, as Chapter 4 describes. A host on an end-of-life major version has no fixes to apply, and its remediation is a major-version upgrade or a re-provision, with all the difficulty catalogued in Chapter 5. The two produce identical scanner findings and require entirely different amounts of work, which is why vulnerability counts alone are a poor planning instrument.

Extended support offerings exist to buy time. They keep a defined subset of critical fixes flowing past the normal end of maintenance, at additional cost and reduced scope, without restoring the release to a fully supported state. Buying an extension to avoid an unplanned migration merely converts a deadline into a more expensive one; the defensible use is to decouple the migration from the deadline so that it can be scheduled, rehearsed and rolled back if it fails. Lifecycle tables are published years in advance, so a migration plan begun one support phase before the deadline is the cheapest risk reduction in this domain.

10.5 Supply-chain integrity of the upgrade path

An upgrade grants a remote party the right to replace executable code on a host, usually as root and without further review. That channel’s integrity is a precondition for everything else in this chapter, and a property of the mechanism rather than of the packages it delivers. The base defence is cryptographic signing: packages and repository metadata are signed by the vendor, the public key is installed into the package manager’s trust store, and verification precedes installation [44].

An adversary controlling a mirror, or the network path to one, need not forge a signature to cause harm: it is enough to serve stale but correctly signed metadata, so the client never learns a fixed version exists, to replay an old signed package, or to withhold content from a chosen target [11]. Later work addresses the residual case of a compromised signing key, arguing for role separation and key hierarchies so that one key does not grant arbitrary control over the content a client accepts [45]. The implications are concrete: verify signatures rather than disabling the check to force a stubborn transaction through, treat metadata expiry as a signal, and sign locally rebuilt content with an organisational key.

Beyond transport, provenance concerns how the artefact came to exist. Frameworks such

as SLSA describe levels of build integrity, from unverified local builds up to builds produced by a hardened service emitting signed provenance attestations linking the artefact to its source [33]. For a distribution consumer this becomes a preference for vendor-built packages and a requirement that any internal pipeline be reproducible and attributable. Third-party and locally built content is both the commonest cause of major-version upgrade failure, as Chapter 5 showed, and the least attestable, so the operational and security arguments for minimising it coincide.

Inventory closes the loop. A software bill of materials enumerates the components present in a system with their suppliers and version identifiers, and the minimum elements for such a document specify the fields and transitive depth required for it to be useful [32]. When a vulnerability is announced in a widely embedded library, the first question is which hosts contain it, and the package database answers only for content the package manager installed. Vendor tarballs, bundled language-ecosystem dependencies and statically linked binaries are invisible to `rpm -qa` and `dpkg -l`, and no erratum will fix them.

10.6 Regulated and air-gapped environments

Regulated environments do not change what an upgrade does; they change what must be demonstrated about it afterwards. The auditable artefact is a record of what was changed, on which hosts, by whose authority, when, and with what result. Much of it is already a by-product of the mechanisms described earlier: the transaction history records the packages added, removed and replaced, with timestamps and a citable identifier [19]; `/var/log/dnf.rpm.log` and `/var/log/dpkg.log` record the same events at the lower layer; and scanner output before and after documents the state the change was intended to alter. These artefacts must be captured as the change is made, since a transaction history records a host's past but never substitutes for a missing approval.

The characteristic tension is between a compliance freeze and an exposure window: change control regimes prohibit modification during defined periods while the patching obligation continues to accrue, so an advisory published on the first day of a freeze is an untreated exposure for its full duration. The resolution must be agreed in advance: a severity threshold above which an emergency change path stays open, a freeze sized against the vendor's publication cadence, and a record of deferred advisories so that the accepted risk is visible to the party that accepted it.

Validated cryptographic configurations impose a stricter constraint. Where validation covers a specific version of a cryptographic module, replacing that version can place the system outside the boundary under which it was assessed, however superior the replacement. Upgrades are consequently constrained to versions inside that boundary, accepting a longer exposure window in exchange for continued validity of the assessment. The boundary must be known beforehand, because discovering it afterwards means an unplanned rollback of exactly the kind Chapter 7 describes.

Air-gapped operation adds the logistics developed in Chapter 8. When content arrives through a captured mirror, the trust path must be reconstructed on the isolated side: keys imported deliberately, signatures verified after transfer as well as before, and the captured set recorded so that an isolated host's state can be reproduced. The experiments in Chapter 9 reinforce the risk, since an offline upgrade succeeded or failed on the completeness of the captured content set and the failure surfaced only inside the maintenance window — worse than an inconvenience where the window is bounded and a failure consumes the approval that

permitted the change.

10.7 Configuration management as a security control

Security-focused configuration management guidance treats a documented baseline, controlled change against it, and assessment of each change's security impact as one integrated process [26]. An upgrade is by definition a change to the baseline: it alters package versions, and frequently shipped defaults, unit files and the contents of `/etc`. Managing upgrades outside that process produces a baseline that is documented and a system that is not.

The mechanisms of Chapter 8 supply the enforcement. A system that declares the intended package state, service state and configuration file contents corrects the drift an upgrade introduces, and reports it where it cannot. This also answers the configuration file handling problem of Chapter 4: a file declared by a manifest does not depend on whether the package manager preserved the local copy or installed the vendor's, since the next enforcement run restores the declared state. Configuration management does not make upgrades safe, but it makes their effect on the baseline observable.

10.8 Summary

Security maintenance is organised around errata rather than packages, and the machine-readable metadata attached to each advisory is what permits a minimal, auditable security-only transaction instead of an indiscriminate upgrade. OVAL definitions evaluated through SCAP tooling reconcile cross-vendor vulnerability identifiers with backported fixes, while cadence policy trades exposure against regression risk and prioritisation is defensible only when reachability is weighed alongside severity and exploitability.

Lifecycle policy sets the outer boundary: a supported minor release is behind, whereas an end-of-life major release is unmaintainable and its remediation is a migration, which is what extended support should buy time to rehearse. Underneath sits the integrity of the upgrade path itself: signed content and managed keys, build provenance for locally built content, and an inventory revealing the components no erratum will reach. Regulated and air-gapped settings add evidence requirements and constraints that lengthen exposure windows deliberately, making the planning discipline of the preceding chapters a compliance instrument as much as an engineering one.

11 Discussion

The preceding chapters each answered a bounded question: how a transaction is resolved, what a major-version tool does, how a kernel reaches a boot entry, what an image-based host replaces. This chapter draws those answers together. It returns to the four research questions, identifies the trade-off recurring beneath every mechanism examined, observes where the two management models have converged, distills the analysis into recommendations, and states what the study cannot support.

11.1 Answering the research questions

11.1.1 RQ1: mechanisms and their differences

The mechanisms surveyed in Chapters 3 to 8 do not form a spectrum of increasing thoroughness; they differ in what they treat as the unit of change. A package transaction takes the individual component as the unit and computes a minimal set of changes against the installed state. A major-version tool takes the release as the unit and rewrites the package set wholesale against a new repository definition. An image-based or transactional system takes the entire root filesystem as the unit and replaces it atomically, keeping the previous version bootable [12, 8, 49].

The consequences follow from that choice. A smaller unit gives finer control, faster operations, and a system whose final state depends on its history, being the accumulation of every transaction ever applied. A larger unit gives reproducibility and a clean rollback boundary at the cost of flexibility: the host cannot be adjusted between deployments without leaving the model. The four package managers differ far less than this axis suggests. `dnf`, `apt`, `zypper` and `pacman` all resolve dependencies over a declarative metadata set, and the first three delegate to comparable solver technology [35, 2]. Their visible differences lie in defaults, configuration file policy and whether an undo facility exists, not in the problem they solve.

11.1.2 RQ2: why major-version upgrades are structurally harder

A patch-level upgrade operates inside guarantees the vendor has committed to maintaining: the application binary interface is stable, configuration syntax does not change, and package names persist [41]. Everything that makes a major-version upgrade difficult follows from those guarantees being lifted at once. Chapter 5 showed that the hard cases are not resolution failures but semantic ones: components removed with no successor, defaults inverted, interpreters advanced past the version an application requires, and configuration files whose syntax the new package no longer accepts.

The difference is therefore structural rather than one of degree. A patch-level upgrade is a problem the solver can reason about, because the constraints are expressible in metadata; a major-version upgrade is one the solver cannot fully see, which is why `leapp` interposes a separate analysis phase built from actors encoding knowledge the package metadata does not carry [29]. The pre-upgrade report is an admission that dependency resolution alone is insufficient at this boundary.

11.1.3 RQ3: what determines reversibility

Reversibility is determined by the layer at which the change was captured, not by the tool that applied it. Chapter 4 established that a package manager’s history records which packages were

replaced and can reinstall the previous versions if they remain obtainable [19]. The experiments in Chapter 9 bear this out and expose its limit: transaction-level undo restored files, not service state, and reversibility degraded sharply once a service had migrated its on-disk state, because no record of that migration exists in the package layer.

Snapshot and image mechanisms invert the relationship. A storage or hypervisor snapshot does not know what changed and does not need to; it captures the whole state and restores it wholesale, at the cost of discarding everything written since. In this study that was the only mechanism that returned a host to a byte-identical prior state. The rule that emerges is that a rollback guarantee is only as broad as the layer holding it: the package layer promises packages, the storage layer promises blocks, and neither promises that a dependent service will resume correctly. Application data, which lives above both, is reversible only through a backup taken for that purpose.

11.1.4 RQ4: how offline constraints change practice

Offline operation does not alter the upgrade mechanism; it removes the ability to repair a mistake during execution. An online transaction that discovers a missing dependency fetches it; an offline transaction fails, and Chapter 9 found that failure typically surfaced only inside the maintenance window. The decisive variable is the completeness of the captured content set, fixed long before the upgrade runs and hard to verify without executing the transaction itself.

This shifts the character of the work rather than its quantity. Air-gapped practice depends on disciplined mirroring, a reproducible capture procedure and rehearsal against a staged copy, so that incompleteness surfaces outside the window. It also becomes a security question, because the trust path must be reconstructed on the isolated side, and repository content whose freshness cannot be checked against the vendor is exactly the condition that the mirror attacks in the literature exploit [11].

11.2 The central trade-off

One pattern recurs beneath every mechanism examined: safety is purchased by moving work earlier, and no mechanism removes the work. A snapshot converts an unbounded recovery into a bounded one, but it must be taken, stored and later discarded. A staging environment converts an unknown regression into a known one, but only if maintained as a faithful copy of production. An image-based host makes deployment atomic and rollback trivial in exchange for a build pipeline, a registry and a discipline that forbids ad hoc changes on the host [8]. A captured offline mirror makes an air-gapped upgrade possible and moves the dependency problem into the capture step.

The same holds for the analysis tools. A pre-upgrade report does not make a major-version upgrade safe; it moves the discovery of the obstacles into the weeks preceding the window, where they can be addressed without time pressure [29]. The apparent exception, live patching, is not one: it defers the reboot without eliminating it, and the deferred reboot must still happen when the patch series ends [27, 30]. The experiments support this reading, since a kernel upgrade proved cheap to apply and expensive to validate, the validation being the reboot itself.

Comparing mechanisms by the duration of the change window is therefore misleading. The relevant quantity is total cost, including preparation, verification and a recovery path that may never be used. On that measure a mechanism that looks cheap in the window is often the most expensive overall, because the work it declined to do earlier reappears as incident response.

11.3 Where the models converge

The taxonomy of Chapter 3 separates package-based and image-based management cleanly, but practice is eroding the boundary from both sides. Package-based systems have adopted the strongest property of the image model, a cheap and complete rollback boundary, by moving it into the storage layer: a snapshot taken immediately before a transaction reproduces the atomicity of an image swap, and SUSE's transactional model makes this explicit by performing the transaction in a snapshot and activating it at the next boot [49]. The boot loader specification contributes the other half, making multiple bootable states a first-class arrangement rather than a manual edit [51].

Image-based systems have moved in the opposite direction. The purest form of the model requires every change to pass through a build pipeline, intolerable where a host genuinely needs one additional package. Both `rpm-ostree` and `bootc` therefore allow packages to be layered onto a committed image, preserving atomic deployment and rollback while readmitting some flexibility [12, 8].

What survives the convergence is the difference in default. A package-based host with snapshots is mutable by default and reversible when someone remembers to prepare; an image-based host with layering is immutable by default and mutable only deliberately. Defaults determine what happens on the day nobody prepared, and that is the strongest argument for the image-based model this analysis supports: not that the mechanism is more capable, but that its guarantee does not require foresight.

11.4 Practical recommendations

The analysis supports a short ordered sequence, each step justified by a specific finding of the preceding chapters.

1. **Classify the operation before planning it.** The three classes differ in blast radius and reversibility, and a plan written for the wrong one prepares the wrong recovery.
2. **Decide the rollback layer first, not last.** The recovery path must be chosen before the change and must cover service state and data, which the package layer does not.
3. **Separate security-only transactions from feature-bearing ones.** Advisory metadata makes a minimal, auditable security transaction possible, reducing exposure without importing unrelated regression risk.
4. **Run the pre-upgrade analysis early and treat its findings as work items.** Its value is lost if the report is first read inside the window.
5. **Rehearse offline upgrades against the captured set.** Content completeness is verifiable only by attempting the transaction where failure is affordable.
6. **Verify by observing the service, not the exit status.** A transaction can succeed while a daemon fails to restart, and the reboot is the only verification a kernel upgrade genuinely has.

11.5 Limitations of this study

The experimental component is deliberately narrow, and its narrowness bounds what may be inferred from it. All measurements were taken on a single architecture, x86-64, and within a

single distribution family. Statements about `apt`, `zypper` and `pacman` rest on documentation and literature rather than measurement, and the differences between families in scriptlet behaviour, configuration handling and undo semantics could plausibly change the observed outcomes.

The platform was a virtual machine rather than production hardware, a methodological necessity since hypervisor snapshots are what make the rollback experiments possible, but one that removes firmware updates, physical device driver loading, storage controllers and the boot path variability real servers exhibit. The study is also single-node, so rolling upgrades across members, version skew between nodes and quorum during reboots remain outside its evidence.

No long-running stateful workload was exercised. Reversibility was observed to degrade once a service migrated its on-disk state, but the severity of that for a database holding substantial live data was not measured, and that is the case in which the cost is greatest. The observation windows were also short: regressions appearing only under sustained load, across weeks of log rotation, or at the second reboot fall outside what could be detected.

11.6 Threats to validity

Internal validity. The experiments ran in a controlled environment with a local mirror and warm caches, so observed durations reflect the arrangement as much as the operation; repository latency, mirror selection and download throughput, which dominate transaction time in many real deployments, were deliberately suppressed. The host was also close to a default installation, so the configuration drift, third-party repositories and hand-edited files that characterise long-lived servers were absent — the very conditions under which upgrades most often fail.

External validity. Generalisation beyond the tested family is the principal risk. The Debian family's configuration file prompting, SUSE's snapshot-integrated model and the absence of transaction undo in the rolling model each alter the conclusions enough that the measured results illustrate mechanisms rather than predict behaviour elsewhere. Generalisation to fleets is a second risk: a single host does not exhibit the partial-failure modes, orchestration errors and heterogeneous states that dominate at scale, which Chapter 8 addresses analytically rather than experimentally.

Construct validity. The deepest threat concerns what was measured as success. A transaction returning a zero exit status has installed files; it has not demonstrated that services restarted, that configuration was accepted, or that dependent applications still function. Utilities such as `needs-restarting` narrow the gap by identifying processes still running replaced code [20], but they detect a stale process rather than a broken one. Every observation here about whether an upgrade succeeded should therefore be read as a statement about the mechanism completing, not about the service remaining correct, and that gap motivates the future work of Chapter 12.

11.7 Summary

The four research questions resolve into a consistent picture. Mechanisms differ chiefly in the unit of change they operate on, and that choice fixes both their flexibility and their rollback boundary. Major-version upgrades are harder because the guarantees a patch-level upgrade relies upon are withdrawn at once, leaving obstacles dependency resolution cannot express. Reversibility belongs to the layer at which state was captured, and offline constraints change nothing about the mechanism and everything about the cost of an error.

Beneath all four sits one trade-off: safety is bought by moving effort earlier, and no mechanism removes the effort. The two management models are converging, package-based systems acquiring snapshot rollback and image-based systems acquiring layering, but they still differ in what happens by default when nobody prepared. The evidence is bounded by a single architecture and distribution family, a virtualised single-node platform, short observation windows, and a notion of success that stops at the exit status of a transaction.

12 Conclusion

This chapter closes the thesis. It restates what was undertaken and by what means, returns a verdict on each of the four research questions set out in Chapter 1, condenses the practical guidance into a form an administrator can act on, and identifies the questions that the work leaves open.

12.1 Summary of the work

The thesis set out to treat the upgrading of Linux systems as an engineering activity with identifiable mechanisms, characteristic failure modes and measurable properties, rather than as a routine administrative chore. It proceeded in three movements. The first established the technical foundations and developed a taxonomy that separates upgrade operations by the version distance they cross, by whether the running system is modified in place or replaced, by whether they execute against a live system or in an offline early-boot environment, and by whether they can be undone.

The second examined the mechanisms in detail: dependency resolution and the transaction layer beneath every package manager, major-version in-place upgrade tooling across the Red Hat, Debian and SUSE families, kernel upgrades together with the boot path and live patching, risk management and rollback planning, and automation at fleet scale including image-based and transactional hosts. The third supplied evidence. Chapter 9 reported controlled experiments on an AlmaLinux 9.8 virtual machine covering package upgrades, a kernel upgrade including its reboot, rollback through transaction history, rollback through a disk snapshot, and an upgrade performed entirely offline from a locally captured repository. Chapter 10 related the whole to security obligations and lifecycle policy, and Chapter 11 synthesised the findings and stated their limits.

The four contributions promised in Chapter 1 were delivered in that sequence: a taxonomy organised by the properties that matter operationally, a comparative analysis of the tooling across the Red Hat, Debian, SUSE and rolling families, a body of measured evidence from the virtual-machine experiments, and a practitioner checklist that turns the analysis into an ordered sequence of actions with an explicit rollback decision point.

12.2 Findings against the objectives

RQ1: what mechanisms exist and how do they differ. The mechanisms are distinguished by the unit of change on which they operate rather than by their thoroughness: individual components for a package transaction, the release for major-version tooling, and the whole root filesystem for image-based and transactional systems, which retain the previous version as a bootable deployment [12, 8, 49]. The verdict is that the choice between mechanisms is a choice of granularity, and granularity determines both flexibility and the boundary at which a rollback can be drawn.

RQ2: why major-version upgrades are qualitatively harder. They are harder because interface stability, configuration syntax, package naming and default behaviour are each maintained within a major version and none is maintained across one. The resulting obstacles are semantic rather than combinatorial, which is why a separate analysis phase exists in tooling such as `leapp` and why its report is the single most valuable artefact of the process [29]. The difference is structural, not a matter of scale.

RQ3: what determines whether an upgrade can be rolled back. Reversibility is a property of the layer at which state was captured before the change, not of the tool that applied it. Transaction-level undo restores files rather than service state, and only while the superseded packages remain obtainable; snapshotting at the storage or hypervisor layer returned a host to a byte-identical prior state but discards everything written since, which makes it unsuitable wherever data accrues. The verdict is that no single layer provides a complete rollback, and a credible recovery plan combines a package-level, a block-level and a data-level mechanism with an explicit understanding of what each covers.

RQ4: how offline and air-gapped constraints change practice. They do not change the mechanism; they remove the opportunity to correct an omission while the change is running. The determining variable becomes the completeness of the content set captured in advance, and incompleteness is typically discovered where it is most expensive. Offline practice is consequently a discipline of preparation and rehearsal rather than of execution, and it carries a security dimension, since the trust path must be reconstructed on the isolated side of the boundary [11].

12.3 Recommendations for practitioners

Four recommendations follow from the findings above. First, classify the operation before planning it, because a plan written for the wrong class prepares the wrong remedy. Second, choose the rollback layer before making the change and verify that it covers service state and application data, not only installed files; the package layer demonstrably does not, and discovering this after a failed upgrade converts a reversible change into an outage.

Third, separate security-only transactions from feature-bearing ones. Advisory metadata makes a minimal and auditable security transaction possible [40], allowing exposure to be reduced on a short cadence while larger upgrades are scheduled into windows where verification is possible. Fourth, verify by observing the service rather than the exit status of the tool, since a transaction can complete successfully while a daemon fails to restart or continues to execute replaced code [20].

12.4 Future work

Five directions follow from the limitations stated in Chapter 11. The first and most consequential is automated verification of upgrade outcomes beyond exit status. Defining a machine-checkable post-condition covering service availability, interface conformance, configuration acceptance and performance within a declared envelope would turn the question of whether an upgrade succeeded from a judgement into a measurement.

The second is the measurement of rollback success on stateful services. The present study established that reversibility degrades once on-disk state has migrated, but did not quantify the degradation for realistic database and message-broker workloads, where the answer determines whether rollback is a usable plan or a nominal one.

The third is image-based host management at scale in constrained networks. The transactional and image-based models assume a reachable registry, and their behaviour under the mirroring, capture and approval constraints of an air-gapped fleet deserves empirical treatment rather than the analytical one given in Chapter 8. The fourth is an empirical study of pre-upgrade report accuracy: how often the predicted obstacles materialise, and how often failures occur that the report did not anticipate, which would establish how much confidence the report

warrants.

The fifth is upgrade behaviour on architectures other than x86-64. Boot path differences, firmware interfaces and the availability of vendor tooling on ARM and other platforms are unlikely to leave the conclusions of this thesis wholly intact, and the extent of the divergence is currently unknown.

12.5 Closing statement

The maxim that one should never touch a running system survives because it names a real hazard. It fails as policy because the alternative to change is not stability but the slow accumulation of defects that will not be fixed. What this thesis has attempted to show is that the choice is not between changing and not changing, but between mechanisms that differ in what they guarantee, and that those guarantees can be stated precisely: which layer holds the rollback, which obstacles the resolver can see and which it cannot, and which part of the work has been moved earlier rather than eliminated. An upgrade planned with those properties in view is not a risk accepted blindly. It is an engineering decision with a recovery path, and that is the difference between maintaining a system and merely operating it.

Bibliography

- [1] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: A separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, 2012. doi: 10.1016/j.jss.2012.02.018.
- [2] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. Dependency solving is still hard, but we are getting better at it. In *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER ’20)*, pages 547–551. IEEE, 2020. doi: 10.1109/SANER48275.2020.9054837.
- [3] AlmaLinux OS Foundation. ELevate: Migrating between enterprise linux derivatives, 2026. URL <https://wiki.almalinux.org/elevate/>. Accessed 2026-09-20.
- [4] Arch Linux. System maintenance — archwiki, 2026. URL https://wiki.archlinux.org/title/System_maintenance. Accessed 2026-09-20.
- [5] Arch Linux. pacman — archwiki, 2026. URL <https://wiki.archlinux.org/title/Pacman>. Accessed 2026-09-20.
- [6] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys ’09)*, pages 187–198. ACM, 2009. doi: 10.1145/1519065.1519085.
- [7] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. Release engineering — site reliability engineering, 2016. URL <https://sre.google/sre-book/release-engineering/>. Accessed 2026-09-20.
- [8] bootc Project. bootc: Boot and upgrade via container images, 2026. URL <https://bootc-dev.github.io/bootc/>. Accessed 2026-09-20.
- [9] Canonical Ltd. Ubuntu release cycle, 2026. URL <https://ubuntu.com/about/release-cycle>. Accessed 2026-09-20.
- [10] Canonical Ltd. Upgrade your release — ubuntu server documentation, 2026. URL <https://documentation.ubuntu.com/server/how-to/software/upgrade-your-release/>. Accessed 2026-09-20.
- [11] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. A look in the mirror: Attacks on package managers. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS ’08)*, pages 565–574. ACM, 2008. doi: 10.1145/1455770.1455841.
- [12] CoreOS Project. rpm-ostree: Hybrid image/package system, 2026. URL <https://coreos.github.io/rpm-ostree/>. Accessed 2026-09-20.
- [13] Debian Project. Debian release notes: Upgrades from previous releases, 2023. URL <https://www.debian.org/releases/bookworm/amd64/release-notes/ch-upgrading.en.html>. Accessed 2026-09-20.
- [14] Debian Project. apt(8) — command-line interface, 2026. URL <https://manpages.debian.org/stable/apt/apt.8.en.html>. Accessed 2026-09-20.

- [15] Debian Project. Debian policy manual, 2026. URL <https://www.debian.org/doc/debian-policy/>. Accessed 2026-09-20.
- [16] Debian Project. Unattendedupgrades — debian wiki, 2026. URL <https://wiki.debian.org/UnattendedUpgrades>. Accessed 2026-09-20.
- [17] DNF Project. DNF5 documentation, 2026. URL <https://dnf5.readthedocs.io/en/latest/>. Accessed 2026-09-20.
- [18] DNF Project. dnf-automatic(8) — automatic dnf upgrades, 2026. URL <https://man7.org/linux/man-pages/man8/dnf-automatic.8.html>. Accessed 2026-09-20.
- [19] DNF Project. dnf(8) — dnf command reference, 2026. URL <https://man7.org/linux/man-pages/man8/dnf.8.html>. Accessed 2026-09-20.
- [20] DNF Project. needs-restarting(1), 2026. URL <https://man7.org/linux/man-pages/man1/needs-restarting.1.html>. Accessed 2026-09-20.
- [21] dracut Project. dracut(8) — initramfs generation tool, 2026. URL <https://man7.org/linux/man-pages/man8/dracut.8.html>. Accessed 2026-09-20.
- [22] Fedora Project. Upgrading fedora linux using the offline method, 2026. URL <https://docs.fedoraproject.org/en-US/quick-docs/upgrading-fedora-offline/>. Accessed 2026-09-20.
- [23] GNU Project. GNU GRUB manual 2.12, 2026. URL <https://www.gnu.org/software/grub/manual/grub/grub.html>. Accessed 2026-09-20.
- [24] GNU Project. libabigail: Application binary interface generic analysis and instrumentation library, 2026. URL <https://sourceware.org/libabigail/manual/libabigail-overview.html>. Accessed 2026-09-20.
- [25] Google LLC. Canarying releases — the site reliability workbook, 2018. URL <https://sre.google/workbook/canarying-releases/>. Accessed 2026-09-20.
- [26] Arnold Johnson, Kelley Dempsey, Ron Ross, Sarbari Gupta, and Dennis Bailey. Guide for security-focused configuration management of information systems. Technical Report NIST Special Publication 800-128, National Institute of Standards and Technology, 2011.
- [27] kpatch Project. kpatch: Dynamic kernel patching, 2026. URL <https://github.com/dynup/kpatch>. Accessed 2026-09-20.
- [28] Greg Kroah-Hartman. The linux kernel driver interface (stable api nonsense), 2026. URL <https://www.kernel.org/doc/html/latest/process/stable-api-nonsense.html>. Accessed 2026-09-20.
- [29] Leapp Project. Leapp framework documentation, 2026. URL <https://leapp.readthedocs.io/en/latest/>. Accessed 2026-09-20.
- [30] Linux Kernel Organization. Livepatch — the linux kernel documentation, 2026. URL <https://www.kernel.org/doc/html/latest/livepatch/livepatch.html>. Accessed 2026-09-20.

- [31] MITRE Corporation. Open vulnerability and assessment language (oval), 2023. URL <https://oval.mitre.org/about/index.html>. Accessed 2026-09-20.
- [32] National Telecommunications and Information Administration. The minimum elements for a software bill of materials (sbom). Technical report, United States Department of Commerce, 2021. URL <https://www.ntia.gov/report/2021/minimum-elements-software-bill-materials-sbom>.
- [33] Open Source Security Foundation. SLSA specification, version 1.0, 2023. URL <https://slsa.dev/spec/v1.0/about>. Accessed 2026-09-20.
- [34] OpenSCAP Project. Openscap: Security compliance solution — features, 2026. URL <https://www.open-scap.org/features/>. Accessed 2026-09-20.
- [35] openSUSE Project. libsolv: Library for solving packages and reading repositories, 2026. URL <https://github.com/openSUSE/libsolv>. Accessed 2026-09-20.
- [36] openSUSE Project. SDB: System upgrade, 2026. URL https://en.opensuse.org/SDB:System_upgrade. Accessed 2026-09-20.
- [37] Tom Preston-Werner. Semantic versioning 2.0.0, 2013. URL <https://semver.org/>. Accessed 2026-09-20.
- [38] Red Hat, Inc. Upgrading from RHEL 9 to RHEL 10, 2026. URL https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/10/html-single/upgrading_from_rhel_9_to_rhel_10/index. Accessed 2026-09-20.
- [39] Red Hat, Inc. Upgrading from RHEL 8 to RHEL 9, 2026. URL https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html-single/upgrading_from_rhel_8_to_rhel_9/index. Accessed 2026-09-20.
- [40] Red Hat, Inc. Managing software with the DNF tool, 2026. URL https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html-single/managing_software_with_the_dnf_tool/index. Accessed 2026-09-20.
- [41] Red Hat, Inc. Red hat enterprise linux life cycle and errata support policy, 2026. URL <https://access.redhat.com/support/policy/updates/errata>. Accessed 2026-09-20.
- [42] Red Hat, Inc. Managing, monitoring, and updating the kernel, 2026. URL https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html-single/managing_monitoring_and_updating_the_kernel/index. Accessed 2026-09-20.
- [43] Rocky Enterprise Software Foundation. Migrating to rocky linux with migrate2rocky, 2026. URL <https://docs.rockylinux.org/guides/migrate2rocky/>. Accessed 2026-09-20.
- [44] RPM Project. RPM reference manual, 2026. URL <https://rpm-software-management.github.io/rpm/manual/>. Accessed 2026-09-20.
- [45] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*, pages 61–72. ACM, 2010. doi: 10.1145/1866307.1866315.

- [46] Karen Scarfone, Wayne Jansen, and Miles Tracy. Guide to general server security. Technical Report NIST Special Publication 800-123, National Institute of Standards and Technology, 2008.
- [47] Murugiah Souppaya and Karen Scarfone. Guide to enterprise patch management planning: Preventive maintenance for technology. Technical Report NIST Special Publication 800-40 Revision 4, National Institute of Standards and Technology, 2022.
- [48] SUSE LLC. Upgrading online — suse linux enterprise server documentation, 2026. URL <https://documentation.suse.com/sles/15-SP6/html/SLES-all/cha-upgrade-online.html>. Accessed 2026-09-20.
- [49] SUSE LLC. Transactional updates in suse linux enterprise micro, 2026. URL <https://documentation.suse.com/sle-micro/6.0/html/Micro-transactional-updates/index.html>. Accessed 2026-09-20.
- [50] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Opium: Optimal package install/uninstall manager. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 178–188. IEEE, 2007. doi: 10.1109/ICSE.2007.59.
- [51] UAPI Group. The boot loader specification, 2026. URL https://uapi-group.org/specifications/specs/boot_loader_specification/. Accessed 2026-09-20.

A Command Reference

This appendix collects the commands used throughout the thesis into one family-by-family reference. Each section gives a table of invocation and purpose, followed by notes on the entries whose behaviour is not obvious. Every command that modifies the system is assumed to run as root.

A.1 RHEL family: dnf, leapp and system-upgrade

Table A.1: Package and upgrade commands on the RHEL family.

Command	Purpose
<code>dnf check-update</code>	Lists available updates; exit status 100 if any exist.
<code>dnf upgrade</code>	Applies all available updates in one transaction.
<code>dnf upgrade --security</code>	Restricts the transaction to advisory packages.
<code>dnf updateinfo list</code>	Enumerates advisories applicable to the installed set.
<code>dnf history list</code>	Lists past transactions with identifier and date.
<code>dnf history info <id></code>	Prints the packages and command line of a transaction.
<code>dnf history undo <id></code>	Applies the inverse of a transaction.
<code>dnf needs-restarting</code>	Lists processes whose on-disk files were replaced.
<code>dnf versionlock add <pkg></code>	Pins a package against later transactions.
<code>leapp preupgrade</code>	Runs the analysis actors and writes a report only.
<code>leapp upgrade</code>	Stages the target content and upgrade initramfs.
<code>dnf system-upgrade download</code>	Fedora: caches a release for an offline upgrade.

The `--security` filter matches only packages listed in the `updateinfo` section of the repository metadata, so a repository lacking that section reports nothing to do, indistinguishably from a fully patched system [19, 40]; this is the commonest defect in local mirrors and recurs in section A.6.

`dnf history undo` is not a general rollback facility: it applies an inverse package set as a new transaction, succeeding only while the older packages remain retrievable, and even then restores files owned by packages rather than the state scriptlets, services or applications created afterwards [40]. The mechanics are treated in chapter 4.

`leapp preupgrade` runs the same inhibitor checks as `leapp upgrade` but stops before content is fetched, so it may be repeated on a production host during planning [29]. The Fedora equivalent, `dnf system-upgrade`, downloads the target release and reboots into a minimal environment to apply the transaction with nothing running [22].

A.2 Debian family: apt and dpkg

`apt upgrade` may never remove an installed package: where a newer version could be installed only by removing another, the package is left alone and reported as kept back. `apt full-upgrade` lifts that constraint, which is why release upgrades use it and why its proposed action list must be read before confirmation [14]. A long kept-back list is therefore not a defect but a signal that a pending transition needs the permissive form.

A package held by `apt-mark hold` still participates in resolution, so a hold on a library can silently prevent unrelated packages from advancing; holds belong in the pre-upgrade inventory,

Table A.2: Package and upgrade commands on the Debian family.

Command	Purpose
<code>apt update</code>	Refreshes the package index from the sources.
<code>apt list --upgradable</code>	Reports packages with newer candidates.
<code>apt upgrade</code>	Installs newer versions, removing nothing.
<code>apt full-upgrade</code>	As above, but may remove packages.
<code>apt-mark hold <pkg></code>	Excludes a package from upgrades.
<code>unattended-upgrade --dry-run</code>	Simulates the unattended run.
<code>do-release-upgrade</code>	Upgrades Ubuntu to the next release.
<code>dpkg --audit</code>	Reports unpacked but unconfigured packages.
<code>dpkg --configure -a</code>	Configures half-installed packages.
<code>debsums -c</code>	Verifies files against dpkg checksums.

recorded with `apt-mark showhold`. `dpkg --audit` exists because a `dpkg` transaction unpacks before it configures, and an interruption between the phases leaves packages unconfigured, a state repaired with `dpkg --configure -a` rather than by repeating the `apt` invocation. `debsums` is the Debian counterpart of `rpm -Va`, limited to packages that shipped checksums.

A.3 SUSE: zypper and transactional-update

Table A.3: Package and upgrade commands on SUSE systems.

Command	Purpose
<code>zypper refresh</code>	Refreshes metadata of enabled repositories.
<code>zypper patch</code>	Applies only what issued advisories prescribe.
<code>zypper update</code>	Moves packages to newer versions, removing nothing.
<code>zypper dup</code>	Aligns the installed set with the repositories, allowing vendor change, downgrade and removal.
<code>zypper ps</code>	Lists processes still using replaced files.
<code>zypper lifecycle</code>	Reports support end dates of packages and products.
<code>transactional-update dup</code>	Upgrades inside a snapshot activated at next boot.

`zypper patch` and `zypper update` are not synonyms: the former is advisory-driven and honours the flags a patch carries, including the indication that the package manager or the system must be restarted, while the latter is version-driven. `zypper dup` is the most permissive form and the supported path for service-pack migration on SUSE Linux Enterprise and for openSUSE Tumbleweed [48].

`zypper ps` answers the question left by every library update: processes that mapped the previous inode run the old code until restarted, so a host can be patched by package version and still execute vulnerable code. On transactional systems the question does not arise, because `transactional-update` writes to a snapshot that takes effect only at the next reboot, leaving the previous one as a rollback target [49].

A.4 Arch Linux: pacman

On a rolling release the only supported operation is the full system upgrade: refreshing the databases with `pacman -Sy` and then installing one package produces a partial upgrade in which that package is linked against library versions the rest of the system does not have

Table A.4: Package and upgrade commands on Arch Linux.

Command	Purpose
<code>pacman -Syu</code>	Synchronises databases and upgrades everything.
<code>checkupdates</code>	Lists pending updates from a temporary database.
<code>pacman -Syuw</code>	Downloads a full upgrade without installing.
<code>pacdiff</code>	Merges <code>.pacnew</code> and <code>.pacsave</code> configuration files.
<code>paccache -r</code>	Prunes the cache, keeping recent versions.

[5]. `checkupdates` reports what is pending without creating that state, because it copies the database to a temporary location first. `pacdiff` addresses the other trait of this family: `pacman` writes a new configuration file beside a modified one rather than overwriting it, and those files accumulate until merged.

A.5 Verification after an upgrade

Table A.5: Commands used to verify a system after a transaction.

Command	Purpose
<code>rpm -Va</code>	Verifies files against size, mode, checksum, owner.
<code>rpm -qa --last</code>	Lists packages by install time, newest first.
<code>needs-restarting -r</code>	Signals by exit status whether a reboot is due.
<code>systemctl list-units --failed</code>	Lists units that failed to start.
<code>journalctl -p err -b</code>	Shows error-priority messages of this boot.
<code>pacman -Qkk</code>	The <code>pacman</code> file-level verification.

`rpm -Va` is informative only against a baseline, since a healthy system reports every configuration file an administrator has edited; the signal is the difference between output taken before and after the upgrade. `rpm -qa --last` reconstructs what changed most recently even when the transaction history is unavailable.

`needs-restarting -r` reports on the system rather than individual processes, signalling through its exit status whether a reboot is needed, which makes it usable as the condition of an automated reboot step [20]; without `-r` it is the RPM-family counterpart of `zypper ps`. `dnf-automatic` commonly applies updates on a schedule while leaving the restart decision to this check [18].

A.6 Repository mirroring and offline capture

Table A.6: Commands for capturing and publishing repository content locally.

Command	Purpose
<code>reposync --download-metadata</code>	Mirrors an RPM repository with its metadata.
<code>createrepo_c <dir></code>	Generates metadata for a directory of RPMs.
<code>dnf download --resolve</code>	Downloads packages and missing dependencies.
<code>apt-mirror</code>	Mirrors Debian suites from a configuration file.
<code>debmirror</code>	Mirrors an archive with explicit suite, architecture and signature selection.
<code>zypper clean --all</code>	Empties local metadata and package caches.

The critical option is `--download-metadata`. Without it, `reposync` leaves the repository to be rebuilt with `createrepo_c`, which yields valid metadata but not the original `updateinfo` advisory data. Such a mirror installs packages correctly and silently breaks every advisory-driven operation, including `dnf upgrade --security` and the compliance scans reading the same data. Whether metadata is mirrored or regenerated should be a recorded decision.

`dnf download --resolve` computes the dependency closure relative to what is installed on the machine running it, so the captured set is correct there and possibly incomplete elsewhere; content for a fleet must be captured as a repository rather than as a closure. The clean commands serve the same end: they drop cached content so that a rehearsal exercises the transported material rather than what the test host already holds.

A.7 Summary

The commands above fall into three groups: those that report without changing anything, those that change the system, and those that verify the result. Reporting commands are safe at any time; changing commands differ chiefly in how permissive their solver may be; verifying commands mean little without a baseline. The checklists in Appendix B sequence the three groups into procedures.

B Operational Checklists

This appendix turns the preceding chapters into four procedures: routine security patching of a single server, a major-version in-place upgrade, an upgrade under offline constraints, and a rehearsal of rollback. Each item states what to do and why, since a checklist whose rationale is invisible is soon abandoned, and each procedure names its point of no return. The commands appear in Appendix A.

B.1 Routine security patching of a single server

This is the highest-frequency procedure and the one that most rewards discipline. It assumes one host with a service owner, a backup and a window permitting a restart.

1. Confirm that change is permitted now, so no freeze is breached.
2. Record the pre-change state: packages, kernel, repositories and last transaction identifier, without which verification cannot separate new faults from old.
3. Refresh metadata and list what would change without applying it, the last chance to notice a moved repository or oversized transaction.
4. Read the advisories behind the change, which state whether a reboot or a restart is required and so how long the window must be.
5. Confirm a usable restore path and an acceptable backup age; an unexercised restore is a plan, not a capability.
6. Take a snapshot where the storage layer or hypervisor offers one: the most complete rollback available, and the cheapest.
7. Apply the transaction and capture its output. **This is the point of no return:** nothing on disk changes until the plan is confirmed, but once the package manager writes, the host can only go forward, be reversed by a further transaction, or be restored.
8. Restart whatever still runs old code, rebooting if the kernel or core libraries changed, since processes keep the library they mapped.
9. Verify: no failed units, no new journal errors this boot, and a check exercising the application, not its port.
10. Close the record with the transaction identifier, advisories and any deviation, where the next investigation will start.

B.2 Major-version in-place upgrade

A major upgrade changes many subsystems at once, so most of the work lies before the window. The checklist covers preparation, execution and verification with three explicit go/no-go decisions, and applies alike to `leapp`, `do-release-upgrade` and `zypper dup`, which differ in where they stage the work rather than in the risks they create [29, 48].

1. Establish that the jump is a supported path and record both versions; an unsupported pair has no documented remedy.
2. Inventory third-party repositories, out-of-tree modules and held packages, which the vendor cannot map onto the new release.
3. Run the analysis phase, which changes nothing, and resolve every inhibitor rather than override it; each predicts a failure in the window.
4. Rehearse on a clone, not a fresh installation: accumulated drift breaks upgrades, and a clean machine has none.
5. Decide in advance how each replaced or shadowed configuration file will be merged; deciding inside the window guarantees deciding badly.
6. Confirm that a full restore is tested and note its duration, which sets the window's true length.
7. Secure console access independent of the network and the shell daemon, which the upgrade may legitimately leave unavailable.

Go/no-go 1 (before the window): proceed only if no inhibitor is unresolved, the rehearsal completed, the restore was tested, a console exists and verifying staff are available. Any no postpones the change at no cost.

8. Announce the window and suspend competing automation, which would make the upgrade unreproducible.
9. Stop the application services, then take the final snapshot; one taken mid-write may need recovery on restore.

Go/no-go 2 (immediately before execution): the last moment at which abandoning costs only the window; after it, it costs a restore.

10. Start the upgrade and let it finish uninterrupted. **This is the point of no return:** once the new package set is applied — after the reboot into the upgrade environment on the RHEL family, or once the permissive transaction starts on Debian and SUSE — the host is neither release, and the only return is a restore.
11. Preserve the upgrade logs first; they record which step failed, and a later reboot may rotate them away.
12. Verify the boot: expected kernel, consistent entries, no failed unit, the upgrade having regenerated the initramfs and boot configuration.
13. Verify the package state: no previous-release packages remain, nothing is half-configured, and file verification differs from the baseline only as expected.
14. Verify the configuration layer: merged files, access control labels and firewall rules, reset more often than packages break.
15. Exercise the application against the signals recorded beforehand; a host can pass every package check and still fail to serve.

Go/no-go 3 (before traffic returns): accept the upgrade or roll back now. Once the application writes data again, restoring the pre-upgrade snapshot discards it, so the decision costs more by the minute.

B.3 Offline or air-gapped upgrade

Offline upgrading fails on the completeness of the captured content set far more often than on the transfer, so completeness is proved before the material is moved.

1. Define the target state: release, repositories and capture time, since the content set, not the command, is the unit of reproducibility.
2. Capture on a connected host of the same architecture and release, mirroring repositories rather than single packages: only a mirror serves other hosts.
3. Include everything the transaction may consult — advisory metadata, signing keys, upgrade and boot tooling — since a missing key stops it like a missing package.
4. Verify signatures on the connected side and record a checksum manifest, the last place a damaged download is replaced cheaply.
5. Rehearse in a clean, disconnected environment built from the target baseline and served only by the captured set: the only test of completeness.
6. Repair gaps by re-capturing the repository, never by adding single packages, which reproduce the capturing machine, not the set intended.
7. Transport the set on the approved medium, carrying the manifest separately so the check does not rely on what it verifies.
8. Re-verify checksums and signatures on arrival, before publishing locally; a medium damaged in transit fails mid-transaction.
9. Publish the content locally and clear the caches, so the dry run consumes the transported material only.
10. Dry-run on the target or its clone, then execute. **The point of no return is that of the online mechanism, but less forgiving:** no network can supply a forgotten dependency, so an incomplete set fails mid-transaction rather than before it.
11. Archive the content set with its manifest, without which the machine cannot later be reproduced.

B.4 Rollback rehearsal

Rollback is the part of an upgrade plan least tested and most likely to be needed under pressure. It is rehearsed on a clone, whose destruction is the only irreversible step.

1. State what rollback means here: which state, which data, how much loss and downtime are acceptable; undefined targets cannot be missed or improved.
2. Clone the production host, storage layout included, and work only there; rehearsing on production creates the incident it was to prevent.

3. Choose the mechanism under test — package reversal, snapshot restore, previous boot environment, or re-provisioning — and exercise each separately, their preconditions differing.
4. Reproduce a realistic failure, such as a service that starts but misbehaves; these are noticed late, which makes rollback hard.
5. Perform the rollback under real constraints, by the console path and documented steps, timed against the recovery time objective.
6. Inspect the restored system for data written between snapshot and failure, making the implied loss visible before it is real.
7. Exercise package-level reversal where previous packages have left the repository, to observe that failure mode outside an incident.
8. Check the boot path when a kernel is involved: an earlier kernel needs a boot entry and initramfs, which aggressive cleanup removes.
9. Record which mechanisms are genuinely available, with preconditions and durations, and feed that back into the checklists above.
10. Repeat after any change to storage, backup tooling or boot configuration, which invalidates the previous rehearsal.

B.5 Summary

The four procedures share a structure: establish a baseline, prove the change on something disposable, decide explicitly whether to proceed, execute without interruption, and verify against the baseline rather than expectation. They differ in where the point of no return lies — at the commit of a small transaction, at the start of a staged major upgrade, at a transaction with no network behind it, and, in the rehearsal, nowhere. Knowing that position separates a planned upgrade from an outage.

C Experimental Data

This appendix records the raw material behind Chapter 9 so that the experiments can be repeated and the reported figures checked. The captured console output of every run is stored alongside the thesis source in the directory `experiments/`; the file names given below identify each record.

C.1 Reproducing the environment

Every guest was created from one unmodified AlmaLinux 9.8 generic cloud image, using a copy-on-write overlay so that each experiment began from an identical state and left the base image untouched. The commands below create the disks and start a guest. The second form isolates the guest: `restrict=on` makes QEMU’s user-mode network drop everything the guest sends, while the forwarded port keeps administrative access available.

Listing C.1: Creating and starting an experimental guest.

```
# an overlay on the shared, read-only base image
qemu-img create -f qcow2 -F qcow2 \
    -b AlmaLinux-9-GenericCloud-9.8-20260810.x86_64.qcow2 vm.qcow2 40G

# connected guest
qemu-kvm -machine q35,accel=kvm -cpu host -smp 8 -m 8G \
    -drive file=vm.qcow2,if=virtio,format=qcow2 \
    -drive file=seed.iso,if=virtio,format=raw,readonly=on \
    -netdev user,id=n0,hostfwd=tcp:127.0.0.1:2522-:22 \
    -device virtio-net-pci,netdev=n0 -display none

# isolated guest: the only difference is restrict=on
-netdev user,id=n0,restrict=on,hostfwd=tcp:127.0.0.1:2722-:22
```

Configuration was supplied by a NoCloud seed image containing one public key and no packages, so the guest remained a stock installation. Isolation was confirmed inside every offline guest before measurement by observing that name resolution timed out and that `dnf` could not reach its mirror list.

C.2 Baseline and upgrade measurements

Table C.1: Baseline of the unmodified image (`e1-baseline.txt`).

Measure	Value
Release	AlmaLinux 9.8 (Olive Jaguar)
Kernel	5.14.0-687.36.1.el9_8.x86_64
Packages installed	388
Size per the RPM database	822,792,041 bytes
Root filesystem used	1,201,078,272 bytes
Kernels installed	1
Start-up time	6.085 s (0.827 s kernel, 1.825 s initrd, 3.432 s userspace)
Updates pending	55
<code>dnf check-update</code> time	0.31 s

The small difference in package counts between the two columns is an artefact of the experiment rather than of the upgrade: the connected guest had the `time` utility installed

Table C.2: Upgrade transactions, online and offline (`e2-upgrade.txt`, `e6a.txt`, `e6b.txt`).

Measure	Online	Offline
Packages upgraded	51	51
Packages installed	4	4
Content size	145 MB	145 MB
Wall-clock time	85.5 s	90.1 s
Peak resident memory of <code>dnf</code>	133.9 MB	not recorded
Packages before / after	388 / 392	387 / 391

beforehand so that the transaction could be measured.

Table C.3: Capture of the content set on a connected twin (`e6a.txt`).

Measure	Value
Packages captured	55
Size of the captured packages	152,438,479 bytes
Download time	5.5 s
Repository metadata generation time	0.38 s
Size of the generated metadata	1,750,246 bytes
Compressed transfer archive	146,014,429 bytes

C.3 Rollback measurements

Table C.4: The two rollback mechanisms as measured (`e4-undo.txt`, `e4b-undo.txt`, `e4c-undo.txt`, `e5-snapshot.txt`).

Operation	Outcome	Time
Undo of the 55-package upgrade	refused: would remove the protected package <code>kernel-core</code> , which the host had booted	0.45 s
Undo of a package installation	succeeded, package removed	0.52 s
Undo of a removal, repositories disabled	failed: “Cannot find rpm nevra” for both packages	—
Undo of the same removal, repositories enabled	succeeded	1.19 s
Snapshot creation (guest powered off)	65,612 bytes of metadata added to the overlay	0.01 s
Snapshot restore	package fingerprint returned to the recorded post-upgrade value; all changes made after the snapshot lost	0.02 s

Four system fingerprints were recorded during the rollback experiments. After the upgrade and reboot the value was `e76f386b83c30793`; it was unchanged after the refused undo; it became `a8e67b2b56420e59` once further changes had been made; and it returned to `e76f386b83c30793` after the snapshot was restored.

The identity of the first two shows that the refused undo changed nothing; the identity of the first and last shows that the snapshot restore returned the package set exactly to its earlier state.

C.4 Detecting an incomplete content set

One package was deleted from the captured directory to simulate a transfer that lost a file. Two pre-flight checks were then compared on the isolated host (`e6c.txt`, `e6d.txt`).

Table C.5: Pre-flight checks against an incomplete content set.

Command	Observed behaviour	Detects
<code>dnf upgrade --assumeno</code>	resolved a complete 55-package transaction from the metadata and stopped at the prompt	no
<code>dnf upgrade --downloadonly</code>	resolved, then failed opening the missing file, reported as an incorrect checksum	yes
<code>dnf upgrade</code>	identical failure, after the transaction had been displayed and accepted; the system was left unchanged	yes, too late

C.5 Index of captured output

Table C.6: Files in `experiments/`.

File	Content
<code>e1-baseline.txt</code>	baseline characterisation of the unmodified image
<code>e2-check.txt</code> , <code>e2-available.txt</code>	the pending update list and the time taken to compute it
<code>e2-upgrade.txt</code>	the complete online upgrade transaction
<code>e2-post.txt</code>	package counts, sizes and history after the upgrade
<code>e3-pre.txt</code> , <code>e3-post.txt</code>	restart analysis and boot entries before and after the reboot
<code>e4-undo.txt</code>	the refused undo of the upgrade transaction
<code>e4b-undo.txt</code>	a successful undo of an ordinary transaction
<code>e4c-undo.txt</code>	undo with and without repository access
<code>e5-snapshot.txt</code>	snapshot creation, the state before and after restoration
<code>e6a.txt</code>	capture of the content set on the connected twin
<code>e6b.txt</code>	the offline upgrade on the isolated host
<code>e6c.txt</code>	the deliberately incomplete content set
<code>e6d.txt</code>	comparison of the two pre-flight checks

The record for Experiment 4 and the state captured immediately before the snapshot restore were reconstructed from the console transcript: the files had been written inside the guest after the snapshot was taken, and the restore destroyed them. The incident is a small demonstration of the property under study, and it is the reason the remaining records were copied out of each guest as soon as they were produced.