PatchMon PATCHMON BLOG

The Complete Guide to Linux Patch Management in 2026

Everything sysadmins need to know about managing patches across Linux servers - from package managers and security updates to automation, compliance, and tooling.

AUTHOR

PatchMon Team

PUBLISHED

21 March 2026

CATEGORIES

Tutorials, Security

READ ONLINE

https://patchmon.net/blog/complete-guide-linux-patch-management

Contents

If you manage Linux servers for a living, you already know the drill: a new CVE drops, your phone buzzes, and you spend the next few hours figuring out which of your 200 machines are affected, whether the patch will break anything, and how to roll it out without waking anyone up at 3 AM.

Patch management on Linux is one of those problems that sounds simple - "just run `apt upgrade`" - until you're responsible for a fleet of machines spanning three distros, two cloud providers, and a handful of bare-metal boxes in a colo that nobody wants to reboot.

This guide covers everything: how Linux package managers actually handle patches, the real challenges of mixed environments, building a workflow that holds up under pressure, compliance requirements, and the tooling landscape. Whether you're managing 10 servers or 10,000, the fundamentals are the same.

Why Patch Management Matters

Let's skip the abstract risk talk and look at what actually happens when patching fails.

Real-World Consequences

Equifax (2017): Apache Struts CVE-2017-5638 was disclosed on March 7, 2017. A patch was available the same day. The US-CERT notified Equifax directly on March 8. Equifax's internal security team sent an email on March 9 telling admins to patch. Their vulnerability scans on March 15 failed to detect the affected system. Attackers exploited the vulnerability starting in May. The breach wasn't discovered until July 29. 147 million people had their data exposed. The eventual cost: $1.38 billion.

The patch existed for over two months before the breach began. This wasn't a zero-day. It was a process failure.

WannaCry (2017): Microsoft released MS17-010 in March 2017. The WannaCry ransomware hit in May, exploiting the exact vulnerability that patch addressed. Organizations that had applied the two-month-old patch were unaffected. Those that hadn't - including parts of the UK's NHS - lost access to critical systems. Surgeries were cancelled. Ambulances were diverted.

The Pattern: Most breaches don't involve exotic zero-days. They involve known vulnerabilities with available patches that weren't applied in time. Verizon's Data Breach Investigations Report has made this point repeatedly: the majority of exploited vulnerabilities had patches available for months or years before the breach occurred.

The CVE Volume Problem

The scale of the problem is growing. After the Linux kernel team became a CVE Numbering Authority (CNA) in 2024, the rate of disclosed kernel CVEs increased dramatically - from around 300 per year to thousands. In 2024 alone, 3,529 Linux kernel CVEs were recorded.

That number grew further in 2025.

Not all of these are equally severe, and many affect subsystems you may not even use. But the sheer volume means you can't evaluate each one manually. You need a process.

How Linux Package Managers Handle Security Updates

Every major Linux distribution has its own package manager, its own repository structure, and its own approach to security updates. Understanding the differences matters, especially if you manage a mixed fleet.

APT (Debian, Ubuntu)

Debian and Ubuntu separate their repositories into distinct channels. On Ubuntu, security updates come from `<release>-security`, while bug fixes and minor updates land in `<release>-updates`.

```
# Check for available security updates only
apt list --upgradable 2>/dev/null | grep -i security

# Apply security updates only
apt-get upgrade -y -o Dir::Etc::SourceList=/etc/apt/sources.list \
  -o Dir::Etc::SourceParts=/dev/null

# Or use unattended-upgrades for automatic security patching
apt install unattended-upgrades
dpkg-reconfigure -plow unattended-upgrades
```

The `unattended-upgrades` package is the standard way to automate security patching on Debian-family systems. Its configuration lives in `/etc/apt/apt.conf.d/50unattended-upgrades`, and you can control exactly which origins are allowed - typically just security updates.

Key configuration in `/etc/apt/apt.conf.d/50unattended-upgrades`:

```
Unattended-Upgrade::Allowed-Origins {
  "${distro_id}:${distro_codename}-security";
};
Unattended-Upgrade::Automatic-Reboot "false";
Unattended-Upgrade::Mail "sysadmin@example.com";
```

One thing to watch: `unattended-upgrades` will not reboot for you by default. If a kernel update lands, you'll need to handle the reboot separately - or explicitly enable `Automatic-Reboot` if that's acceptable in your environment.

DNF / YUM (RHEL, CentOS, Fedora, Rocky, Alma)

Red Hat-family distributions use errata - structured advisories that classify updates as security, bugfix, or enhancement. This makes it straightforward to filter for security-only patches.

```
# List available security updates
dnf updateinfo list --security

# Apply security updates only
dnf update --security -y

# Check which advisories affect your system
dnf updateinfo info --security
```

For automation, `dnf-automatic` replaces the older `yum-cron`:

```
# /etc/dnf/automatic.conf
[commands]
upgrade_type = security
apply_updates = yes
download_updates = yes
```

Enable it with:

```
systemctl enable --now dnf-automatic-install.timer
```

Note the distinction: you enable the timer, not the service directly. The default timer fires once daily, about an hour after boot. Check `/var/log/dnf.rpm.log` for what was installed, or use `journalctl -u dnf-automatic-install.service`.

APK (Alpine Linux)

Alpine is everywhere in containers. Its package manager is minimal and fast, which is the point.

```
# Update package index
apk update

# List available upgrades
apk version -l '<'

# Upgrade all packages
apk upgrade

# Fix a specific package
apk fix <package-name>
```

Alpine doesn't have a built-in equivalent of `unattended-upgrades` or `dnf-automatic`. If you're running Alpine on long-lived servers (rather than ephemeral containers), you'll need to set up your own automation - typically a cron job or an external tool.

Alpine also doesn't have the same errata/advisory system as RHEL. Security updates are tracked through the Alpine security database and typically addressed by updating to the latest package version in the repository.

Pacman (Arch Linux)

Arch follows a rolling-release model, which means there's no distinction between "security updates" and "regular updates." When you run `pacman -Syu`, you get everything.

```
# Full system upgrade (the only way to update on Arch)
pacman -Syu

# Check for available updates without installing
checkupdates
```

This rolling model makes Arch a poor fit for production servers where you need to selectively apply security patches without pulling in feature changes. If you're running Arch in production (and some teams do, for specific workloads), understand that you're accepting the full update stream every time you patch.

FreeBSD pkg

FreeBSD isn't Linux, but it shows up in enough mixed fleets that it's worth covering.

```
# Check for available updates
pkg update
pkg upgrade -n  # Dry run

# Apply updates
pkg upgrade -y

# Audit installed packages for known vulnerabilities
pkg audit -F
```

The `pkg audit` command is particularly useful - it checks installed packages against the FreeBSD VuXML vulnerability database and tells you exactly which packages have known issues. It's one of the better built-in vulnerability scanning tools across any of these systems.

FreeBSD also offers `freebsd-update` for base system patches:

```
# Fetch and install base system security patches
freebsd-update fetch
freebsd-update install
```

The Challenge of Mixed-Distro Environments

In theory, you pick one distro and standardize. In practice, your environment looks something like this:

- Production web servers on Ubuntu 22.04 LTS
- Database servers on RHEL 9 because that's what the vendor certifies
- Container hosts running whatever the base image uses (often Alpine or Debian slim)
- A few legacy CentOS 7 boxes that nobody wants to migrate
- That one Arch box the previous sysadmin set up for "testing" that somehow became production-critical
- FreeBSD on the storage servers because of ZFS

Each of these has different package managers, different repository structures, different advisory systems, and different commands. A script that works on Ubuntu will fail on RHEL. An Ansible playbook that uses `apt` needs conditional logic for `dnf` hosts.

This is where most ad-hoc patching approaches break down. It's not that any individual system is hard to patch - it's that maintaining consistent processes across a diverse fleet takes real effort.

The honest answer is: you either standardize your fleet (easier said than done), or you use tooling that abstracts over the differences. We'll cover both approaches.

Manual vs. Automated: The Progression

Most teams follow a similar evolution in how they handle patching. There's no shame in being at any stage - the question is whether your current approach scales with your responsibilities.

Stage 1: SSH and Run Commands

```
ssh server1 "apt update && apt upgrade -y"
ssh server2 "dnf update --security -y"
# ... repeat for every server
```

This works for 5 servers. It's painful at 20. It's untenable at 100. You'll miss servers, forget to check results, and have no audit trail.

Stage 2: Shell Scripts and SSH Loops

```
#!/bin/bash
for host in $(cat /path/to/hostlist.txt); do
  ssh "$host" "apt update && apt upgrade -y" 2>&1 | tee -a /var/log/patching.log
done
```

Better. You have a list of hosts and a log. But you still lack error handling, rollback capability, distro awareness, and any kind of approval workflow. And if a patch breaks something on server 47 of 200, you won't know until someone complains.

Stage 3: Cron + Package Manager Automation

Using `unattended-upgrades` on Debian/Ubuntu or `dnf-automatic` on RHEL-family systems, you can automate security updates on individual machines. This is a solid approach for security-only patches on non-critical systems.

The limitation is visibility. Each machine patches itself independently. You don't have a centralized view of what's been patched, what failed, or what's still pending. You're relying on each machine's local logs and hoping nothing went wrong.

Stage 4: Configuration Management (Ansible, Puppet, Chef, Salt)

```
# Ansible playbook for multi-distro patching
- hosts: all
  become: true
  tasks:
  - name: Update apt packages (Debian/Ubuntu)
    apt:
      upgrade: safe
      update_cache: yes
    when: ansible_os_family == "Debian"

  - name: Apply security updates (RHEL-family)
    dnf:
      name: "*"
      security: yes
      state: latest
    when: ansible_os_family == "RedHat"
```

Ansible (or similar tools) gives you centralized execution, distro-aware logic, idempotency, and better error reporting. For many teams, this is the right level of tooling. You write playbooks, run them on schedule or on demand, and review the output.

The gap: Ansible tells you what it did during the run. It doesn't give you an ongoing inventory of patch status, a dashboard of which servers are behind, or integration with your compliance reporting. You build those on top, or you look at dedicated patch management tools.

Stage 5: Dedicated Patch Management Platforms

Purpose-built tools that provide inventory, scanning, approval workflows, staged rollouts, compliance reporting, and cross-distro support. We'll cover the options in the tooling section below.

Security Updates vs. Feature Updates

This distinction is critical, and it's one that junior admins sometimes miss.

Security updates fix known vulnerabilities. They're typically small, well-tested, and low-risk. Distributions backport security fixes to maintain ABI compatibility - a security update to OpenSSL on Ubuntu 22.04 patches the vulnerability without changing the OpenSSL version or behavior. The goal is to fix the hole without changing anything else.

Feature updates add new functionality, change behavior, or upgrade to new upstream versions. They're more likely to introduce breaking changes, require configuration adjustments, or affect dependent applications.

Why this matters for patch management:

Security updates should be applied quickly. The risk of not patching almost always outweighs the risk of the patch itself. For critical CVEs, "quickly" means hours or days, not weeks.

Feature updates deserve more testing. They can wait for a maintenance window, go through staging environments, and get proper validation before hitting production.

Most package managers and patching tools let you distinguish between the two. Use that capability. Treating all updates the same - either rushing everything or delaying everything - is a mistake in both directions.

One nuance: the line isn't always clean. Sometimes a security fix requires a minor version bump that also includes non-security changes. Distributions try to avoid this, but it happens. Read the changelogs.

Building a Patch Management Workflow

A repeatable workflow keeps you from making decisions under pressure. Here's a framework that works across team sizes.

1. Inventory

You can't patch what you don't know about. Maintain a current inventory of:

- All servers and their operating systems (distro, version, architecture)
- Installed packages and their versions
- Which services run on which hosts
- Ownership - who's responsible for each system

This doesn't need to be fancy. A CMDB is nice. A well-maintained spreadsheet works. An automated inventory tool is best. But you need *something*.

```
# Quick inventory helpers
# Debian/Ubuntu
dpkg -l | wc -l
lsb_release -a

# RHEL-family
rpm -qa | wc -l
cat /etc/redhat-release

# Any Linux
uname -r
hostnamectl
```

2. Assess

When new patches are available, assess them before applying:

Severity: Is this a CVSS 9.8 remote code execution, or a CVSS 3.1 local-only info disclosure?

Applicability: Does this affect packages you actually have installed and services you actually run?

Urgency: Is there active exploitation in the wild? Check CISA's Known Exploited Vulnerabilities (KEV) catalog.

Dependencies: Will this patch require a reboot? Will it restart a critical service?

3. Test

For anything beyond routine security patches, test in a non-production environment first.

If you have staging environments, patch those first and run your test suite.

If you don't, pick your least critical production servers as an initial cohort.

Wait at least 24-48 hours before rolling out to the full fleet. Most patch-related issues surface in the first day.

4. Approve

Even in small teams, someone should explicitly approve production patching. This can be as lightweight as a message in a Slack channel: "Applying March security patches to prod web tier, here's the list." The point is a human reviewed what's going in and said "go."

For regulated environments, you'll need formal change requests with documented approvals.

5. Deploy

Roll out in stages:

Canary group (5-10% of servers): Apply patches, monitor for 1-4 hours.

Early adopters (25-50%): If canary is healthy, expand.

Full fleet: Complete the rollout.

If something breaks in the canary group, you've limited the blast radius. This is the entire point of staged rollouts.

6. Verify

After deployment, confirm patches were actually applied:

```
# Verify a specific package version (Debian/Ubuntu)
dpkg -l | grep openssl

# Verify a specific package version (RHEL-family)
rpm -q openssl

# Check if a reboot is needed (Debian/Ubuntu)
cat /var/run/reboot-required 2>/dev/null && echo "Reboot needed" || echo "No
reboot needed"

# Check if a reboot is needed (RHEL-family)
needs-restarting -r
```

Don't just trust that your deployment tool reported success. Verify independently that the expected versions are installed and services are running correctly.

7. Document

Record what was patched, when, on which systems, and by whom. This serves three purposes:

Troubleshooting: When something breaks next Tuesday, you can check if it correlates with patches applied last Friday.

Compliance: Auditors will ask for patch records. Having them ready saves weeks of scrambling.

Institutional knowledge: When you're on vacation and your colleague needs to know why server X is running a different version than server Y, the record explains it.

Compliance Requirements

If your organization handles sensitive data or operates in a regulated industry, patch management isn't optional - it's auditable.

PCI-DSS

PCI-DSS Requirement 6.3.3 (as updated in version 4.0.1, effective 31 December 2024) is explicit: install patches for critical and high-severity vulnerabilities within one month of release, with other applicable patches installed within an appropriate timeframe as defined by the entity. Most Qualified Security Assessors (QSAs) expect faster response times in practice for actively-exploited CVEs, but the 30-day window is the documented baseline.

You need to demonstrate:

A defined process for identifying available patches

Risk-ranked assessment of patches

- Timely installation with documented evidence
- A way to handle exceptions when a patch can't be applied immediately

ISO 27001

ISO 27001 control A.8.8 (Technical Vulnerability Management) requires organizations to establish a process for identifying and managing vulnerabilities in a "timely manner." Unlike PCI-DSS, ISO 27001 doesn't prescribe specific timelines - your organization defines what "timely" means based on your risk assessment, and then you're held to your own standard.

What auditors look for:

- A documented vulnerability management policy
- Evidence that you follow your own policy consistently
- Risk-based prioritization of patches
- Records of patching activity

SOC 2

SOC 2 (specifically the Common Criteria related to risk mitigation and system operations) requires evidence that systems are kept up to date. Auditors typically want to see:

- A patch management policy
- Proof that patches are applied within your stated SLA
- Historical patch records showing consistent application
- Visibility into current patch status across your environment

The Common Thread

All three frameworks want the same basic things: a documented process, evidence you follow it, risk-based prioritization, and records. The tooling you use matters less than whether you can demonstrate consistent, timely patching with an audit trail.

The Tooling Landscape

There's no shortage of tools for Linux patch management. The right choice depends on your fleet size, distro mix, existing tooling, and budget.

Ansible Playbooks

Best for: Teams already using Ansible, mixed-distro environments, those who want full control.

Ansible can handle patching across any Linux distribution (and FreeBSD) using its built-in modules - `apt`, `dnf`, `yum`, `apk`, `pacman`, `pkgng`. You write the playbooks, define the inventory, and control the execution.

Strengths: Agentless (SSH-based), infinitely customizable, handles any distro, free. You own the entire workflow.

Limitations: No built-in patch status dashboard. No built-in approval workflow. No vulnerability scanning. You're building these capabilities yourself or integrating with other tools. Reporting for compliance requires additional work.

Red Hat Satellite

Best for: Large RHEL-centric environments with Red Hat subscriptions.

Satellite (based on the open-source Foreman + Katello projects) provides content management, lifecycle environments (dev/QA/prod), errata management, and integration with Ansible for execution. It's the recommended solution for RHEL fleet management.

Strengths: Deep RHEL integration, content views for controlled promotion, errata classification, compliance reporting, subscription management.

Limitations: Commercial product (requires Red Hat subscription). Primarily designed for RHEL-family. Heavy infrastructure requirements - Satellite itself needs dedicated hardware and a Postgres database. Steep learning curve.

Foreman + Katello

Best for: Teams that want Satellite-like capabilities without the Red Hat subscription cost.

Foreman is the open-source upstream of Red Hat Satellite. With the Katello plugin, you get content management and lifecycle environments. Supports RHEL-family, Debian/Ubuntu, and SUSE.

Strengths: Free. Multi-distro content management. Lifecycle promotion. Active community.

Limitations: Significant setup and maintenance burden. Less polish than the commercial version. Documentation can be inconsistent. You're your own support team.

SUSE Manager

Best for: SUSE-centric environments or organizations standardized on SUSE products.

Built on the Uyuni open-source project (formerly Spacewalk). Provides patch management, compliance auditing (OpenSCAP integration), and configuration management.

Strengths: Strong SUSE/openSUSE support. Compliance scanning built in. Can manage RHEL and Ubuntu hosts too.

Limitations: Best experience is with SUSE products. Smaller community than Foreman. Commercial product for full support.

Automox

Best for: Cloud-first teams managing Linux alongside Windows and macOS.

SaaS-based endpoint management with cross-platform support. Agent-based, with a web console for policy management.

Strengths: Easy setup, SaaS model means no infrastructure to manage, cross-OS support, good for distributed fleets.

Limitations: Per-device pricing can get expensive at scale. Less depth for Linux-specific needs than dedicated Linux tools. Requires outbound connectivity from every agent.

ManageEngine Patch Manager Plus

Best for: Organizations already in the ManageEngine/Zoho ecosystem.

Supports Linux (multiple distros), Windows, and macOS. On-premises or cloud deployment. Automates patch scanning, testing, and deployment.

Strengths: Wide distro support, automated patch testing, decent reporting for compliance.

Limitations: UI can feel dated. Some advanced features are enterprise-tier only. Linux support, while present, historically plays second fiddle to Windows capabilities.

PatchMon

Full disclosure: you are reading the PatchMon blog, so take this section with the appropriate grain of salt. We will try to be honest anyway.

Best for: Teams managing mixed Linux/FreeBSD fleets who want real-time visibility without heavyweight infrastructure.

PatchMon provides agent-based scanning across multiple Linux distributions and FreeBSD, with a centralized dashboard showing patch status, pending updates, and compliance posture. Single Go binary, Docker Compose deployment, five minutes to first dashboard. Available two ways: PatchMon Cloud is the managed SaaS, billed per host that actually checks in, with a 14-day trial up front. The AGPLv3 Community Edition is free to self-host on infrastructure you manage.

Strengths: Built for Linux and FreeBSD from the ground up. Lightweight agent. Centralized visibility across mixed distros. Compliance scanning with OpenSCAP, CIS, and Docker Bench (Max tier). Built-in SSH and RDP remote access (Max tier). Dry-run support on all tiers; approval workflows on Plus and above. Cloud is per-host with no fleet minimum; self-hosted Community has no PatchMon per-endpoint licence fee at all.

Limitations: Newer entrant compared to established tools like Satellite or Foreman. Smaller community. Does not handle provisioning or full lifecycle management; it is focused on patching, compliance, and visibility.

Which Tool Should You Use?

Honestly, it depends - but we will tell you what we would do:

> **Small fleet, single distro:** `unattended-upgrades` or `dnf-automatic` gets you started. But even here, you have no dashboard, no audit trail, and no alerting when updates fail silently. PatchMon adds that visibility with minimal overhead.
>
> **Medium fleet, mixed distros:** This is where Ansible playbooks start becoming a second job. You are writing YAML to patch servers, then writing more YAML to report on patching, then debugging why the Arch box broke the playbook. PatchMon handles the cross-distro complexity natively: one agent per host, one dashboard for everything.
>
> **Large RHEL-only fleet:** Red Hat Satellite is the path of least resistance if you are already paying for RHEL subscriptions. But if your fleet is not 100% RHEL (and it rarely is), you are back to multiple tools.
>
> **Large mixed fleet:** PatchMon was built specifically for this scenario. APT, DNF, YUM, APK, Pacman, and FreeBSD pkg with full patch deployment, plus Windows Update monitoring (Windows patch deployment is on the roadmap), in a single dashboard with unified policies.
>
> **Compliance-heavy environment:** You need an audit trail, compliance scanning, and automated reporting. PatchMon includes OpenSCAP, CIS, and Docker Bench scanning on the Max tier and generates the reports your auditor will ask for. Satellite and ManageEngine do this too, but at significantly higher cost and complexity.

We are obviously biased (you are reading this on the PatchMon blog). But here is our honest case: the easiest way to try it is PatchMon Cloud (https://patchmon.net/pricing). You get a managed control plane, billing that scales with the hosts that actually check in, and a 14-day trial to see whether the thing is any good before your card is charged. If you'd rather keep everything on your own infrastructure, PatchMon Community is AGPLv3 with no per-endpoint fee. Either way you can decide within an hour of first login whether it solves your problem.

Best Practices Checklist

These aren't aspirational ideals. They're the minimum for a defensible patching process.

Process:

- [ ] Maintain an accurate inventory of all systems and their OS versions
- [ ] Define SLAs for patch application by severity (e.g., critical within 72 hours, high within 7 days, medium within 30 days)
- [ ] Separate security updates from feature updates in your workflow

- [ ] Test patches in a non-production environment before full rollout (or at minimum, use canary deployments)
- [ ] Use staged rollouts - never patch the entire fleet simultaneously
- [ ] Verify patches were applied successfully, don't just trust the tool's output
- [ ] Document every patching cycle: what was applied, when, where, by whom

Technical:

- [ ] Configure automatic security updates on all systems as a baseline (`unattended-upgrades`, `dnf-automatic`)
- [ ] Monitor for reboot-required status after kernel updates
- [ ] Subscribe to your distributions' security mailing lists (USN for Ubuntu, RHSA for Red Hat, DSA for Debian)
- [ ] Monitor CISA's Known Exploited Vulnerabilities catalog for urgently-exploited CVEs
- [ ] Use `needrestart` (Debian/Ubuntu) or `needs-restarting` (RHEL) to identify services that need restarting after library updates
- [ ] Pin critical application dependencies when needed, but review pins regularly - a pinned package is an unpatched package

Organizational:

- [ ] Assign clear ownership for patching - if everyone is responsible, nobody is
- [ ] Include patching metrics in your regular security reviews
- [ ] Plan for emergency patching: when a critical CVE drops on a Friday afternoon, who's authorized to push patches without a full change process?
- [ ] Conduct periodic audits: compare actual patch status against your policy SLAs
- [ ] Budget time for patching - it's not overhead, it's operations

Conclusion

Patch management on Linux isn't a problem you solve once. It's a discipline you maintain. The tools and automation help, but they don't replace understanding your systems, making deliberate decisions about risk, and building processes that work when things get messy - which they will.

Start where you are. If you are SSH-ing into servers manually, set up `unattended-upgrades` or `dnf-automatic` today as a safety net. But if you manage more than a handful of hosts - especially across multiple distributions - you need actual visibility, not just automation running in the dark.

That is what we built PatchMon to do. One dashboard showing every host, every package, every pending security update across your entire fleet. Dry-run before you commit. Approval workflows for production (Plus tier and above). Compliance scanning with OpenSCAP, CIS,

and Docker Bench (Max tier). An audit trail that satisfies your auditor. Browser-based SSH and RDP when you need to jump in and fix something (Max tier). Start with PatchMon Cloud (https://patchmon.net/pricing) if you want it managed for you, or grab the Community Edition (https://patchmon.net/open-source) if you'd rather keep it on your own kit.

The worst patch management strategy is the one you abandon because it is too painful to follow. The second worst is having no strategy at all. Build something sustainable, automate the tedious parts, and focus your human attention on the decisions that actually require judgment.

Your future self - the one who gets paged about the next critical CVE - will thank you for the process you build today.

PatchMon

ABOUT PATCHMON

The open source Linux patch management platform

PatchMon gives sysadmins one dashboard for patching across Linux, FreeBSD, and Windows fleets. Run it as a managed SaaS on PatchMon Cloud (per-host billing, 14-day trial, no fleet minimum) or self-host the AGPLv3 Community Edition on your own infrastructure.

Start a trial: patchmon.net/pricing