OTA Updates in the Wild: Lessons from the Field

Jana Perić, Northern.tech

September 2025.
NDC TechTown

$whoami

Jana Perić

jana.peric@northern.tech

Embedded Linux Customer Engineer for Mender at Northern.tech

Jana is an Embedded Linux Engineer at Northern.tech, where she helps customers succeed (and occasionally survive) with Mender — the open-source OTA update solution. With hands-on experience shipping firmware updates into the wild, Jana has seen everything from flawless rollouts to devices that went mysteriously silent mid-update. She specializes in turning complex OTA setups into something humans can actually maintain, and she secretly enjoys debugging broken bootloaders more than she probably should. When not neck-deep in Yocto layers or U-Boot environment puzzles, she’s usually trying to explain to her friends what “firmware” actually means.

Agenda

1. Why OTA matters?
2. What makes OTA hard?
3. OTA update modules & architectures
4. Tooling landscape
5. Best practices and takeaways
6. Q&A

SECTION 1

Why Over-The-Air (OTA) updates matter?

What happens after your product leaves the factory — and why it is still your problem.

What are OTA updates, really?

OTA (Over-the-Air) updates are a method of remotely delivering and installing software updates to connected devices, such as smartphones, vehicles, and IoT devices, **without the need for a physical interaction**.

Can update:

- Full system images (system-level)
- Individual apps or services (app-level)
- Configs, certificates, security keys

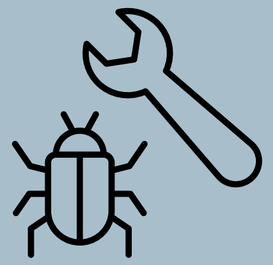
Applies to:

- Embedded Linux devices (gateways, appliances, sensors)
- IoT devices (industrial, consumer, medical, etc.)
- Essentially - every device with Internet connectivity

What's at stake?

Security

Patch CVEs, rotate keys, respond to zero-days



Bug fixes

Resolve crashes, regressions, logic errors

Feature updates

Add capabilities after deployment

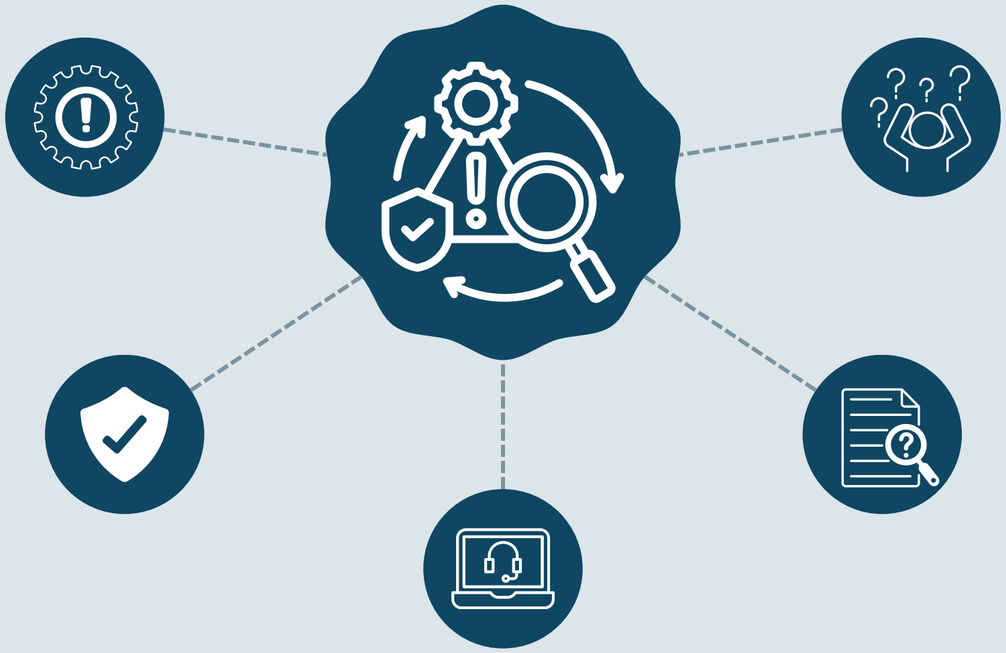
Cost savings

Avoid expensive on-site service calls, shipping costs, or downtime

The risks of not having OTA updates

Bricked Devices

Device becomes dysfunctional & unresponsive



Brand Damage

Users notice when devices fail silently - trust is broken

Security Exposure

Devices stuck with known vulnerabilities

Lost Opportunities

Can’t iterate, improve, or respond to user needs

Support Load

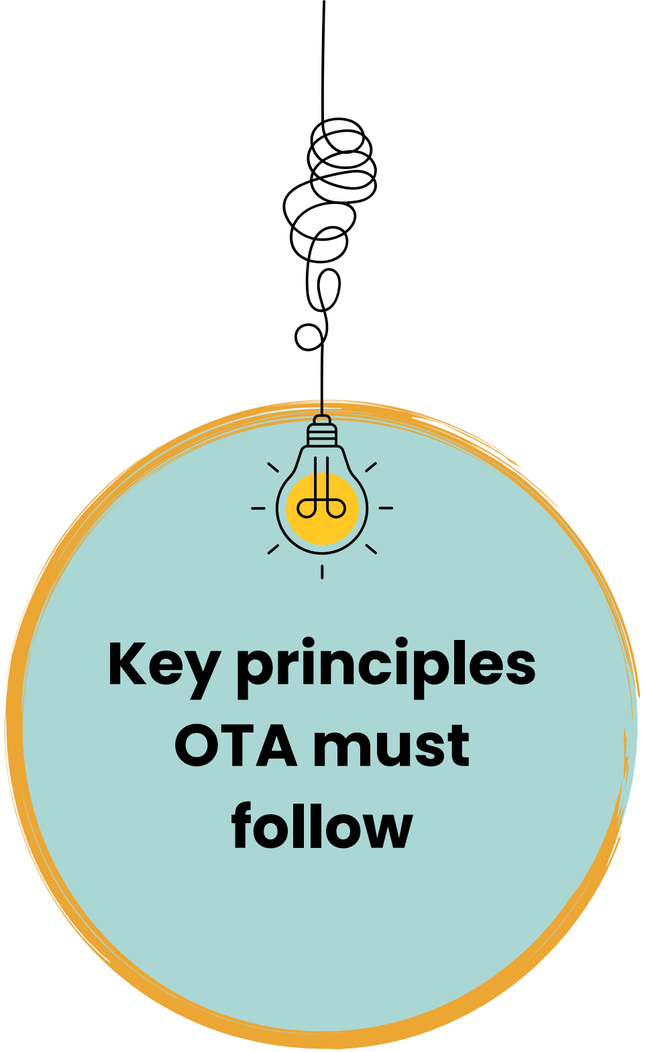
Manual intervention, increased RMA, user frustration

SECTION 2

What makes OTA hard?

It sounds simple: send update → install it on the device → everyone’s happy.

In reality… not so simple.



Key principles OTA must follow

INTERRUPTION RESILIENT

→ What happens if power or connectivity are lost mid-update?
→ You must detect partial installs and recover cleanly.

ATOMIC

→ Updates must fully succeed — or roll back cleanly.
→ No “half-installed” firmware allowed.

RETRY SAFE

→ Failed updates should resume or safely retry without corrupting the system.
→ Avoid "stuck in update mode" states.

SECURE

→ Updates should be cryptographically signed and verified.
→ Secrets (e.g. API keys, credentials) must be protected during and after updates.

Unreliable network = Unreliable updates?

- **Devices may be:**
  - Offline for hours/days/weeks/months...
  - Behind NATs, firewalls, captive portals
  - On metered or intermittent LTE
- **What can go wrong:**
  - Partial downloads
  - Timeout edge cases
- **What you need:**
  - Resumable downloads, retries
  - Connectivity checks before and after update
  - Handling for different connectivity scenarios

Storage constraints

Many devices have just one flash chip and a few hundred MBs (and sometimes even less) to work with. You may need space for:

- The current system
- The new update payload
- A rollback image or recovery partition

Compression, streaming, and delta updates can help, but require **careful planning.**

Signing and security

Updates **should** be:

- **Signed** (to prevent tampering)
- **Verified** (to detect corruption)
- **Authenticated** (only accepted from trusted sources)

You also need to protect:

- Secrets and credentials on the device
- Firmware integrity during and after install

🚨 **A broken or bypassed update mechanism can become a backdoor into every deployed device.**

Safe rollback & atomicity

What happens if something goes wrong mid-update?

Without rollback, a failed update can brick your device.

Atomicity means either:

- The entire update succeeds
- Or the system safely rolls back without leaving things half-broken

and **nothing else**.

Solutions like A/B partitioning (Mender-style) solve this — but cost extra flash and complexity.

Observability & feedback

How do you know if the update succeeded?
Did the device even attempt it?
Did it boot into the new version?
Did it roll back silently?

Logging and reporting are critical for debugging, especially at scale.

Versioning & compatibility

What if a device missed three updates and now tries to jump ahead?

Is it safe to apply v5 over v2?

How do you manage:

- Dependency changes (e.g. config, storage formats)
- Application data migrations
- Rollback logic

Testing

You’re not just testing software, you’re testing:

- Storage limits,
- Power cycles,
- Bootloaders,
- Reboots and retries.

Unit tests aren’t enough — you need end-to-end, failure-mode testing.

Summary:

Why OTA is hard (but worth doing right)

1. **Unreliable connectivity** makes delivery tricky and unpredictable
2. **Limited storage** forces tradeoffs between safety, speed, and size
3. **Security** is non-negotiable — but easy to misconfigure or forget
4. **Rollback and failure recovery** are mandatory, not optional
5. **Testing** is complex — updates must work across hardware, versions, and real-world mess

It’s not just about delivering updates — it’s about doing it safely, at scale, without breaking what’s already working.

SECTION 3

OTA update modules and architectures

There are many ways to update a device — some more reliable than others.

Different OTA update models

Not all OTA systems are built the same.

The right architecture depends on your constraints. Core considerations are **storage**, **safety, recovery needs**, and **update frequency.**

Let’s look at the most common patterns and tradeoffs.

A/B dual root filesystem

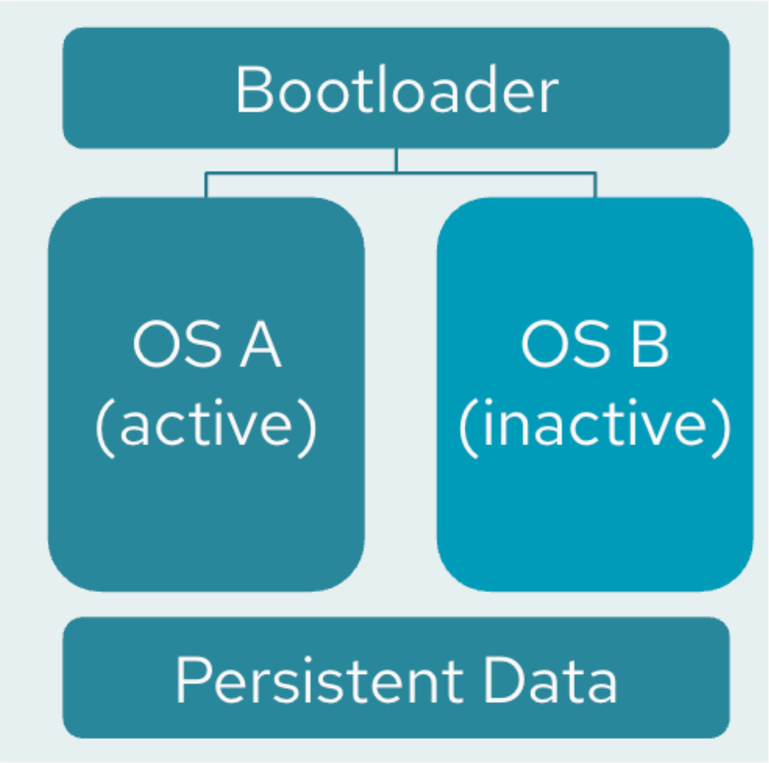
A/B system uses **two separate root filesystems**:

- **active** = currently running
- **inactive** = standby, used for the next update

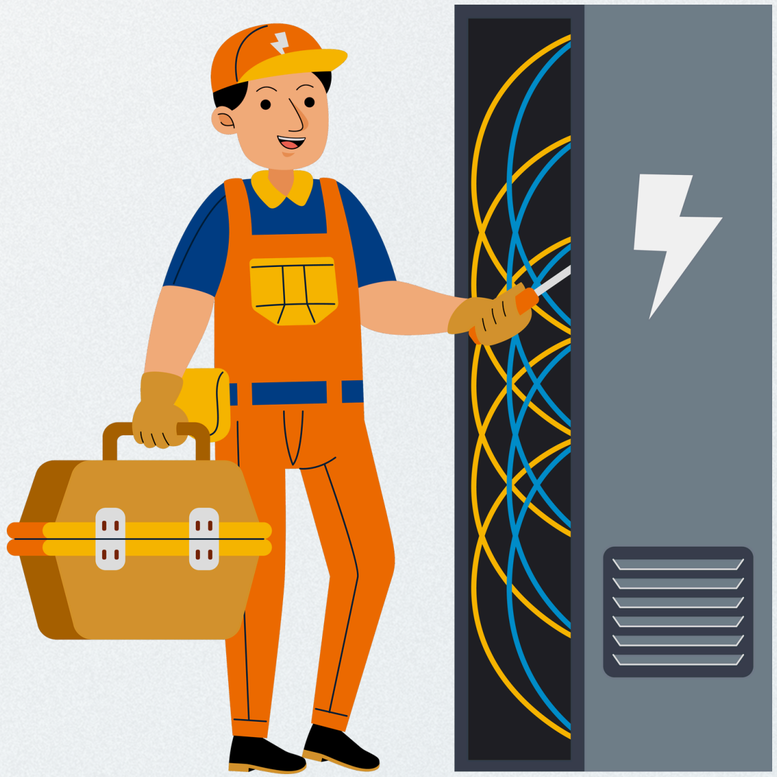
The update is written to the inactive partition (B) while the system runs from active partition (A).

On reboot, the bootloader switches to B (if valid).

If B fails to boot or passes a timeout threshold, it rolls back to A.



Why bother with A/B?



Safe rollback

If an update fails, system reverts to known-good version

Atomic updates

System is either fully updated or not at all

Minimal downtime

Devices are operational while the update is being installed

Real-world proven

Used in ChromeOS, Android, Mender, and other production systems

OTA without A/B: The recovery partition approach

- Device has **one root filesystem** (where updates are applied directly)
- A separate recovery partition contains a minimal, stable OS image
- If an update to the main rootfs fails or the device becomes unbootable:
  - The bootloader can fall back to the recovery image
  - The recovery system can re-install a known-good image or request help from a remote server

🚨 **This approach imposes downtime for the device - especially if something goes wrong mid-update.**

Application-level or System-level updates

Application-Level Update

- Update only part of the system, typically user-space apps or services.
- Delivered as:
  - Containers (e.g., Docker)
  - Files dropped into /opt or mounted volumes
  - Packages (.deb, .rpm, etc.)
- Often applied without rebooting the device.

System-Level Updates

- Replace the entire root filesystem or major system components.
- Usually involves:
  - Rootfs A/B swaps
  - Immutable image replacement
  - Reboot into new system

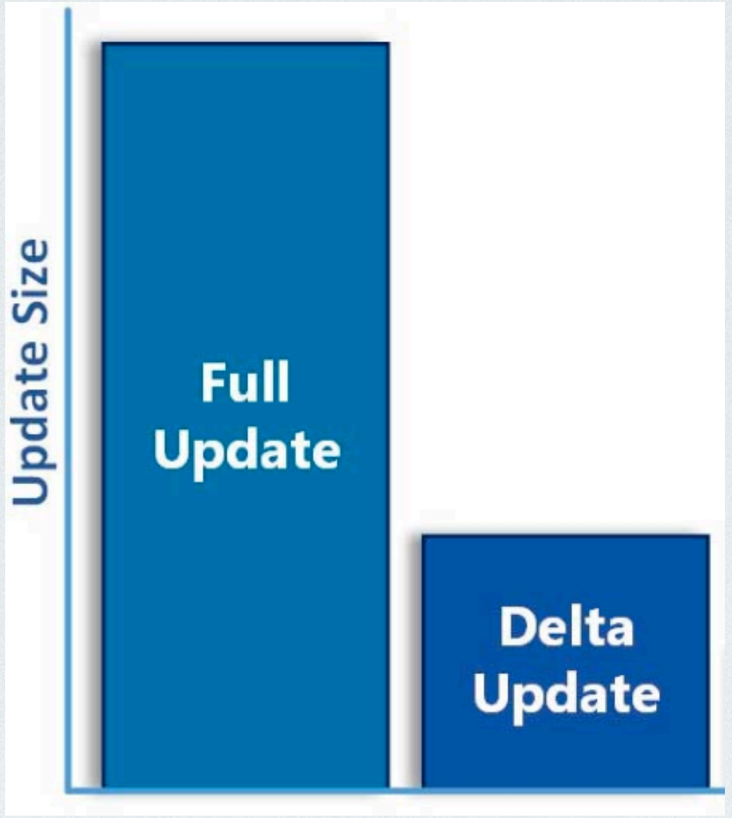
Delta or Full Image: How much are you sending?

Full image

- Sends a complete root filesystem or firmware image, replacing the existing one.
- Simple to implement, easier to test.

Delta

- Sends only the difference (delta) between the current version and the new one.
- The device patches the old version into the new version locally.



Update triggers: automatic, manual, phased rollouts

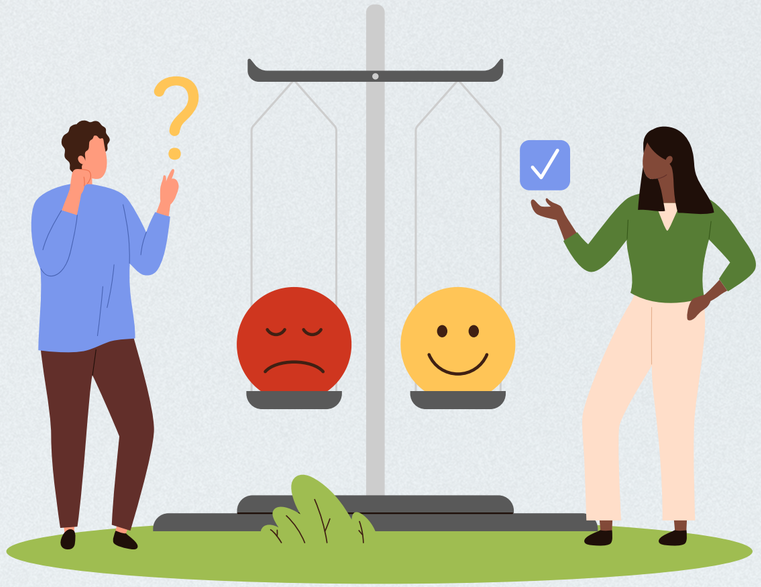
Manual updates

Updates are initiated explicitly by a user or admin (via UI, SSH, button press).
Common in:

- Development and staging
- Safety-critical systems requiring human oversight

CONS:

- Doesn’t scale
- Error-prone (forgotten updates, human error)



PROS:

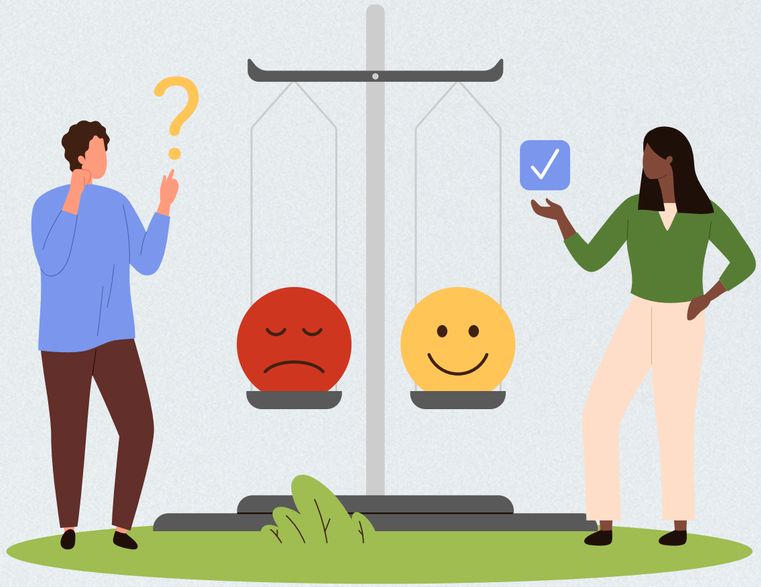
- Maximum control
- Good for diagnostics and testing

Automatic updates

- Devices periodically check for updates and apply them on their own.
- Can be time-based (e.g., every 6h) or event-based (e.g., on boot, battery charge).
- Common in:
  - Consumer devices (TVs, phones)
  - Unattended field devices

CONS:

- Risky if update breaks something
- Requires robust rollback and validation



PROS:

- Hands-off, scalable
- Ensures fleet stays up-to-date

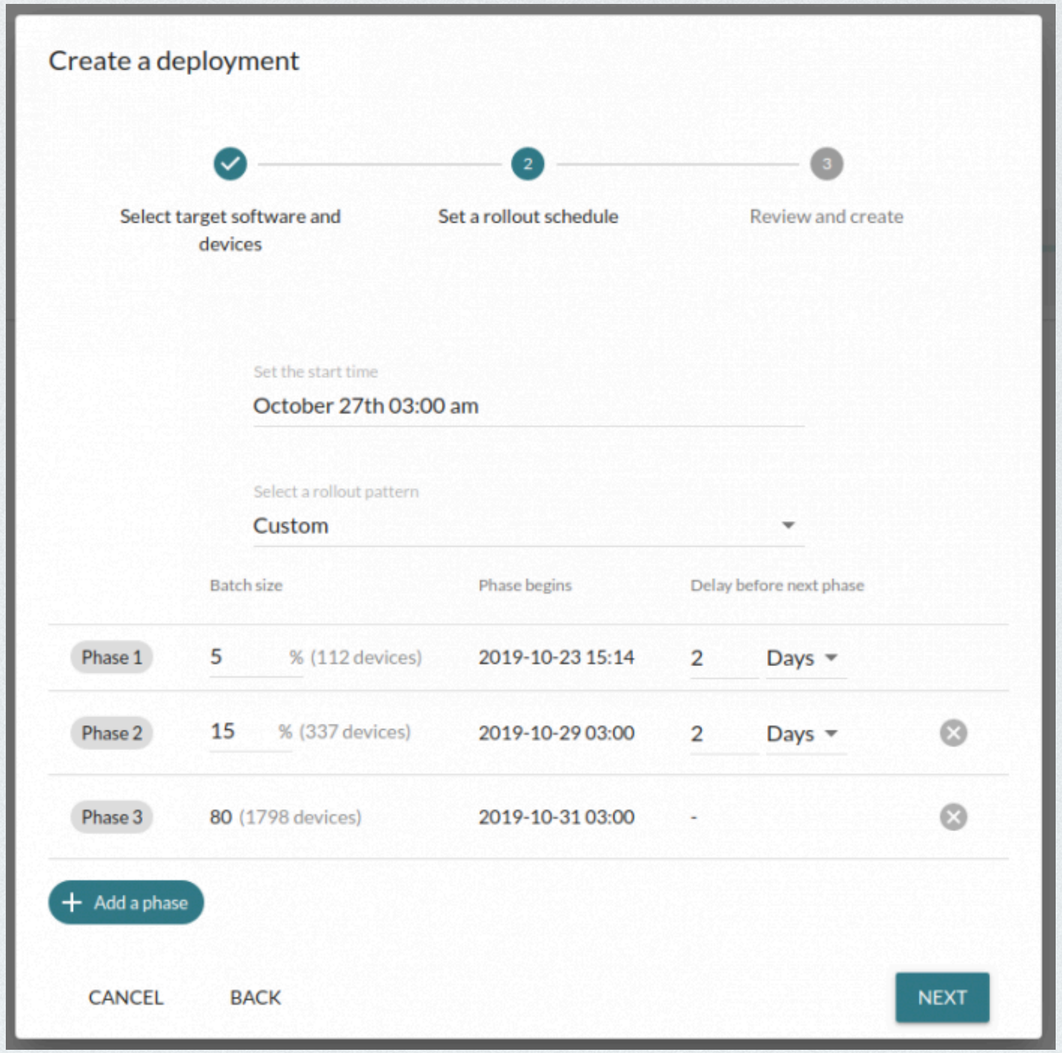
Phased rollout

What is it?

- Update is released to a small percentage of the fleet first.
- If no issues are reported, rollout continues to more devices.
- Also called canary deployments or staged rollouts.

Why it matters:

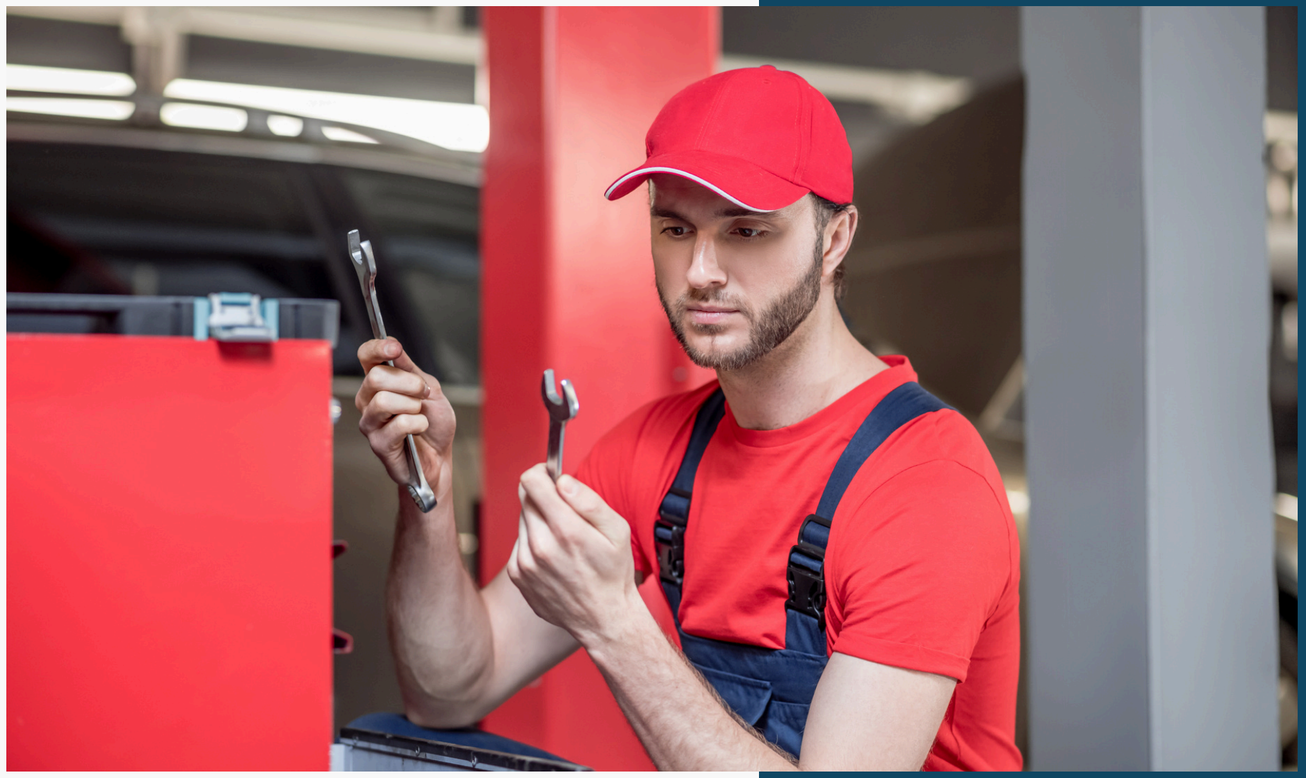
- Prevents bricking all devices at once with a bad update
- Gives you time to catch regressions or edge-case failures

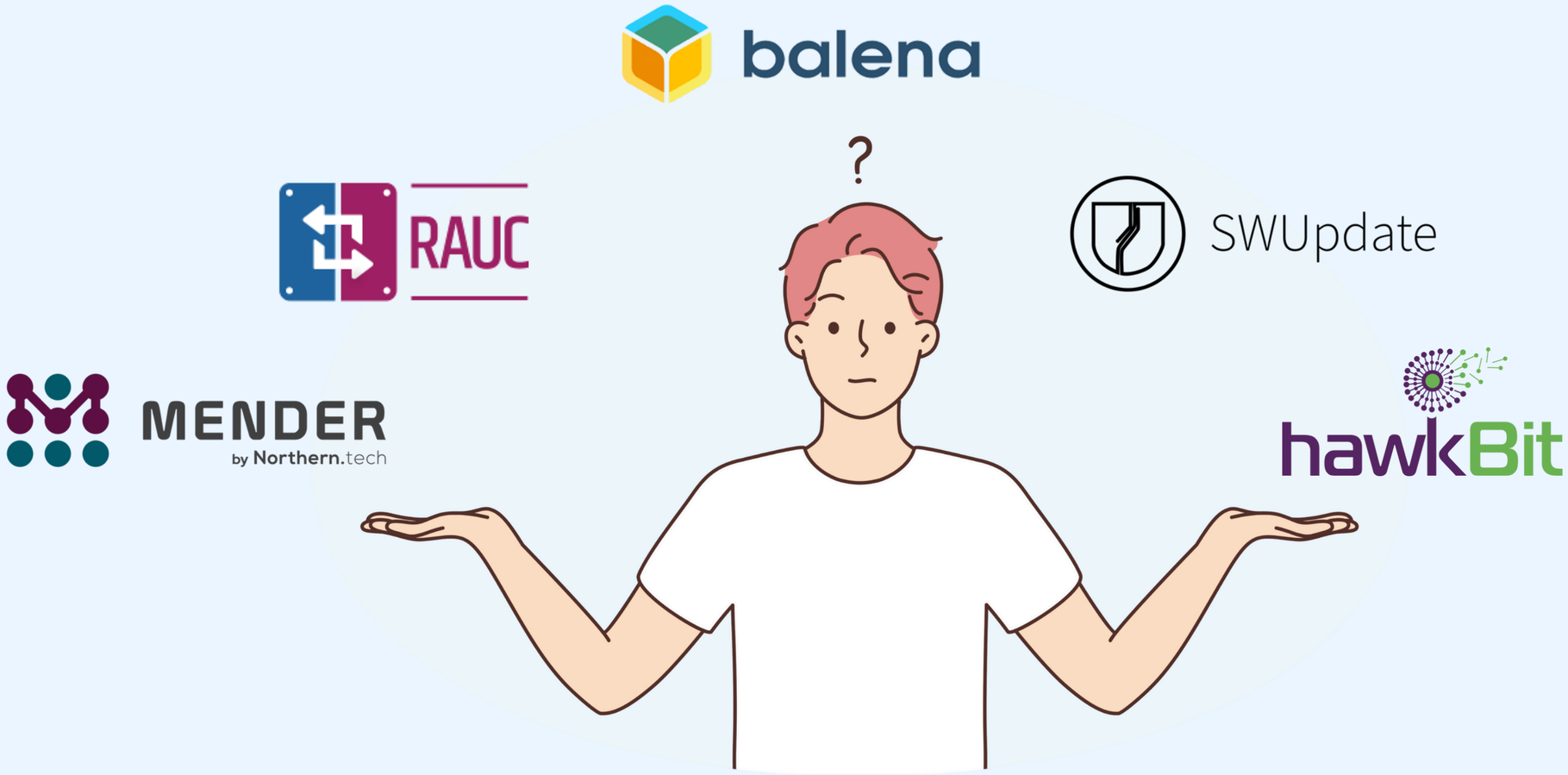


SECTION 4

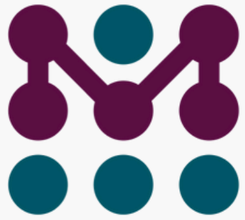
Tooling landscape

Which tool fits your system?





Choose your fighter: OTA edition



MENDER

by Northern.tech

- Open source with enterprise features
- Includes both server and client side components
- Focus on robust A/B updates and device fleet management
- Works with Linux and Zephyr, example integrations available for Yocto and Debian-style distributions
- Hosted server or self-hosted
- Supports multiple update types: full images, containers or custom
- Supported bootloaders: U-Boot, GRUB and custom

Good for: Production-grade systems needing reliability, rollback, and scaling.

SWUpdate

- Self-contained device-only solution
- Modular framework: you build your own update flow
- Supports multiple update types: full images, partitions, files
- Requires manual integration and bootloader configuration
- Supports recovery partition AND A/B scheme
- Supported bootloaders: U-Boot, GRUB and EFI Boot Guard
- HawkBit-compatible backend

Good for: Advanced users with custom needs and existing infrastructure.

- OTA management server (web UI + API)
- Meant to integrate with clients like SWUpdate, RAUC or custom agents
- Provides orchestration layer, not the update mechanism itself
- **Not actively maintained**

Good for: Enterprises building a custom OTA stack with server-side logic.

- Bundle-based approach; good Yocto integration
- Bootloader must support boot slot handling
- Simple and lightweight, but no built-in orchestration
- HawkBit-compatible backend

Good for: Teams comfortable building their own deployment infra.

- Container-based OTA platform (Docker-style)
- Designed for fast deployment of app changes
- Full fleet management
- Defined platform (Yocto-based)

Good for: IoT applications with containerized apps, not full system updates.

Which one should you choose?

Decision Factors:

- How much storage do you have?
- Do you need rollback support?
- What kind of license do you need?
- Are you updating the OS, the app, or both?
- Is customer support an important factor?
- How much money are you willing to spend on this?
- Do you need full device management or just OTA?
- How much customizations you want to make to the update flow?

🛠️ Popular OTA Solutions:

Tool	Type	Open Source	A/B Support	Delta Updates	Application Updates	Device Management
Mender	Open Core	✅ Yes	✅ Yes	✅ (opt-in)	✅ (Update Modules)	✅ Yes
SWUpdate	Framework	✅ Yes	✅ Optional	✅ (via plugins)	✅ (custom logic)	❌ No (DIY)
RAUC	Framework	✅ Yes	✅ Yes	⚠️ (via external)	✅ (bundles)	❌ No
Balena	Cloud Platform	❌ No	❌ No	✅ (containers)	✅ (Docker images)	✅ Yes

SECTION 5

Best Practices and Takeaways

The OTA survival kit

- **Rollback early** → Failures will happen, design safety nets upfront
- **Version everything** → Images, configs, artifacts must be reproducible
- **Monitor updates** → Track success/failure, rollbacks, anomalies
- **Make updates boring** → Predictable, low-drama releases are the goal
- **OTA is a journey** → Start simple, iterate, improve over time
- **Trust is the product** → Firmware updates carry user confidence

SECTION 6

Q&A

Thank you