

Release Engineering

Chris Timperley · Wednesday, September 16, 2026

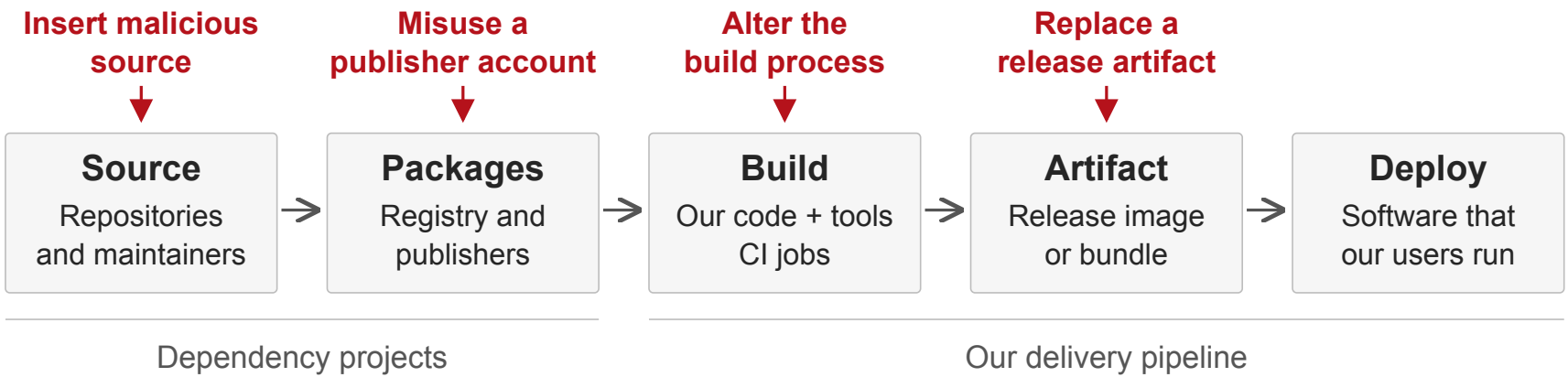
Supply-Chain Trust

Our software comes through a supply chain



- **Software supply chain:** the people and systems that produce and deliver software
- Each dependency brings its own supply chain

Attackers can target any stage of the chain



- A **supply-chain attack** compromises this chain to introduce malicious behavior

Why not trust packages we've used for years?



A trusted dependency can become malicious

event-stream incident · November 2018



New dependency → targeted development environment → changed application

- A new maintainer introduced a malicious dependency.
- It targeted Copay builds to steal wallet information and private keys.

Original diagram based on npm's incident account.

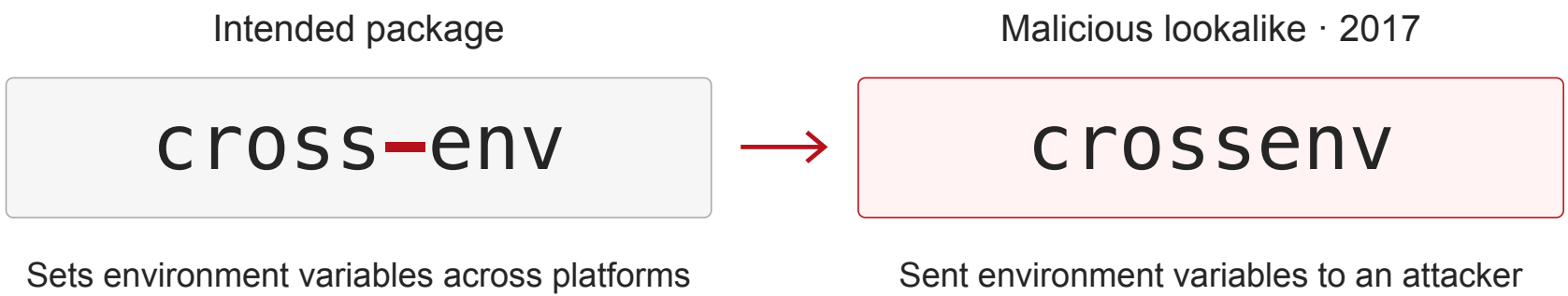
Passing checks does not establish trust

Check	What it tells us
Integrity hash	Downloaded bytes match the expected bytes.
Functional tests	Tested behavior works in the tested environment.
Vulnerability scan	Whether covered packages match known reports.

- A locked malicious version can pass an integrity check
- Targeted malicious behavior can escape tests and scans

Similar names can hide a different package

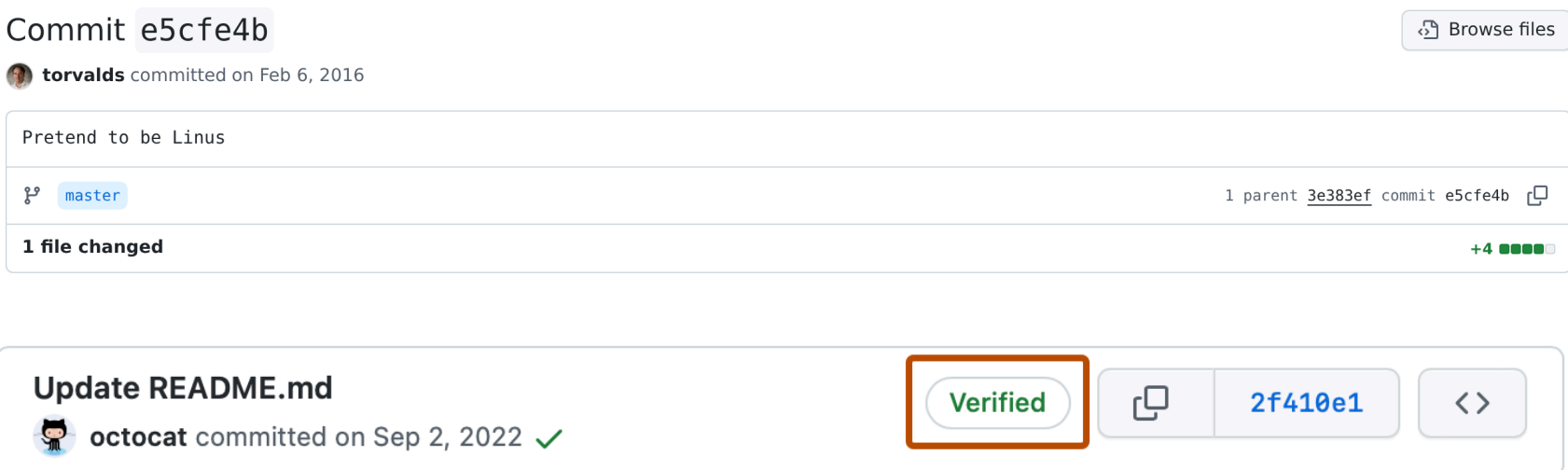
- **Check the exact name, repository, and publisher in the registry.**
- **Typosquatting** uses similar names to attract mistaken installs.



npm's 2017 incident report · Original illustration of the package-name difference.

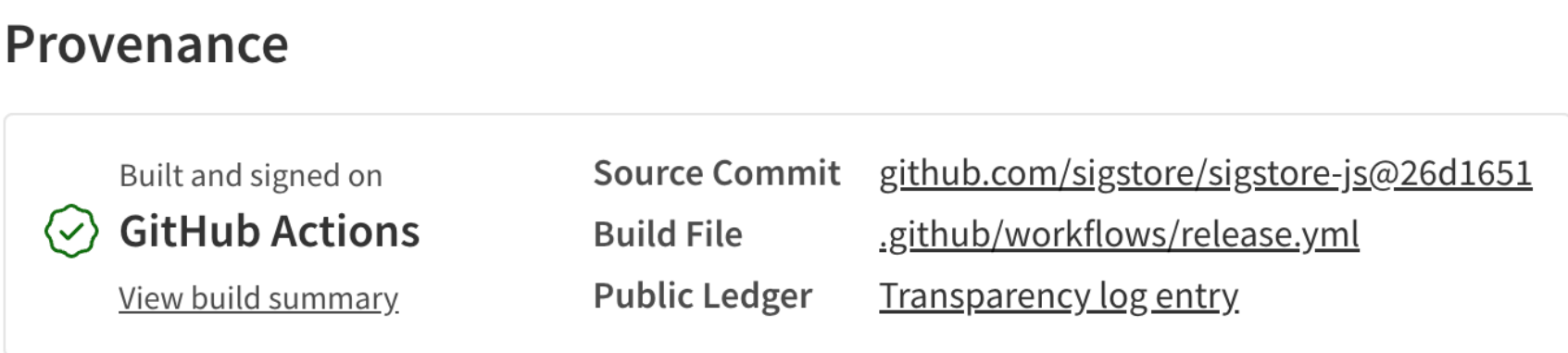
An author name is easy to fake

- **Git author names and emails are self-reported.**
- Inspect the signature and expected signer; still review the code.



Connect the package to its source and build

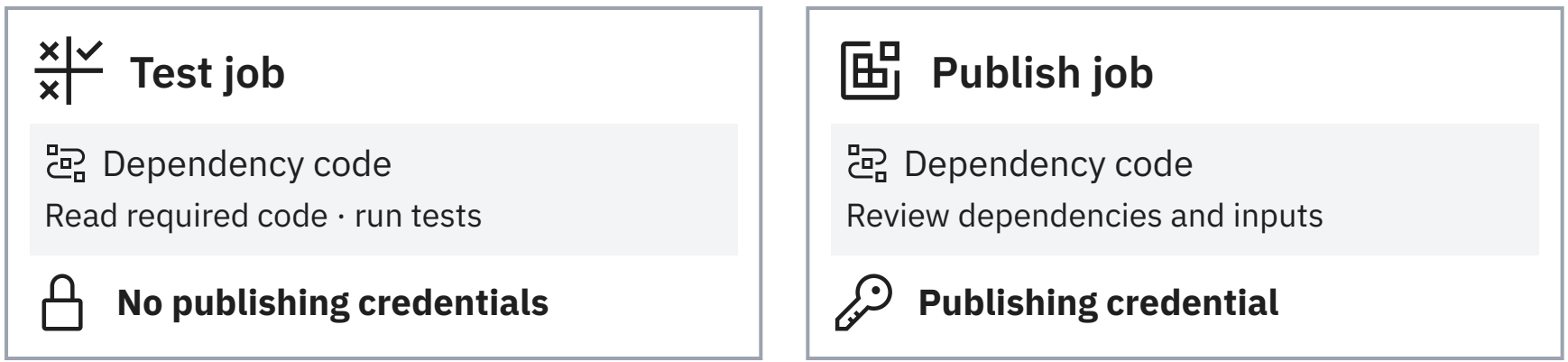
- A **provenance attestation** is a signed record of a package's source and build.
  - e.g., verify with `npm audit signatures`; inspect the source and workflow.



npm documentation example

Dependency code inherits the CI job's access

Dependency code can run during **installation, builds, and tests**.



Test dependencies need no publishing access.

Code running here can use this access.

Recall: Apply the principle of Least Privilege: grant only the access each job needs.

Today: Others depend on our releases



We consume software

Check

Where it came from
What changed
What access it receives





Others consume our releases

Provide

Checked artifacts
Evidence of their origin
A path to updates

Release engineering: decide what ships, how it reaches users, and how we keep it supported.

The fix is merged. CI is green. Are we done?

A team has corrected a bug and checked the release candidate.

Users are still running the affected version.

🎯 Learning Goals

- Identify a release's artifacts and justify using the exact outputs that were checked
- Compare who controls adoption and recovery across different products
- Use constraints to decide release scope, frequency, and required checks
- Plan compatibility checks, migration steps, and release communication
- Apply a support policy to choose fixes and backports for affected versions

Who is still running the bug?

Hosted service
The team updates the running service.

Self-managed server
A customer administrator upgrades.

Library
Another team adopts it and ships its application.

Desktop or mobile app
The update must reach users' devices.

ASK FOR EVERY PRODUCT

1. What ships, and what must pass?
2. Who decides when to adopt it?
3. How does a fix reach users?
4. What remains supported?

Release Foundations

Consumers receive a build artifact

Build inputs

Selected source

Dependencies

Build tools



Build

Artifact

A package, executable, or image

A **candidate** is an artifact being evaluated for release.

Example: a container image

Version names the release

`2026.8.0`

Digest: a hash identifying the exact image

`sha256:eeeccf3257bc…`

Rebuilding can produce an untested artifact

Tested image A
Stored after the build

→

Publish image A
The checked output reaches users.

Rebuild the same source
Dependencies or tools may change.

→

New image B
Needs its own checks

- **Checks on A do not establish B's behavior.**
- Attach check results to the artifact's **digest**; publish that checked artifact.

This candidate is not ready for general release

- A **release policy** states what may ship, when, and under what conditions
- A **release gate** enforces those requirements through automation or human review

Example Requirement	Candidate Result
Integration tests pass	✅ Passed
No known, fixable Critical runtime vulnerability	❌ One remains
Release owner approves candidate and publication time	⏳ Pending
Release notes, upgrade steps, and support period ready to publish	⏳ Pending

Pre-releases invite testing before general use

A **pre-release** is a version made available for testing before general release.

2026.8.0b3 Pre-release — **Beta:** early testing and feedback.

2026.8.0 — **Stable:** intended for general use.

Release candidate (RC): a pre-release labeled as a proposed final version.

Dogfooding is a testing practice: the team uses its own software in everyday work.

Home Assistant on GitHub: beta · stable

Releasing a version does not mean it is in use



RELEASE: MAKE THE VERSION AVAILABLE

The project publishes an update.



DEPLOYMENT: INSTALL AND RUN IT

An operator installs and runs the update.



ADOPTION: START USING THE VERSION

Users or downstream teams begin using it.

Hosted and Self-Managed Software

Who installs the update?

GitLab.com

GitLab updates the servers.

Customers use the updated service.

Self-Managed

Each customer chooses when to upgrade.

Older releases remain in use.

API clients and integrations update independently in both models.

A predictable schedule helps users plan

Release cadence is how often new releases become available.

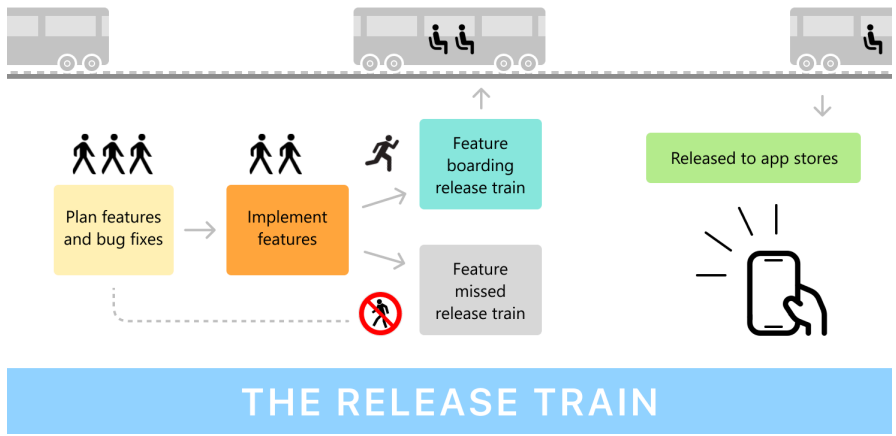
- Give the team time to test and prepare release notes.
- Give customers time to plan and check upgrades.
- Allow urgent fixes outside the routine schedule.



GitLab example: monthly feature releases, with patches between them.

GitLab · release cadence

Ready changes catch the release train



A **release train** ships ready changes on schedule; late features wait.

Feature freeze: stop adding features to this release; test and fix blockers while new development continues separately.

Runway · mobile release-train illustration

Some upgrades need an intermediate stop



Installed version
Start from the customer's version.

Required stop
Finish the data migration.

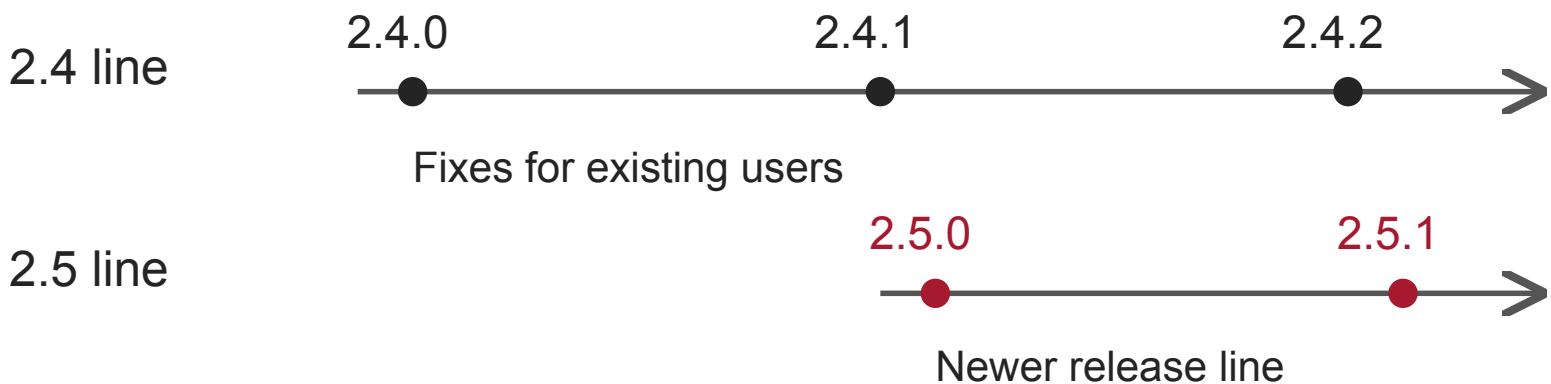
Target version
Run code that uses the updated data.

GitLab example: background migrations update stored data while the service runs. Let them finish before the next upgrade.

Illustrative path · GitLab upgrade guidance

Several release lines can remain in use

A **release line** is a series of versions maintained together.



Customers on the older line still need fixes after a new line ships.

Support has a scope

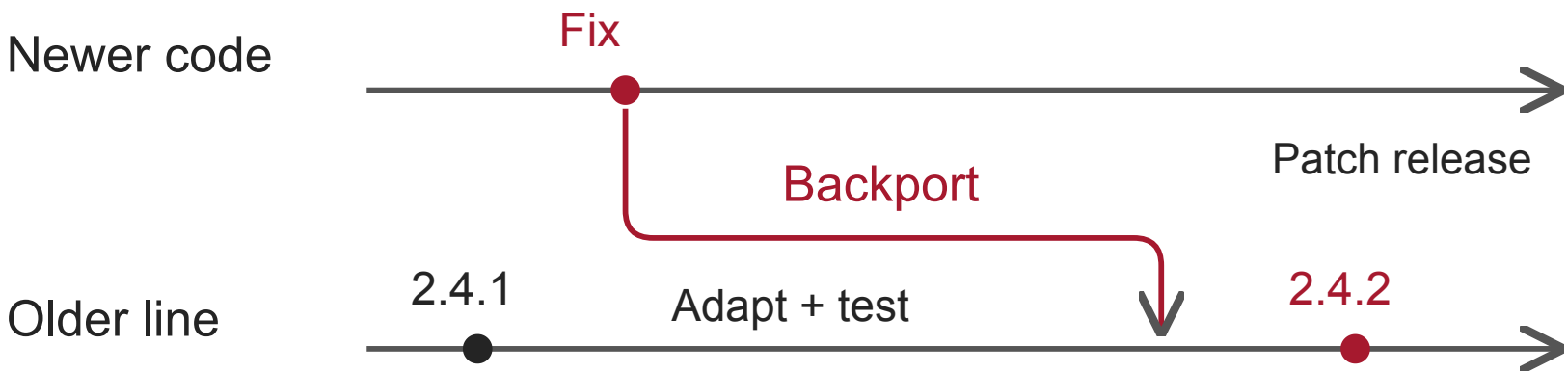
A support policy says **which versions get which fixes, and for how long**.

EXAMPLE: GITLAB'S ROUTINE MAINTENANCE

Release line	Fixes provided
Current	Bug fixes and security fixes
Previous two monthly lines	Security fixes*
Older	Outside routine support

*Some lower-risk issues have narrower coverage. GitLab maintenance policy

A backport carries a fix to older code



- **Backport:** apply a newer fix to an older line, adapting it as needed.
- Test it on that line, then deliver it in a **patch release**.

Longer support gives users time to upgrade

- **Long-term support (LTS):** selected releases receive fixes for a longer, stated period
- **Extended support:** coverage beyond standard support, often narrower or paid
- **End of life (EOL):** the applicable support commitment has ended
- **Cost:** Support requires keeping build tools, tests, and a way to publish patches



Ubuntu example: LTS releases have standard security maintenance, with additional coverage available through Ubuntu Pro.

Ubuntu · release and support cycle

Who waits for the fix?

TinyNotes: hosted and self-managed editions. Both are affected.

Line	Support commitment	Installations
3.x	Full support	40%
2.x	Security fixes; preserve integrations	50%
1.x	Support ended	10%

- **All versions:** sharing API exposes private documents without sign-in.
- **Disclosure in 48 hours.** No attacks observed; logs are inconclusive.

Patches and temporary protection

- **3.x:** checked fix ready.
- **2.x:** breaking prototype needs checks; compatible fix takes **2–4 days**.
- **1.x:** no patch estimate.

Temporary protection disrupts customer workflows.

Use the handout for mitigation options and prerequisites. Work on alternatives delays the compatible fix.

Make a plan for the next 48 hours

Groups of 3–4 · Complete the handout · Include everyone's Andrew ID

1. What will you **deploy, publish, or hold**?
2. What should **each administrator do today**?
3. Which **specialist task comes first**? What waits?

“Send us the patch. We accept the risk.”

- A large **2.x customer** uses the API for invoices; both mitigations stop it.
- They want the breaking patch **today** and can pause account creation. Existing accounts and invoice sharing still work.
- Exploit test passes; regression testing is incomplete.

Would you release it? Revise your plan.

Which cost did your plan accept?

Compare two plans:

- **Protection:** who remains exposed?
- **Compatibility:** whose work is disrupted?
- **Evidence:** what justifies each release?
- **Commitment:** what do supported and EOL customers receive?

What changed after the customer said “we accept the risk”?

Libraries and SDKs

A library update needs other teams to act

- **Libraries** provide reusable code
- **Software development kits (SDK)** bundle code and tools for building with a platform.



Example: NumPy, a Python library for working with numerical arrays.

NUMPY	→	DEPENDENT PACKAGE	→	APPLICATION	→	USERS
Publish		Adapt, test, release		Adopt and ship		Update

Downstream projects depend on your code, directly or through another package.

Removing an API creates migration work

Example: NumPy 2.0 removed `np.cast`

OLD CALL · FAILS WITH NUMPY 2.0

```
np.cast[float]([1, 2])
```

REPLACEMENT · WORKS WITH NUMPY 1.26 AND 2.0

```
np.asarray([1, 2], dtype=float)
```

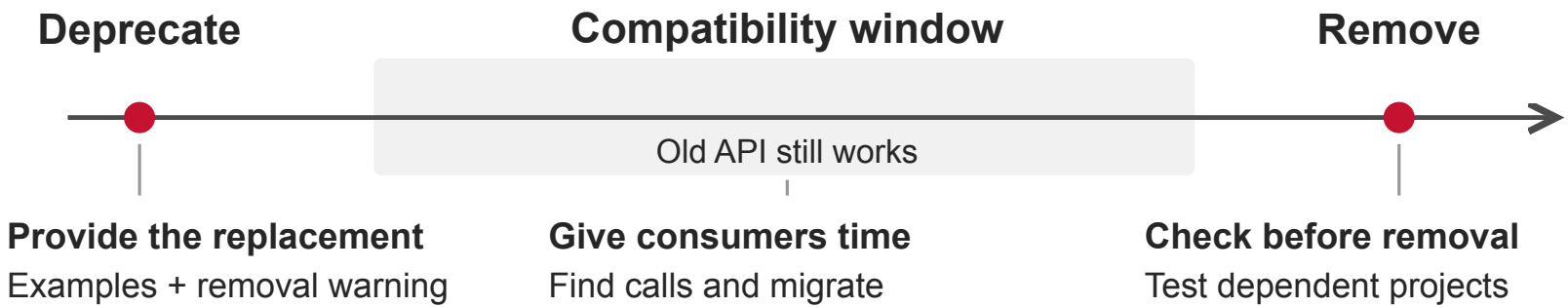
Consumers must **change the call, test their code, and release an update**.

Candidates give downstream teams time

- **Publish early:** provide a **release candidate** before the final release
- **Allow preparation:** dependent projects adapt, rebuild if needed, and test
- **Use their feedback:** their tests can expose failures your own tests miss

NumPy: prepare downstream releases between **2.0.0rc1 and 2.0 final**.

Give consumers time to stop using an old API



- **Deprecation:** an interface is being retired, but it still works.
  - Warn developers when they use it (e.g., via compiler or runtime warnings)

Compatibility choices shift the work



Keep the old API
Existing code keeps working

→ **Producer maintains it**
Keep and test old behavior



Remove the old API
The library can be simpler

→ **Consumers migrate**
Adapt, test, and release their code

MAKE MIGRATION EASIER

- **Codemods** rewrite clients automatically; developers review and test the changes
- **Monorepos** can let teams update an internal library and its callers together

Desktop and CLI Applications

Different machines need different artifacts



WINDOWS

Installer

x64 · ARM64



MACOS

Disk image

Intel · Apple silicon



LINUX

Package or archive

x64 · ARM64

- **OS and processor architecture** affect what can run
- You need to test installation and upgrades on the platforms you support

Releases reach users through different channels



Direct download
Get an installer or archive from the website.



App store
The store distributes approved apps and updates.



Package manager
Install or upgrade from a package repository.

The same release may become available at different times.

Publication can fail on one channel



- **2.77.0:** official Linux repository publication failed; release marked incomplete.
- **2.78.0:** users directed to a new release, primarily to repair distribution.

Test installation through the channels your users actually use.

GitHub CLI · 2.77.0 release · 2.78.0 release

Firefox offers streams at different stages



A **release stream** chooses which builds you receive.

Nightly
Try new work early.

→

Beta
Test and stabilize.

→

Release
Ship to general users.

A **distribution channel** is where you get it; a **release stream** is which builds you follow.

Shipped code can remain disabled

A **feature flag** switches a feature on or off.



FLAG OFF

The build contains the code.
Users cannot access the feature.



Enable



FLAG ON

The same build exposes
the feature to users.

Publishing a build and enabling a feature are separate decisions.

A fix is ready the day before release. What must we know before it ships?



Late fixes must justify the risk they add

Suppose a new feature crashes in Firefox Beta.

A fix is ready on the development branch.



Defer the fix if the problem can safely wait



Disable the feature if that safely avoids the crash



Accept a verified fix if the urgency justifies its risk

An **uplift (backport)** brings a selected patch into a more stable branch.

Available updates still need to be installed

Example



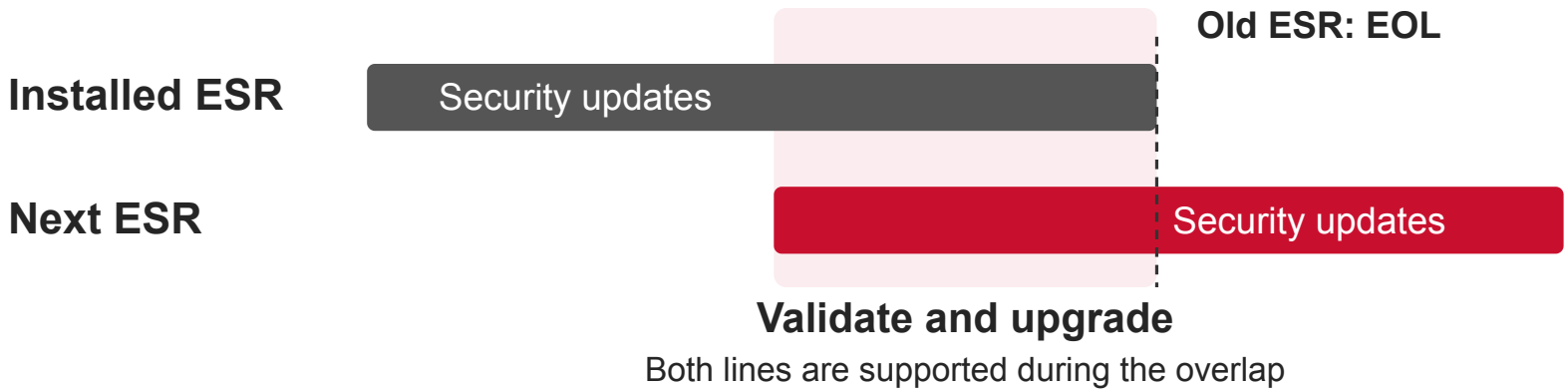
Restart to apply

- **Manual:** download a replacement or run a package-manager command
- **Prompted:** the app asks the user to install or restart
- **Automatic:** an updater handles the installation

Permissions, restarts, and IT policies still affect adoption

Firefox ESR allows time for validation

Extended Support Release (ESR) provides a stable feature set with security updates.



Organizations can **test the next ESR before deploying it** across their desktops.

Mozilla · ESR release cycle

Mobile Applications

Store approval and device updates take time

Our case: Spotify. The app team, store, and device each control a step.



APP TEAM

Checks the build and submits it.



APP STORE

Reviews the submission before distribution.



DEVICE AND USER

Install as settings and connectivity allow.

Plan for overlapping app versions: backend changes and launch dates must account for users who have not updated.

Readiness comes before store approval

Spotify's example dashboard: this candidate is not ready.



- **A blocker remains; the crash rate is too high.**
- Clear the team's release checks, then submit for store review.

Spotify Engineering · example release dashboard · two excerpts

Available, installed, and enabled differ

Spotify Audiobooks: ship the code behind a feature flag before the scheduled launch.



AVAILABLE

The store offers the update.

INSTALLED

Devices run the new version.

ENABLED

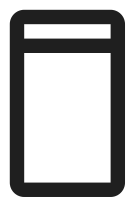
Users can access the feature.

Spotify waited for sufficient adoption, then gradually enabled **Audiobooks**.

Spotify Engineering · app distribution and feature launch

Some devices cannot install the update

A **minimum supported OS version** sets which devices can run an app release.



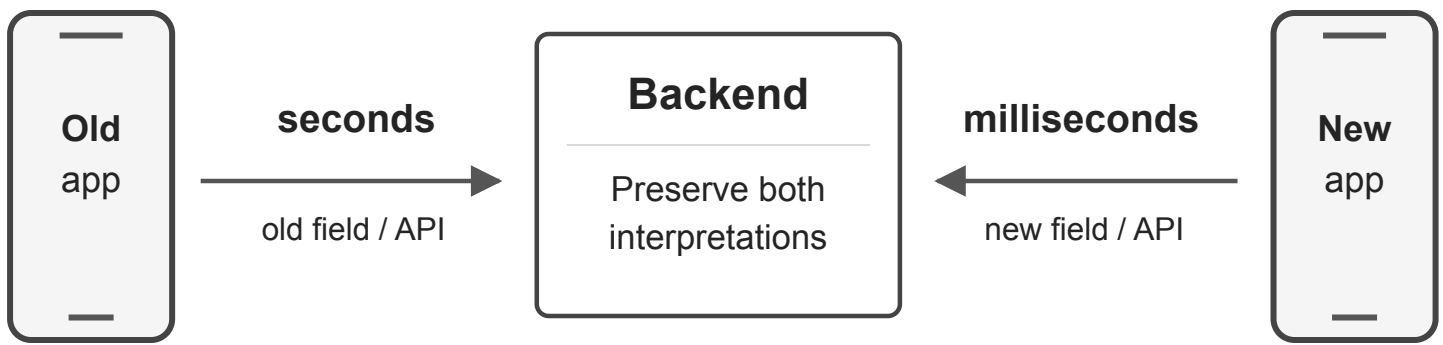
Older phone

OS cannot upgrade further
New app requires a newer OS

- Supporting older OS versions needs more testing and fallback behavior.
- Raising the minimum can leave users **on an older app version**.

The backend must support older app versions

Illustrative sync change: old and new apps remain installed when the backend changes.



Reusing one field without a version would misread one group of clients.

Use a distinct field or API version; **test both supported clients against the backend**.

Pausing distribution leaves installed copies



PAUSE DISTRIBUTION

Limit how many more users receive the faulty app.



MITIGATE REMOTELY, WHEN POSSIBLE

A backend change or flag may disable the affected behavior.



DELIVER A CORRECTED APP

A code repair still needs review, distribution, and installation.

Recovery options informed by Spotify's release account

The same fix needs different release plans

Case	Constraint	Release response
GitLab	Operator or customer controls the server	Customer upgrade guidance
NumPy	Other products embed the library	Time for downstream preparation
Firefox	Desktop users need stability; organizations need time	Stabilization and support overlap
Spotify	Stores and devices separate shipment from use	Separate shipment and feature enablement

🚀 Today, we learned how to ...

- Identify a release's artifacts and justify using the exact outputs that were checked
- Compare who controls adoption and recovery across different products
- Use constraints to decide release scope, frequency, and required checks
- Plan compatibility checks, migration steps, and release communication
- Apply a support policy to choose fixes and backports for affected versions