Testing published packages across WordPress versions

- Map what runs in production
- Choose the versions to test
- Test an existing consumer
- Test four version combinations
  - Check the build
  - Check the behaviour
- Ship changes in a safe order
- Keep public and private APIs distinct
- Recognize package identity problems
- Examples from Gutenberg
  - ThemeProvider
  - DataViews

A plugin can bundle one `@wordpress/*` package while loading its dependencies from WordPress. The bundled package and its WordPress dependencies can then come from different releases.

For example, a plugin might bundle a newer `@wordpress/dataviews` package but run on a WordPress version that provides an older `@wordpress/components` package. The reverse can also happen: an older plugin bundle can run on a newer WordPress version.

Use this guide when a change affects parts that can update separately, such as:

- A public or private API.
- A package dependency or entrypoint.
- Package registration or a check, such as an allowlist or opt-in, that decides whether an entrypoint can load.
- A compatibility fallback.

If the package and all its dependencies always ship together, you do not need the four-version test matrix. Public API changes still need the existing-consumer check below.

Map what runs in production

Start by recording:

- Which packages are included in the plugin or application bundle.
- Which dependencies are loaded from WordPress.
- The oldest WordPress version the plugin or application supports.
- Older bundle versions that maintained plugins or applications still use.

Check every affected entrypoint. An entrypoint is an import path with its own build, such as the main package import or a `/wp` import.

Check package metadata, build settings, generated asset files, and dependency extraction. Dependency extraction decides which packages the bundle leaves for WordPress to load.

Do not rely only on package names or source imports. They do not show what the production build contains. Publishing a new package also does not update bundles that are already deployed.

Choose the versions to test

For the old package, start with the last release before the change. Include earlier releases when maintained consumers still use them.

For WordPress, start with the oldest supported version and the version that contains the proposed change. Add intermediate versions when they expose a different API or load a different entrypoint.

If a combination is outside the support policy, state that clearly. Record the package version, entrypoint, or WordPress version that consumers must use instead.

Test an existing consumer

For a public API change, use representative consumer code with the candidate package:

1. Compile it against the package’s published TypeScript declarations.
2. Build it with its normal build settings.
3. Exercise the changed behaviour.

Updating call sites inside the Gutenberg repository proves that the new API works. It does not prove that existing consumers still work or have a migration path.

Test four version combinations

When WordPress supplies a dependency, test both mixed-version combinations:

Bundle	WordPress dependency	What to confirm
Old	Old	Existing behaviour still works.
Old	New	The new dependency still supports the old bundle.
New	Old	The new bundle works on every supported WordPress version.
New	New	The new code works as intended.

Use the built package for these checks. Unbuilt source is not the code that consumers install.

Check the build

For each required combination and entrypoint:

1. Install the exact package versions in an isolated consumer.
2. Compile against their published TypeScript declarations.
3. Build with the consumer's real dependency extraction settings.
4. Check the generated dependencies and exported APIs.
5. In the relevant WordPress version, confirm that registration and other load checks select the expected entrypoint.

Check the behaviour

Run the affected behaviour on the relevant WordPress version. Include the fallback path when the change adds one.

Also check state that must come from the same package copy. React contexts, private API locks, registries, and symbols are common examples. Identical exports do not make separate package copies share the same state.

Record every required combination as `pass`, `fail`, or `unverified`. A source test or mock can show which branch runs, but it cannot reproduce dependency extraction, separate package copies, or an older WordPress runtime.

Ship changes in a safe order

The two mixed-version combinations need different protections:

- **New bundle with old WordPress:** Check whether the API exists before using it, provide a fallback, or explicitly raise the minimum supported WordPress version.
- **Old bundle with new WordPress:** Keep the old API or compatibility path until maintained consumers no longer need it.

When old and new APIs can coexist, use this release order:

1. Add the new API to the WordPress-supplied package. Keep the old API while supported bundles still need it.
2. Update the bundled package to prefer the new API and fall back when an older WordPress version does not provide it.
3. Prevent new code from using the old API outside the fallback.
4. Publish the required npm and WordPress releases before removing either compatibility path.

Check whether the API exists instead of comparing version strings. Keep the fallback in one place in the bundled package.

Remove a private compatibility API and the bundle's fallback as separate release changes. Before each removal, repeat the consumer and version checks. A planned removal version is a review point, not proof that deployed bundles have updated.

Keep public and private APIs distinct

Production public APIs follow the backward compatibility policy.

New bundled packages must not use private APIs. Repository lint and bundle checks flag new direct and indirect use, but they cannot change older bundles that are already deployed.

Private APIs can be removed. A temporary compatibility path does not make a private API public. Keep it only while a supported older bundle needs it. Remove it after the replacement has shipped and the required combinations pass, or make the support change explicit.

Recognize package identity problems

An export fallback cannot fix a package identity problem. A React context created by one package copy cannot provide a value to a consumer using another copy. A private API lock created by one copy has the same limitation.

For these cases, use an API that does not share package-local state. Otherwise, change the entrypoint or supported WordPress version so both sides use the same package copy.

Examples from Gutenberg

ThemeProvider

The `ThemeProvider` change in #78958 made the API public and removed its private path in the same release. Published bundles broke.

#79594 restored the private path. #79620 then kept both paths while consumers migrated.

DataViews

The DataViews cleanup discussion in #81230 found the reverse problem. An older bundled `@wordpress/dataviews` package could still request a private `@wordpress/components` API after a newer WordPress runtime removed it.

#82221 restored the opt-in for those older bundles. A bundled `@wordpress/private-apis` copy still cannot unlock an object created by a different runtime copy.

□□□ #2
□ □□□ □□□ 2026-10-31 18:25:56 UTC
□ □□□ □□□ 2026-10-31 18:25:58 UTC