Managing transitive dependency conflicts during package migrations

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Managing transitive dependency conflicts during package migrations

Table of Contents

Introduction **1**
Intended audience 1
Objectives 1
Overview 2
Dependency types 2
Transitive dependency conflicts 2
Shared dependencies in distributed systems 3
Dependency structure 3
When version conflicts arise 3
The migration attempt 3
Why the build fails 4
The solution 4
When conflicts do not cause errors 4
How transitive dependency conflicts cause build errors 5
Configured dependencies 5
Build-time dependencies 5
Using breadth-first search to sequence a migration 6
Step 1: Understand your project's dependency graph 6
Step 2: Define your migration goal 6
Step 3: Start from a known good state 6
Step 4: Traverse the graph with breadth-first search 7
Step 5: Detect and resolve conflicts 7
Step 6: Automate conflict detection 7
Step 7: Iterate and test 7
Step 8: Document and communicate 8
Step 9: Prepare a rollback plan 8
Step 10: Monitor and maintain 8
Migration example 8
Transitive dependency analyzer 9
What the tool reports 9
Prerequisites and setup 9
Running the script 10
Interpreting the report 10
Reanalyze after each pass 11

Best practices **12**
FAQ **13**
What is a transitive dependency conflict? 13
What does the transitive dependency analyzer detect? 13
When should I pin a transitive dependency version instead of upgrading it? 13
How do I scope a migration when many packages share the same dependency? 14
Can I use this approach with package managers other than npm? 14
How do I handle npm peer dependency warnings during a migration? 14
Next steps **15**
Resources **16**
Tools 16
References 16
Contributors **17**
Document History **18**

Managing transitive dependency conflicts during package migrations

Tom Ron, David Guardiola, Mike Kight, and Anisha Salunkhe, Amazon Web Services

Dependency migration in distributed systems requires careful sequencing. When a foundational package undergoes a major version change that breaks backward compatibility, all packages in the dependency tree must be updated in the correct order. If transitive dependencies are not updated in the proper sequence, applications fail to build. This guide provides a systematic approach to identifying and resolving these conflicts using breadth-first search (BFS) methodology.

Intended audience

This guide targets software engineers, technical leads, and engineering managers who plan package migrations in distributed systems.

Readers should have the following skills:

- Working knowledge of how a build system resolves direct and transitive dependencies
- Experience with a package manager such as `npm`, Maven, NuGet, Go modules, or Cargo
- Comfort running command line tools and reading their output

Objectives

This guide contributes to the following outcomes:

- **Reduced migration planning time**: Teams derive a migration sequence from tool output instead of inspecting dependency trees by hand. Planning time drops without sacrificing productivity or quality.
- **Fewer version migration errors**: Teams migrate shared dependencies before the packages that consume them. This removes the most common cause of failed migration builds.

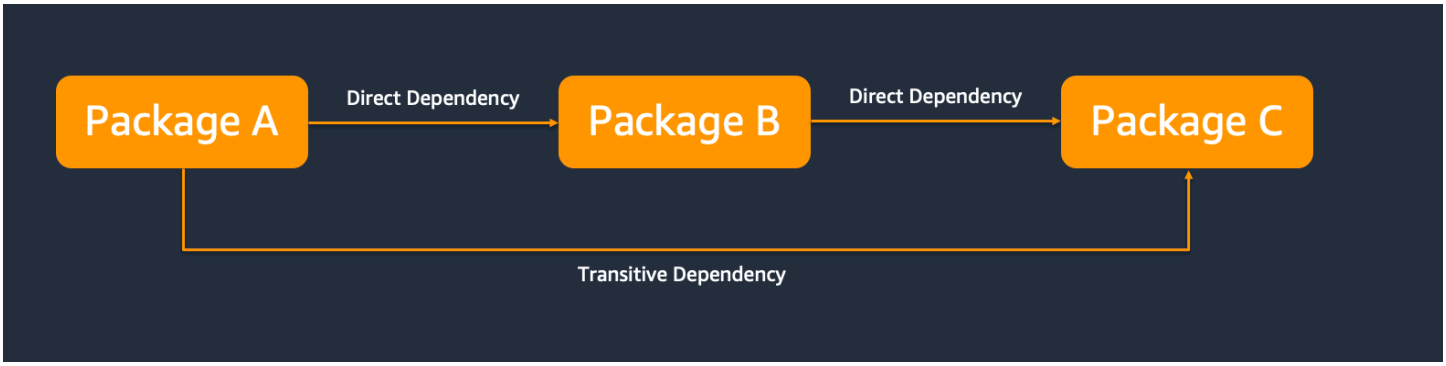
Overview

Transitive dependencies, indirect dependencies inherited through other packages, create complex relationship chains that are not always visible in project configuration files. During a major version migration, these hidden relationships can block progress when lower-level packages have not yet been updated to support the new version.

Build systems manage two fundamental categories of dependencies that determine how packages interact and compile together.

Dependency types

- **Direct dependency**: When Package A relies on Package B, Package B is a direct dependency of Package A. Direct dependencies are declared explicitly in project configuration files such as `package.json` or `pom.xml`.
- **Transitive dependency**: When Package B relies on Package C, and Package A relies on Package B, Package C becomes a dependency of Package A. This indirect relationship makes Package C a transitive dependency of Package A.



Transitive dependency conflicts

In distributed software systems, shared components frequently rely on third-party core packages. Using reusable components and third-party packages accelerates innovation and reduces development time, but introduces complexity in dependency management.

Shared dependencies in distributed systems

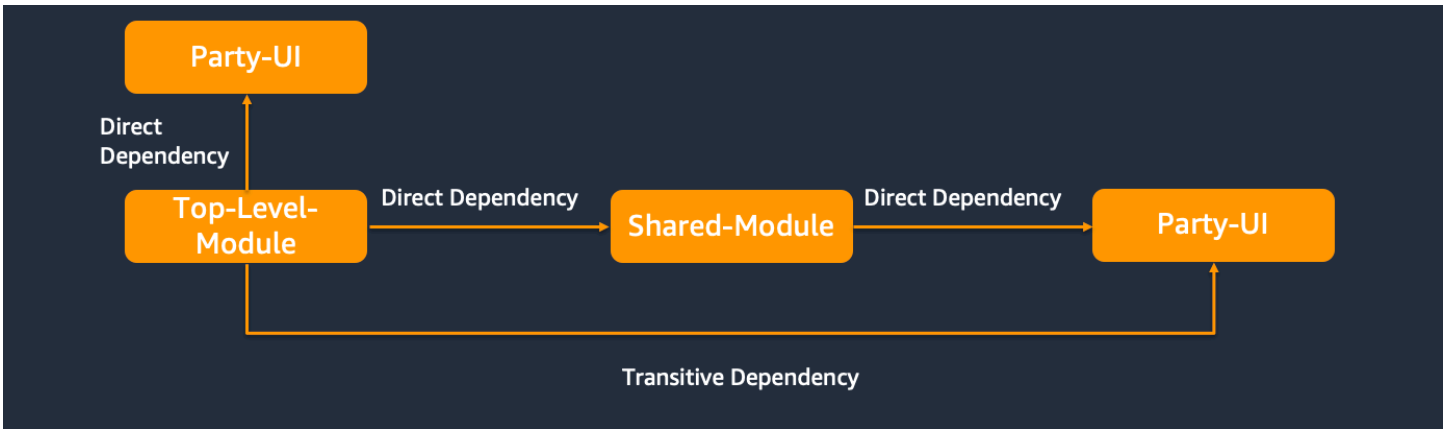
It is common to encounter shared dependencies between top-level components and reusable components within a system. Consider the following example:

Your organization has adopted a third-party UI/UX library called Party-UI. Party-UI is a direct dependency for a UI/UX component named Top-Level-Module, which you are developing to introduce a new feature.

As part of implementing this feature, you use a commonly employed package called Shared-Module, which also relies on Party-UI. Party-UI now becomes both a direct dependency of Top-Level-Module and a transitive dependency through Shared-Module.

Dependency structure

The following diagram shows Party-UI as both a direct dependency of Top-Level-Module and a transitive dependency through Shared-Module.



When version conflicts arise

When the version of a direct dependency is incompatible with the version of a transitive dependency, the build fails. In npm, this class of mismatch often surfaces as a peer dependency conflict.

The migration attempt

Party-UI releases version 3.0.0. It deprecates components, changes namespaces, and drops compatibility with the 2.x line. Your team updates Top-Level-Module for the new APIs. The build still fails, and the errors come from files in Shared-Module.

Why the build fails

When Top-Level-Module declares Party-UI 3.0.0, the build system resolves that version for the whole project. Shared-Module sits in the same dependency tree, so the build system compiles it against 3.0.0 as well.

Shared-Module was written for Party-UI 2.5.8. Compiling it against 3.0.0 produces API mismatches, missing components, and changed method signatures. The compilation fails.

Top-Level-Module cannot move to Party-UI 3.0.0 until Shared-Module moves first.

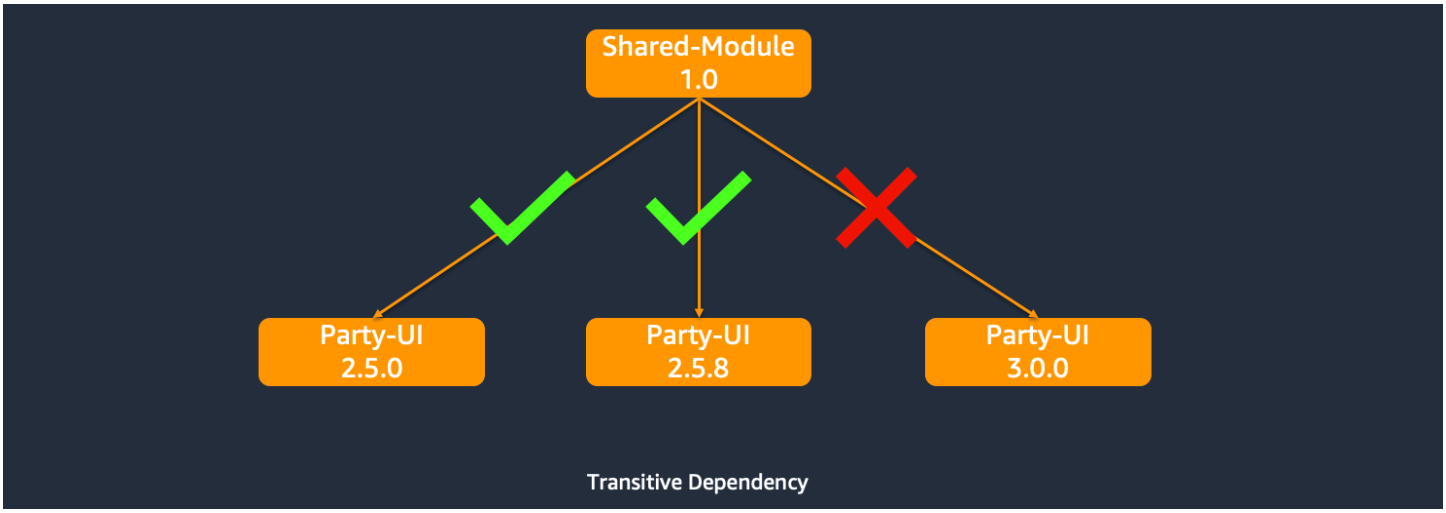
The solution

Map the relationships between packages before you migrate anything. A breadth-first search of the dependency tree produces that map. It identifies which packages sit below your target, so you update lower-level dependencies first and avoid the build failure above.

When conflicts do not cause errors

Transitive dependency conflicts do not always result in errors. Differences between patch versions or minor iterations are often minimal enough to avoid triggering conflicts. For example, Party-UI 2.5.0 and Party-UI 2.5.8 exhibit only slight alterations between them and maintain backward compatibility.

Errors from transitive dependency conflicts typically occur during major version transitions. Party-UI 2.0 might feature components that are deprecated in Party-UI 3.0. Packages built on Party-UI 2.0 may encounter compatibility issues when forced to operate on Party-UI 3.0.



How transitive dependency conflicts cause build errors

Transitive dependency conflicts can cause build errors through both configured dependencies and build-time dependencies.

Configured dependencies

Configured dependencies are those you explicitly specify in your project's configuration files or build scripts.

Version mismatch: You pin a direct dependency to a specific version, and that dependency requires a conflicting version of a transitive dependency. For example, you declare Dependency A version 1.0, and Dependency A requires Transitive Dependency X version 2.0.

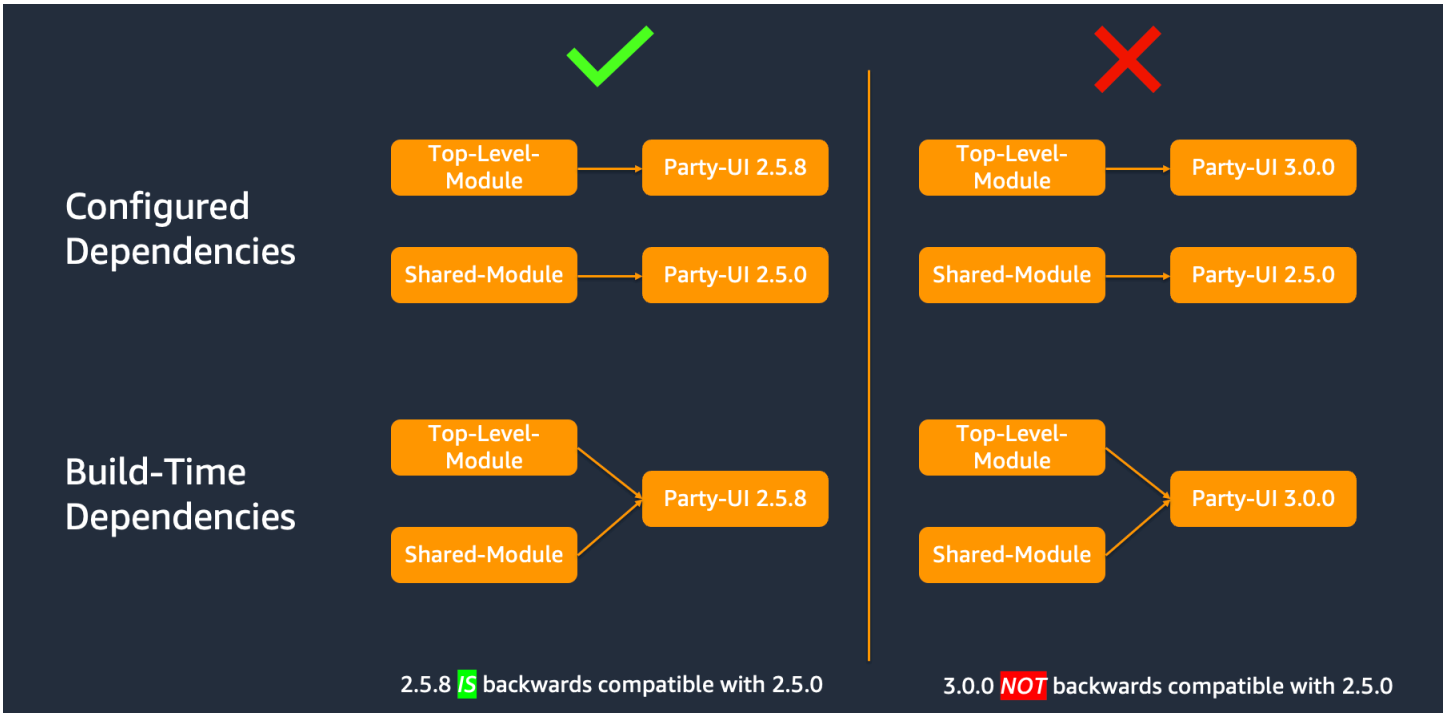
Incompatibility: You select a direct dependency that does not work with the resolved versions of its transitive dependencies. The result is compilation errors, runtime errors, or unexpected behavior.

Build-time dependencies

Build-time dependencies are those your build tool relies on during compilation.

Conflict resolution: Different parts of your project require different versions of the same transitive dependency. The build tool must pick one version. If it picks an incompatible version, or cannot resolve the conflict, the build fails.

Build process failures: The build tool fails to download, install, or link a required dependency because of the conflict. The project never compiles.



Using breadth-first search to sequence a migration

Avoiding dependency conflicts during a migration using breadth-first search (BFS) is a systematic approach to identify and address potential issues in your project's dependencies. BFS helps you traverse the dependency tree layer by layer, starting from the root package being migrated and expanding outward.

Step 1: Understand your project's dependency graph

Identify every direct and transitive dependency, its version, and its relationships. Use `npm ls`, `mvn dependency:tree`, or the equivalent inspection command for your package manager.

Step 2: Define your migration goal

State the goal precisely. Record the target package, the version you are moving from, and the version you are moving to.

Step 3: Start from a known good state

Confirm that the project builds before you change anything. Create a branch or snapshot so you can return to the working state.

Step 4: Traverse the graph with breadth-first search

Apply the following traversal:

1. Start with your project's direct dependencies.
2. For each direct dependency, read its own dependencies.
3. Add those dependencies to a queue for processing.
4. Process the queue, and record every package that depends on the target package and the version it uses.

Step 5: Detect and resolve conflicts

Look for these conflicts during the traversal:

- **Version conflicts**: A package depends on a version of the target that your goal version replaces.
- **Compatibility issues**: The new version of the target introduces breaking changes.
- **Deprecated dependencies**: A package in the path is no longer maintained or supported.

For each conflict, take one of these actions:

- Update the dependency to a compatible version.
- Replace the dependency with an alternative that meets your requirements.
- Change your code to work with the updated dependency.

Step 6: Automate conflict detection

Run a script or dependency analysis tool instead of reading trees by hand. Automation removes transcription errors and makes the traversal repeatable.

Step 7: Iterate and test

After you resolve conflicts, test the affected packages. Run unit tests, integration tests, and targeted manual checks to confirm the migration introduced no regressions.

Step 8: Document and communicate

Record the dependency updates you made and the order you made them in. Share the roadmap with the teams that own packages in the tree.

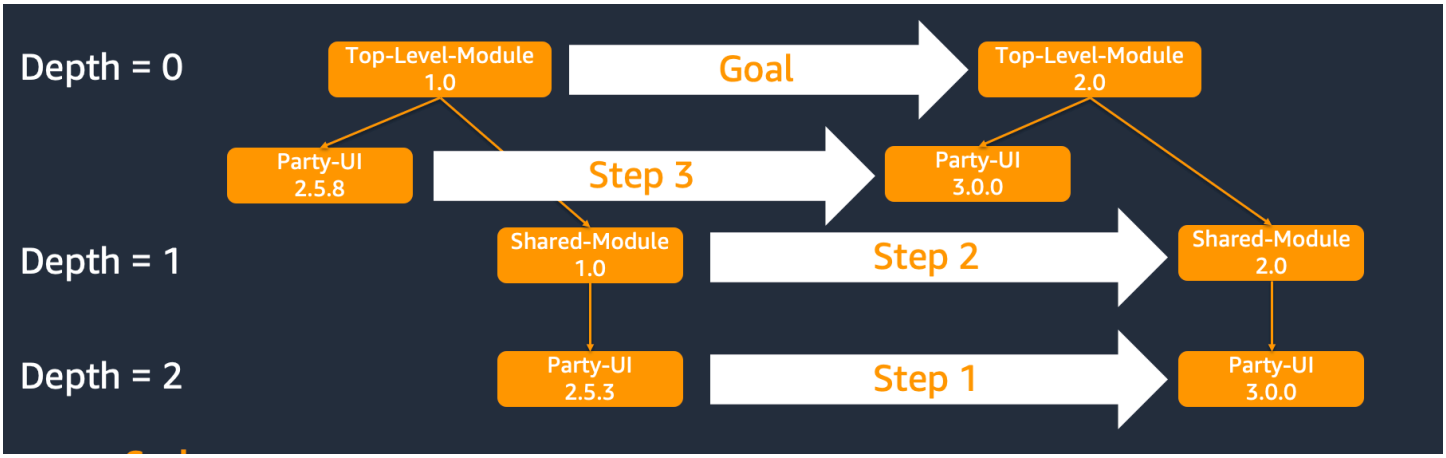
Step 9: Prepare a rollback plan

Define how to return to the previous working state. You need this if a migrated package exposes a defect you cannot fix quickly.

Step 10: Monitor and maintain

Review the dependency graph after the migration. Repeat the analysis when the target package publishes its next major version.

Migration example



Goal: Migrate Top-Level-Module 1.0 to a 2.0 version that uses Party-UI 3.0.0.

1. Starting at the lowest level of the dependency tree, migrate the Party-UI usage in Shared-Module 1.0 to Party-UI 3.0.0.
2. Publish Shared-Module 2.0 to mark the version that uses Party-UI 3.0.0.
3. In Top-Level-Module, replace Party-UI 2.5.8 with Party-UI 3.0.0, and Shared-Module 1.0 with Shared-Module 2.0.

Transitive dependency analyzer

The Transitive Dependency Analyzer for Node.js Projects is a Python command line script that applies the breadth-first traversal described earlier. It reads the dependency tree of a Node.js project and reports every package that depends on a target package, along with the version each one uses.

What the tool reports

The script writes a text report. Each occurrence in the report contains the following fields.

Field	Meaning
Parent Package	The package that depends on the target package
Depth Count	How far the parent package sits from the root of the tree. A higher count means the package sits lower in the tree.
Breadth Count	The order in which the traversal discovered the package
Version	The version of the target package that the parent package depends on

Use the depth count to build the migration sequence. The script reports occurrences and versions. You compare those versions against your target version to decide which packages need work.

Prerequisites and setup

The script requires Python 3.8 or later, Node.js and `npm` on your `PATH`, and a Node.js project that contains a `package.json` file. It uses only the Python standard library, so no package installation is needed. Clone the repository and follow the installation instructions in the repository README.

Running the script

Run the script with two required arguments. Set `--target` to the dependency you are searching for in the dependency tree. Set `--path` to the directory of the project you are analyzing.

```
python tree_traverse.py --target [TARGET_NAME] --path [PROJECT_PATH]
```

The script writes the report to `<target>_dependency_analysis.txt` unless you set an output path with `--output`. Add `--verbose` for detailed logging. Because the script has no third-party dependencies, you can also run it as a step in a build pipeline.

Interpreting the report

The report identifies every package in the dependency tree that depends on the target package. Those parent packages are the candidates for your migration roadmap.

The following output comes from a customer migration, with package names obscured. The customer needed to move a top-level component to Transitive-Dependency-Component v4.0.0. The analysis shows which packages still depended on v3.0.0.

```
Dependency occurrences

Parent Package: node
Depth Count : 1
    currently depends on version 3.0.0 of Transitive-Dependency-Component

Parent Package: UI-Module-Card
Depth Count : 2
    currently depends on version 3.0.0 of Transitive-Dependency-Component

Parent Package: UI-Components-Card
Depth Count : 2
    currently depends on version 3.0.0 of Transitive-Dependency-Component

Parent Package: UI-Module-Container
Depth Count : 2
    currently depends on version 3.0.0 of Transitive-Dependency-Component

Parent Package: UI-Components-Order
Depth Count : 2
    currently depends on version 3.0.0 of Transitive-Dependency-Component
```

```
Parent Package: UI-Module-Shell
Depth Count : 2
    currently depends on version 3.0.0 of Transitive-Dependency-Component

Parent Package: UI-Components-Dependency
Depth Count : 3
    currently depends on version 3.0.0 of Transitive-Dependency-Component

Parent Package: UI-Components-Workflow
Depth Count : 2
    currently depends on version 3.0.0 of Transitive-Dependency-Component

Parent Package: UI-Widget-Common
Depth Count : 2
    currently depends on version 3.0.0 of Transitive-Dependency-Component
```

> **Note**
>
> The analyzed package appears as its own entry at depth 0, labeled `root`. The earlier release that produced the example above labeled it node.

Build the roadmap from the report as follows:

1. Identify the parent packages that depend on the version you are replacing.
2. Sort those packages by depth count, highest first. Packages lower in the tree have a higher depth count.
3. Migrate from the bottom of the tree upward. Place the packages at the greatest depth first in the roadmap, and the packages at depth 1 last.
4. Migrate the package you ran the analysis against after every package in the report is migrated.

Reanalyze after each pass

Dependency analysis is iterative on large projects. Rerun the script after each migration wave. A clean report confirms that no package in the tree still depends on the version you are replacing.

Best practices

Analyze your most-used components first. In a distributed system, in-house shared packages often have many components depending on them. A transitive dependency conflict in one of these shared packages can block progress across every component that uses it.

Run the transitive dependency analysis on your application's most-used, customer-facing components first. Analyzing several complex components surfaces the shared lower-level dependencies that block the most other work. Migrating those shared dependencies first unblocks the largest amount of progress for the least effort.

Treat every pinned version as work you still owe. Pinning unblocks a release, but it leaves the conflict in place. Record a revisit date when you pin. Automate reanalysis in your build pipeline. Add the dependency analysis script as a CI step that runs on pull requests touching package configuration files. Automated checks surface new conflicts before they reach the main branch.

Set version-range policies for shared packages. Define whether shared packages accept minor-version ranges or require exact pins. A written policy reduces ad-hoc decisions during migrations and makes pinning exceptions visible.

Document your dependency governance model. Record who owns each shared package, who approves version bumps, and how cross-team migrations are coordinated. This removes ambiguity when multiple teams share the same dependency tree.

FAQ

What is a transitive dependency conflict?

A transitive dependency conflict occurs when your package depends on Package X, Package X depends on Package Y version 1.x, and your migration target requires Package Y version 2.x. You never declared a dependency on Package Y. It arrived through the package you do depend on. During migration, these hidden version requirements collide.

What does the transitive dependency analyzer detect?

The analyzer reports occurrences, not verdicts. It runs a breadth-first traversal of the dependency tree of a Node.js project and records every package that depends on the target package. For each occurrence, it reports the parent package, the depth count, the breadth count, and the version in use. You compare the reported versions against your target version to identify the conflicts.

When should I pin a transitive dependency version instead of upgrading it?

Pin the version when the following conditions apply:

- The breaking change in the new version does not affect how you use the package.
- You need a fast, low-risk fix to unblock a release.
- The maintainer has announced a compatibility shim for the next minor release.

Upgrade the version when the following conditions apply:

- The pinned version has known security vulnerabilities.
- The test suite of your target package runs against the newer version.
- Pinning would force you to fork or patch the dependency yourself.

Pinning is a short-term tactic. Treat every pin as technical debt with a revisit date.

How do I scope a migration when many packages share the same dependency?

Start with a breadth-first impact analysis:

1. Identify the direct consumers of the package you are migrating.
2. For each consumer, map its transitive dependents.
3. Score each path by risk. Consider the number of hops, test coverage, and deployment frequency.
4. Group the packages into migration waves. Start with the packages that have the highest test coverage and the fewest hops.

The Interpreting the report section shows this in practice. The depth count tells you which packages sit lowest in the tree, so you know which ones to migrate first.

Can I use this approach with package managers other than npm?

Yes. The method applies wherever packages declare dependencies on other packages, including Maven, NuGet, Go modules, and Cargo. Impact scoring, wave planning, and bottom-up sequencing do not depend on the ecosystem.

The sample script is specific to Node.js, because it reads `npm` dependency output. For other ecosystems, generate the tree with the native command, such as `mvn dependency:tree`, and apply the same traversal.

How do I handle npm peer dependency warnings during a migration?

When npm reports `ERESOLVE` or peer dependency warnings during migration, it signals that a transitive package requires a different version than the one you declared. Run `npm ls` to identify which paths pull in the conflicting version. Then decide whether to update the transitive consumer first (preferred) or use `--legacy-peer-deps` as a temporary workaround. Treat `--legacy-peer-deps` the same as a pinned version: record a revisit date and resolve the underlying conflict before your next release.

Next steps

- Migrate components in order from the lowest level of the dependency tree to the top-level component.
- Run the transitive dependency analysis on every package you plan to migrate, before you commit to a date.
- Include the time and resources needed to migrate lower-level dependencies in your roadmap.

Resources

Tools

- Transitive Dependency Analyzer for Node.js Projects

References

- Anti-patterns for software component management (AWS Well-Architected DevOps Guidance)
- npm ls (npm Docs)
- Introduction to the dependency mechanism (Apache Maven)

Contributors

- David Guardiola, Professional Services Consultant, AWS
- Mike Kight, Cloud Application Developer, AWS
- Tom Ron, Generative AI Architect, AWS
- Anisha Salunkhe, Cloud Applications Architect, AWS

Document History

The following table describes significant changes to this guide.

Change	Description	Date
Initial publication	—	August 17, 2026