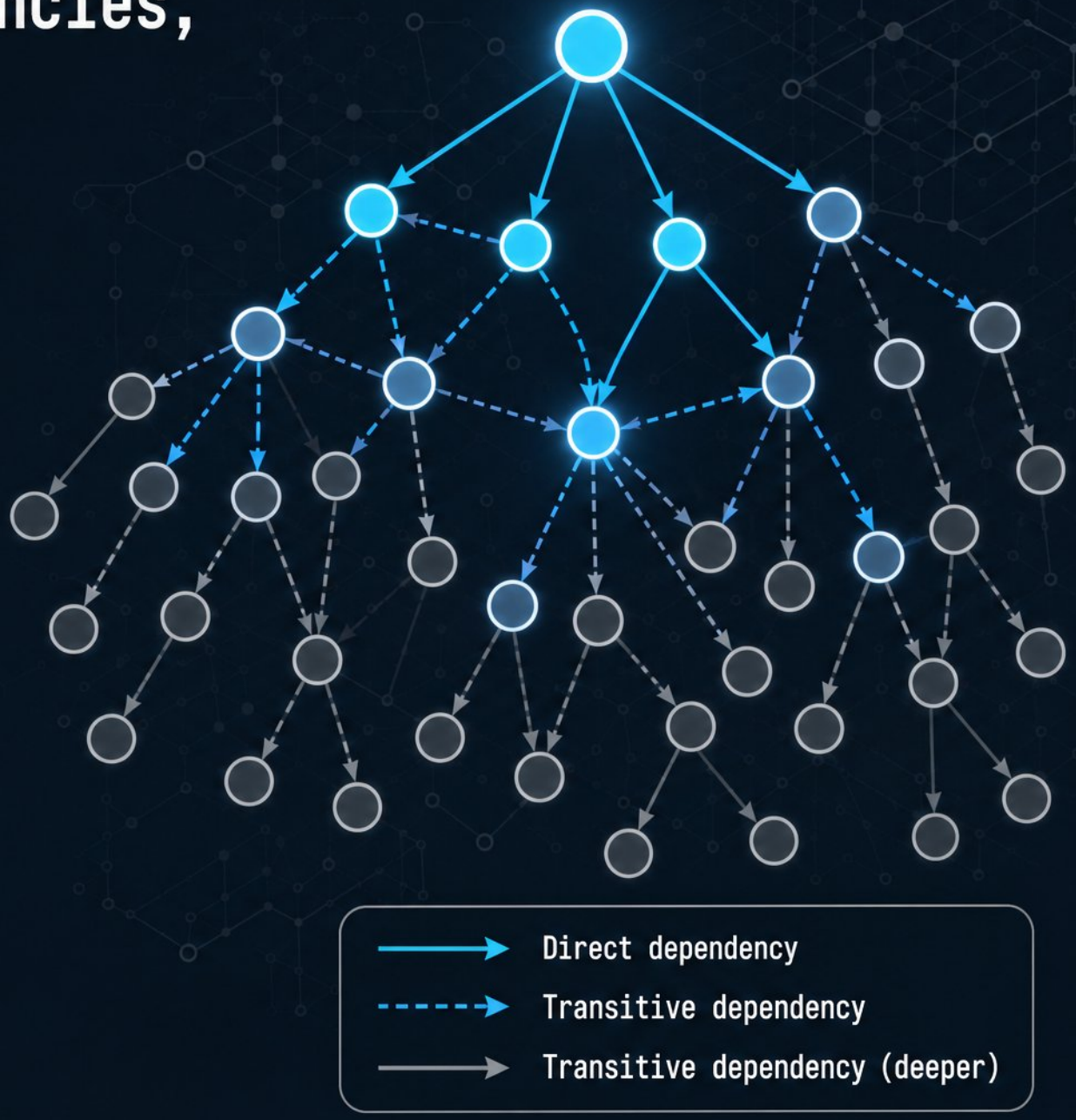
Understanding Package Dependencies: Versions, Requirements, and Compatibility

- Modern projects rely on external packages that evolve independently
- Getting it wrong leads to breaches (xz-utils, Axios, Shai-Hulud worm), build failures, and runtime errors
- This training covers versioning, requirements, breaking changes, resolution algorithms, and supply-chain health

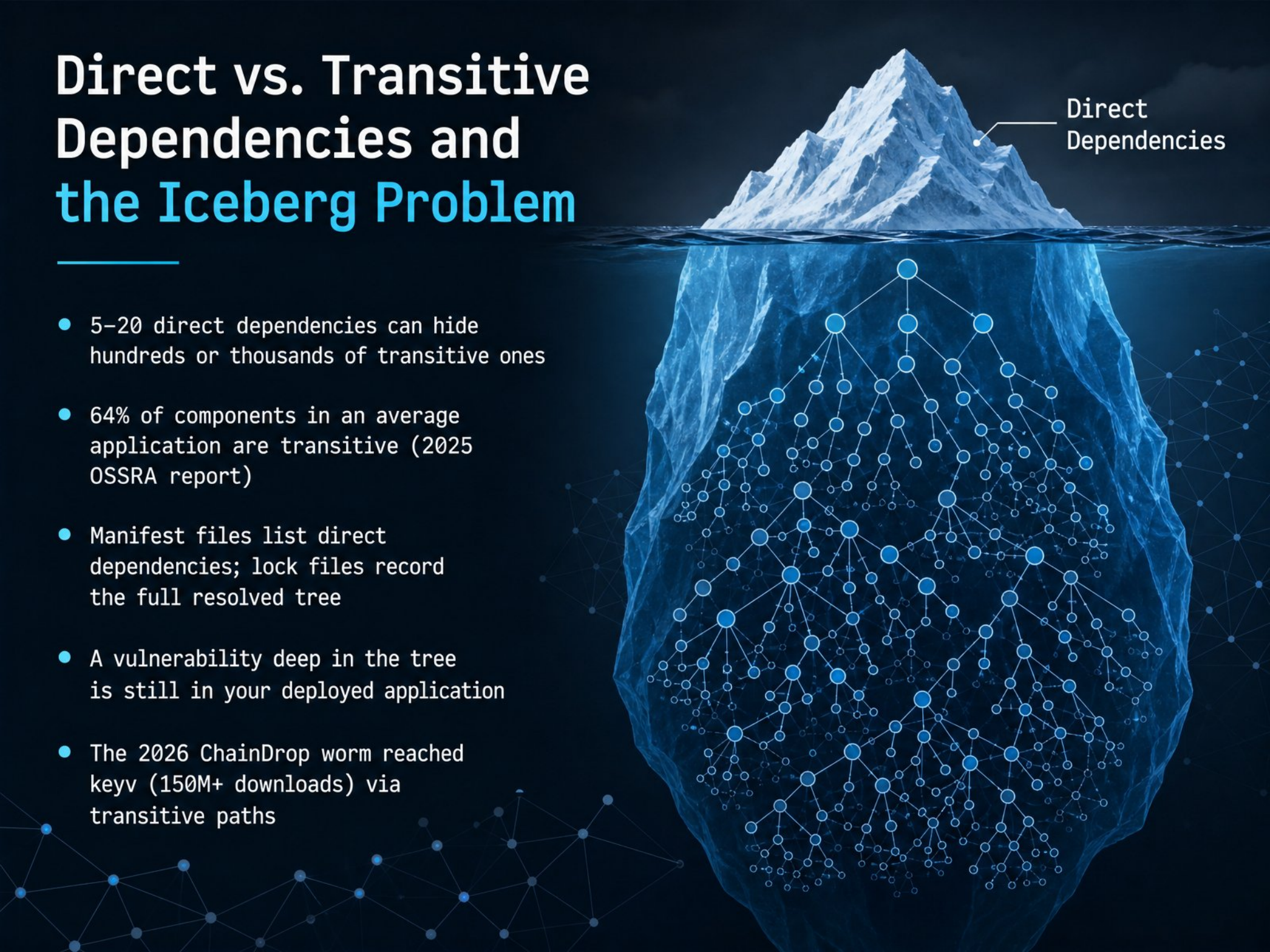
Packages, Dependencies, and the Dependency Graph

- Packages, libraries, modules, frameworks — terminology varies across npm, pip, Maven, and Cargo
- Direct vs. transitive dependencies: your manifest lists 21 packages, but install can pull 1,000+
- The dependency graph: nodes are packages, edges are requirements — trace why any transitive appeared
- Resolution is NP-complete in general; the diamond dependency problem makes it fail

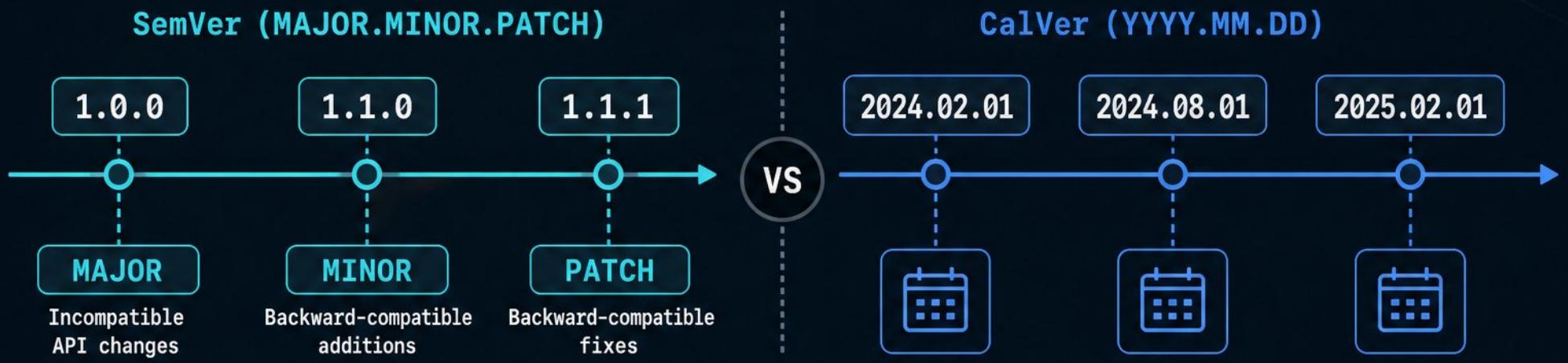


Direct vs. Transitive Dependencies and the Iceberg Problem

- 5–20 direct dependencies can hide hundreds or thousands of transitive ones
- 64% of components in an average application are transitive (2025 OSSRA report)
- Manifest files list direct dependencies; lock files record the full resolved tree
- A vulnerability deep in the tree is still in your deployed application
- The 2026 ChainDrop worm reached keyv (150M+ downloads) via transitive paths



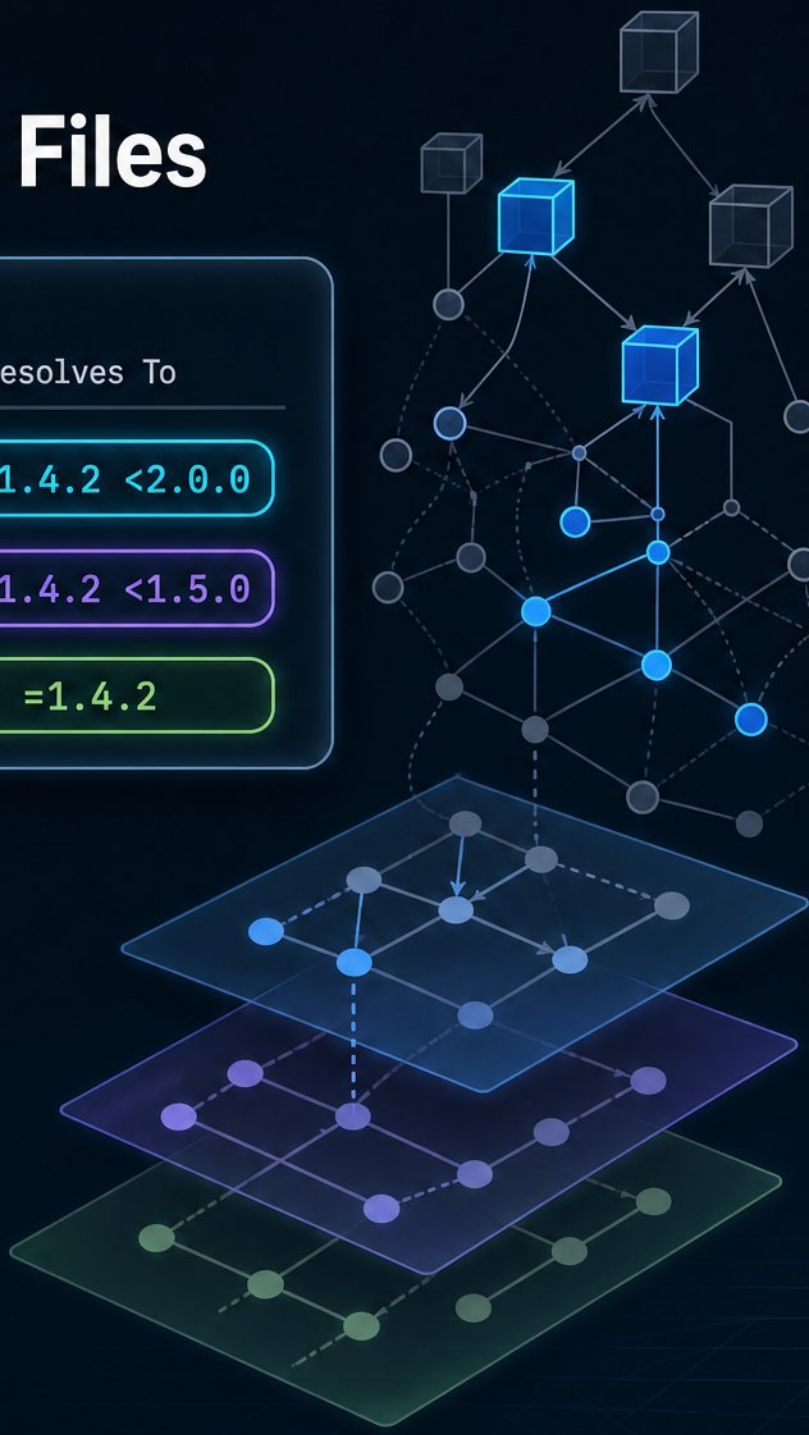
Versioning Schemes: SemVer, CalVer, and Ecosystem Variants



- SemVer 2.0.0: MAJOR.MINOR.PATCH — bump MAJOR for incompatible API changes
- CalVer ties versions to dates: Ubuntu 24.04, Python PEP 2026 proposing 3.YY.0
- SemVer breaks when maintainers disagree on "compatible" — Babel 8, Puppeteer 25
- Ecosystem variants: Python PEP 440 (~=), npm pre-release exclusions, Maven ranges

Declaring Requirements: Ranges, Pinning, and Lock Files

Operator & Example	Meaning	Resolves To
^ "^1.4.2"	Caret (compatible releases)	>=1.4.2 <2.0.0
~ "~1.4.2"	Tilde (patch-level only)	>=1.4.2 <1.5.0
= "1.4.2"	Exact (pin to this version)	=1.4.2



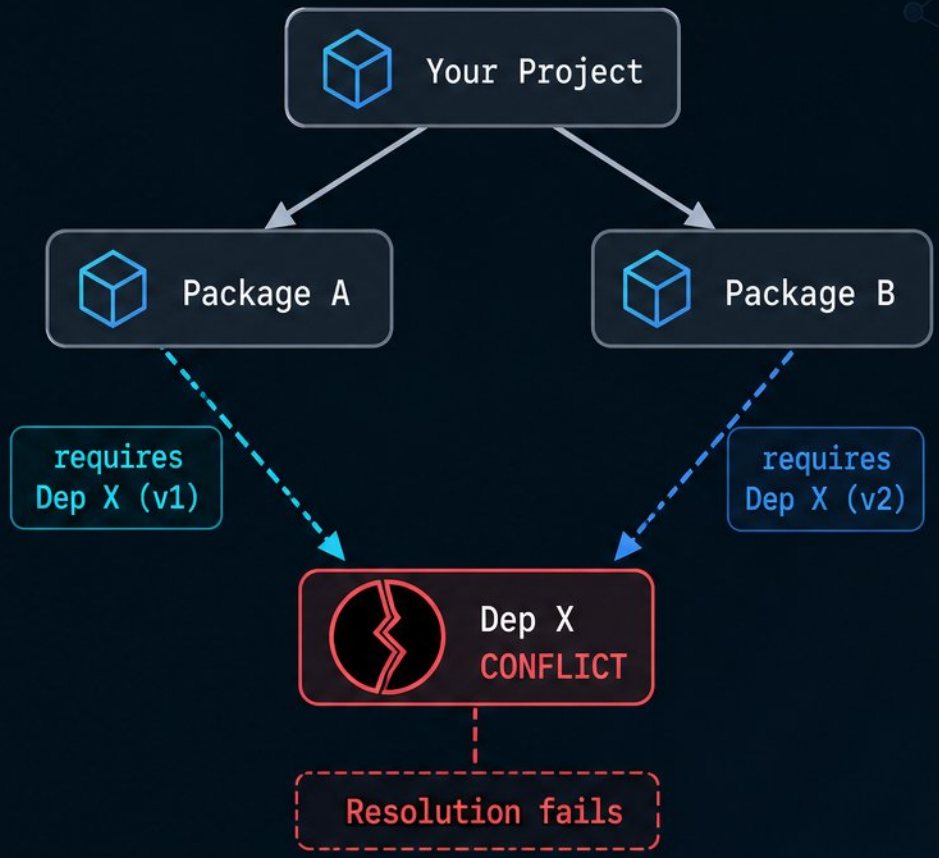
- Caret (`^`) expands to compatible releases; tilde (`~`) restricts to patch-level only
- `^0.2.3` behaves as `>=0.2.3 <0.3.0`: MAJOR-zero is treated as unstable
- Manifest files declare intent; lock files record the exact resolved tree
- Applications: commit lock files, consider exact pins for build tooling
- Libraries: use wide ranges to avoid forcing downstream conflicts

What Is a Breaking Change?

- - API signature changes, behavioral shifts, removed functions, or environment bumps
- - SemVer is a social contract — maintainers often disagree on "compatible"
- - Webpack 5.109 changed CSS/HTML/TS handling in a minor release
- - MUI resolved type-only breaking changes via module augmentation in a minor version
- - Assess risk: read changelogs, run tests, check transitive impact, gate with CI

How Dependency Resolution Works — and Why It Sometimes Fails

- **Resolution:** turning version constraints into a concrete, conflict-free install plan
- The **diamond dependency problem:** two packages demand incompatible transitive versions
- npm's **ERESOLVE** and pip's **ResolutionImpossible** pinpoint the conflicting packages
- **Fix properly:** align versions, or use overrides/resolutions for transitive control



Analyze — Identify conflicting constraints → **Align** — Choose compatible versions → **Resolve** — Apply overrides/ resolutions if needed → **Install** — Conflict-free install plan

Peer Dependencies and the ERESOLVE Error Deep Dive

- Peer dependencies are compatibility contracts between a library and its host application.
- npm 7+ makes unsatisfied peers a hard error to catch conflicts at install time.
- Trace the chain with `npm explain <pkg>` and `npm outdated <pkg>`.
- Upgrade the narrow package or use `overrides` in root `package.json`.
- `--legacy-peer-deps` is only acceptable for known-harmless dev tools.

```
npm ERR! code ERESOLVE
npm ERR! ERESOLVE unable to resolve dependency tree
npm ERR!
npm ERR! While resolving: host-app@1.0.0
npm ERR! Found: react@18.x
npm ERR! node_modules/react
npm ERR!   react@"18.x" from the root project

npm ERR! Could not resolve dependency:
npm ERR! peer react@"^16.8.0 || ^17.0.0" from narrow-lib@1.2.3
npm ERR! node_modules/narrow-lib
npm ERR!   narrow-lib@"1.2.3" from the root project
```

Offending Package (nested dependency)

```
{
  "name": "narrow-lib",
  "version": "1.2.3",
  "peerDependencies": {
    "react": "^16.8.0 || ^17.0.0"
  },
  "dependencies": {
    "other-dep": "^1.0.0"
  }
}
```

Host Application (root project)

```
{
  "name": "host-app",
  "version": "1.0.0",
  "dependencies": {
    "react": "18.x",
    "narrow-lib": "1.2.3"
  },
  "overrides": {
    // optional override example
    // "narrow-lib": { "react": "^18.0.0" }
  }
}
```

Practical Conflict Resolution: npm, pip, and Beyond

npm

```
> npm explain conflict-pkg
  app@1.0.0
  └─ package-a@*
  │  └─ conflict-pkg@^2.0.0
  └─ package-b@*
     └─ conflict-pkg@^1.0.0
-------------------------------------
> npm ls conflict-pkg
  app@1.0.0
  └─ conflict-pkg@2.0.0  (via package-a)
  └─ conflict-pkg@1.0.0  (via package-b)
-------------------------------------
> npm pkg set overrides.conflict-pkg='^2.0.0"
  # Overrides pin transitive versions.
```

pip

```
> pipdeptree
app
├─ package-a
│  └─ conflict_pkg>=2.0
└─ package-b
   └─ conflict_pkg>=1.0,<2.0
Warning: conflicting dependency requirements
-----------------------------------------
Recommended practices
> python -m venv .venv   # clean environment
> source .venv/bin/activate
> pip install -r requirements.txt
> pipdeptree             # visualize
```

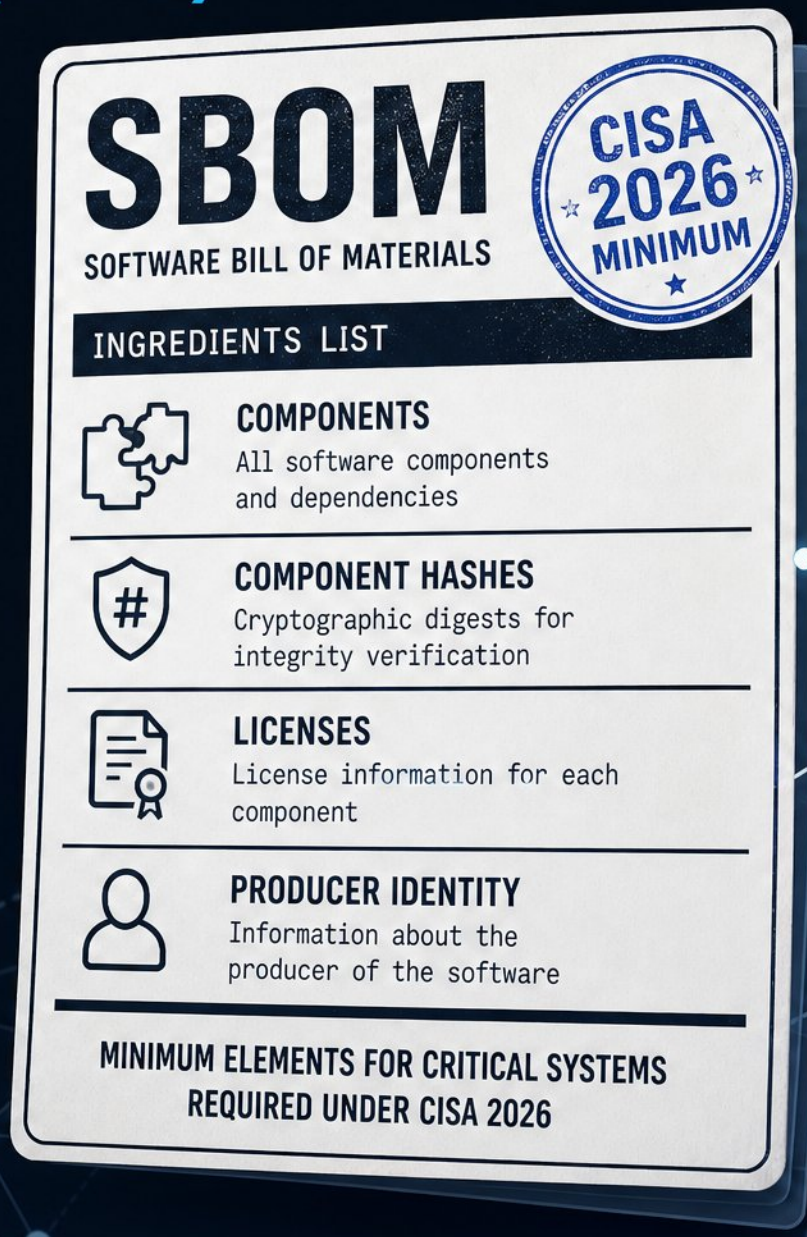
- npm: `explain` traces chains; `ls` shows versions; `overrides` pins transitive versions.
- pip: clean venv + `pipdeptree` to visualize; prefer `>=` ranges over `==` pins.
- Constraints set global limits; lock files pin exact versions. Use both for apps.
- Best diagnostic: always reproduce in a clean environment and read the error completely.

Security, Supply-Chain Attacks, and Dependency Risk

- Transitive dependencies silently expand your attack surface
- Dependency confusion and typosquatting attacks are now fully industrialized
- Maintainer account takeovers can compromise millions of downstream projects
- CI/CD hijacking can publish malicious packages with valid provenance
- License obligations (GPL, AGPL) can force open-sourcing your proprietary code

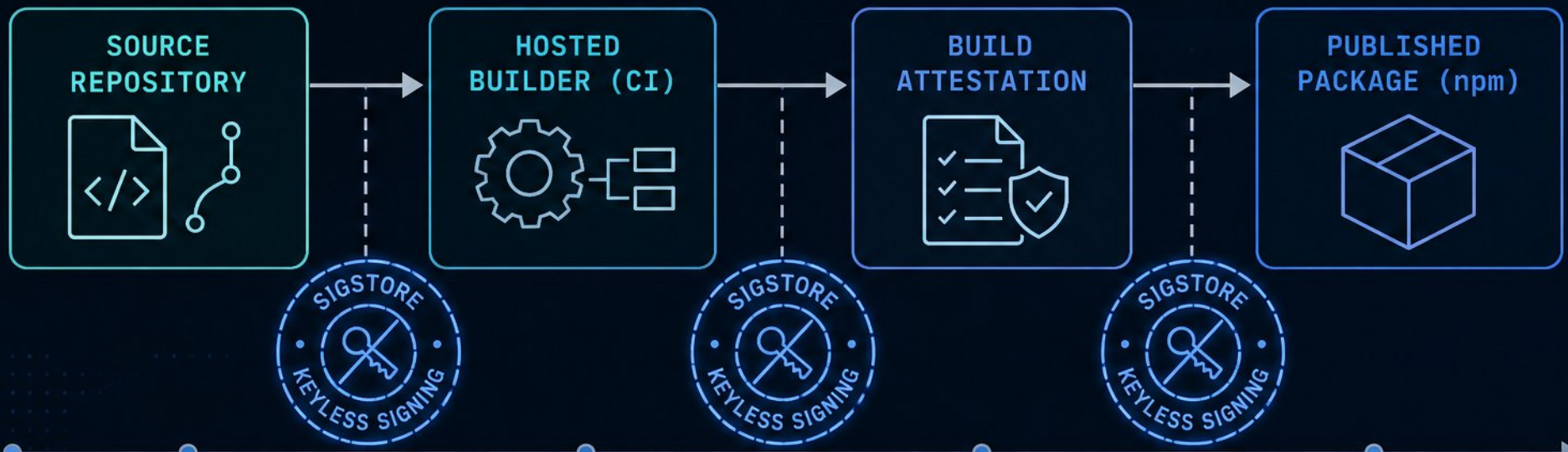
Auditing Tools and the Software Bill of Materials (SBOM)

- SBOM: a formal inventory of all components in your software, now mandatory-like for critical systems under CISA 2026
- SPDX 3.0 and CycloneDX 1.6 are the two dominant machine-readable standards
- Scan with `npm audit`, `pip-audit`, `osv-scanner`, or automated Dependabot/Renovate
- SBOM gives you the ingredients list, but signing and provenance verify its accuracy



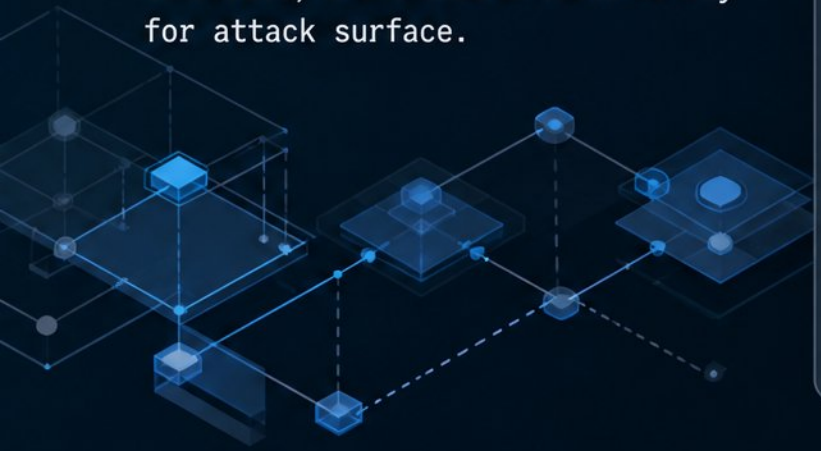
SLSA, Sigstore, and Proving What Was Built

- SLSA levels range from basic provenance to hermetic, two-person-reviewed builds
- Sigstore uses keyless signing tied to CI identities, not long-lived keys
- npm provenance links packages to source via verifiable build attestations
- Shai-Hulud worm proved valid provenance doesn't guarantee a clean pipeline



Keeping Dependencies Healthy: Automation and Strategy

- – Automate updates with Dependabot (GitHub-native) or Renovate (multi-platform, fine-grained grouping).
- – Avoid PR fatigue: group minor/patch updates weekly; fast-track security patches separately.
- – Target fewer than 5 dependency PRs per week in steady state; tighten config if exceeded.
- – Shrink your footprint: remove unused deps, prefer well-maintained libraries, and evaluate build-vs-buy for attack surface.



AUTOMATION CONTROL PANEL

Dependabot
(GitHub-native)

Renovate
(multi-platform, fine-grained grouping)

BATCH GROUPING

Group minor/patch updates

Security patches

Processed separately

WEEKLY MERGE CALENDAR

MON TUE WED THU FRI SAT SUN

Group minor/patch updates weekly; fast-track security patches separately.

PRS PER WEEK

Target: fewer than 5 dependency PRs per week in steady state.

Tighten config if exceeded.

AI-Assisted Dependency Management and Future Trends

- AI tools: DepAdvisor, Dependabump, Android Studio's Gemini auto-resolution
- LLM hallucination risk: recommending nonexistent or malware versions
- Grounding in live registry data eliminates hallucinated packages
- 'No Breaking Changes' strategy: similar security at 1/5 the cost of 'Latest'
- Zero-trust future: immutable packages, WASM, and policy-enforced provenance

PersonWise.

PersonWise • AI digital-human course platform

Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/2cb436e3/public