CS 4530: Fundamentals of Software Engineering

Module 16A: Dependency Management

Adeel Bhutta and Mitch Wand (with material by Donald Pinckney)

Khoury College of Computer Sciences

Learning Objectives for this Module

- By the end of this module, you should be able to:
  - Explain why you need dependencies
  - Explain the major risks of dependencies
  - Explain the principles of semantic versioning
  - Explain what a package manager does
  - Understand that different package managers may solve dependencies differently

Software isn't written in a vacuum

- Writing a JS app?
  - you depend on: React, 100s of small JS packages, Node, V8, …
- Writing ML code in Python?
  - You depend on: PyTorch, Numpy, CUDA, C libraries, compilers, …
- And so on for nearly all software

Our context:

- You are writing an application in JS/TS
- You need some services
- Is there a dependency you can use, or should you build your own?

Risks of Dependencies

- You are reliant on the designer's choices
  - (but: they may have done a better job than you would)
- Security risks
- Upstream risks (transitive dependencies)
- May need multiple copies of some dependencies
- How to keep them all up to date??

Dependency Management Isn't Easy

- Too many dependencies to manage manually
  - Often 100+ for JavaScript projects when considering transitive dependencies
- Too frequent dependency updates to apply manually
  - Even though they may be very important, e.g. critical security patches!
- Dependency updates can’t be done in isolation: you may have to update other dependencies to match

We can control the direct dependencies, but not the transitive dependencies

- We declare our immediate dependencies in a *manifest*: eg package.json
- But we don't/can't control our dependencies' dependencies

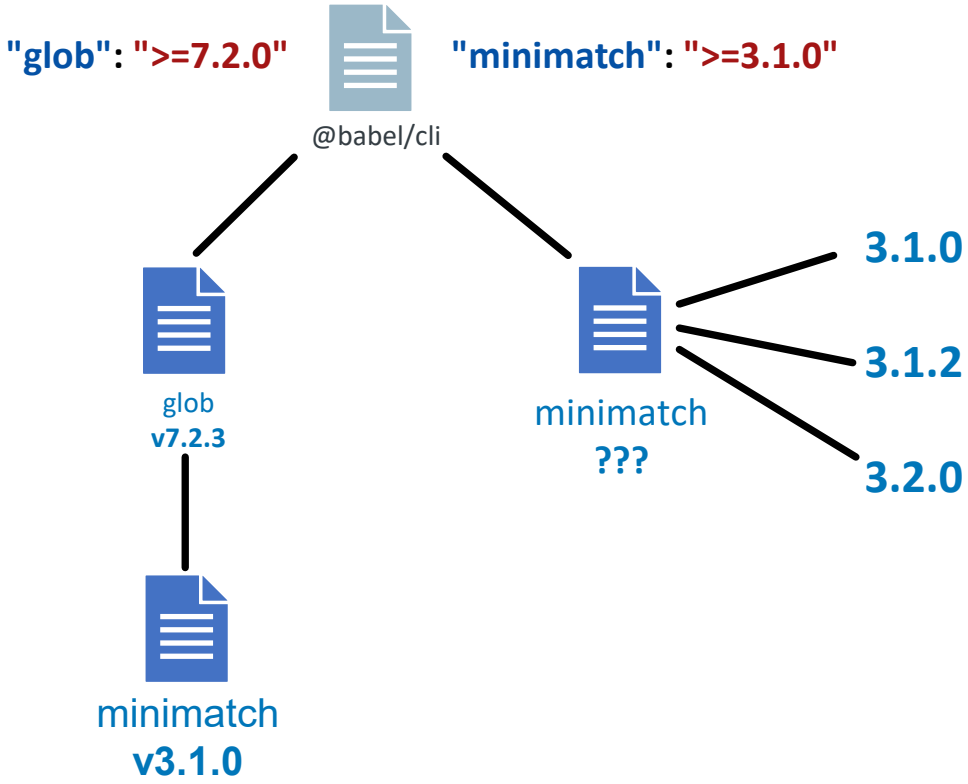
What the *#*!?

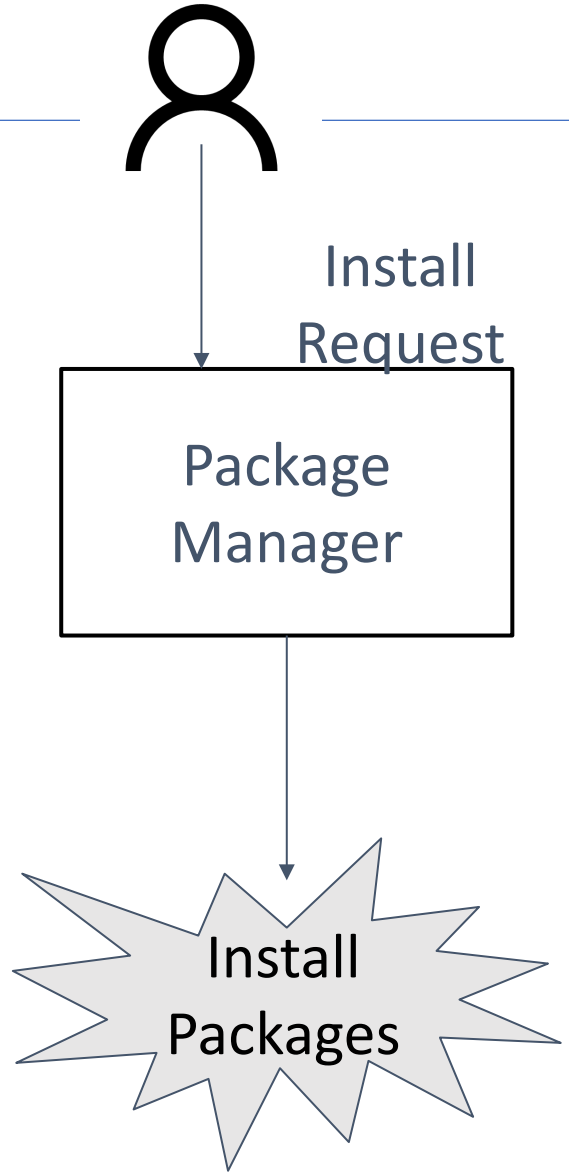
```
$ npm install
npm WARN deprecated inflight@1.0.6: This module is not supported, and leaks memory. Do
not use it. Check out lru-cache if you want a good and tested way to coalesce async
requests by a key value, which is much more comprehensive and powerful.
npm WARN deprecated @humanwhocodes/config-array@0.13.0: Use @eslint/config-array instead
npm WARN deprecated rimraf@3.0.2: Rimraf versions prior to v4 are no longer supported
npm WARN deprecated glob@7.2.3: Glob versions prior to v9 are no longer supported
npm WARN deprecated glob@7.2.3: Glob versions prior to v9 are no longer supported
npm WARN deprecated glob@7.2.3: Glob versions prior to v9 are no longer supported
npm WARN deprecated glob@7.2.3: Glob versions prior to v9 are no longer supported
npm WARN deprecated glob@7.2.3: Glob versions prior to v9 are no longer supported
npm WARN deprecated @humanwhocodes/object-schema@2.0.3: Use @eslint/object-schema instead
npm WARN deprecated eslint@8.57.1: This version is no longer supported. Please see
https://eslint.org/version-support for other options.

added 750 packages, and audited 751 packages in 1m
```

Package Managers Manage the Transitive Dependencies

- NPM: 3 million+ packages
- Complex graph of dependencies
- 20TB+ of package code
- Fairly rich dependency specification language

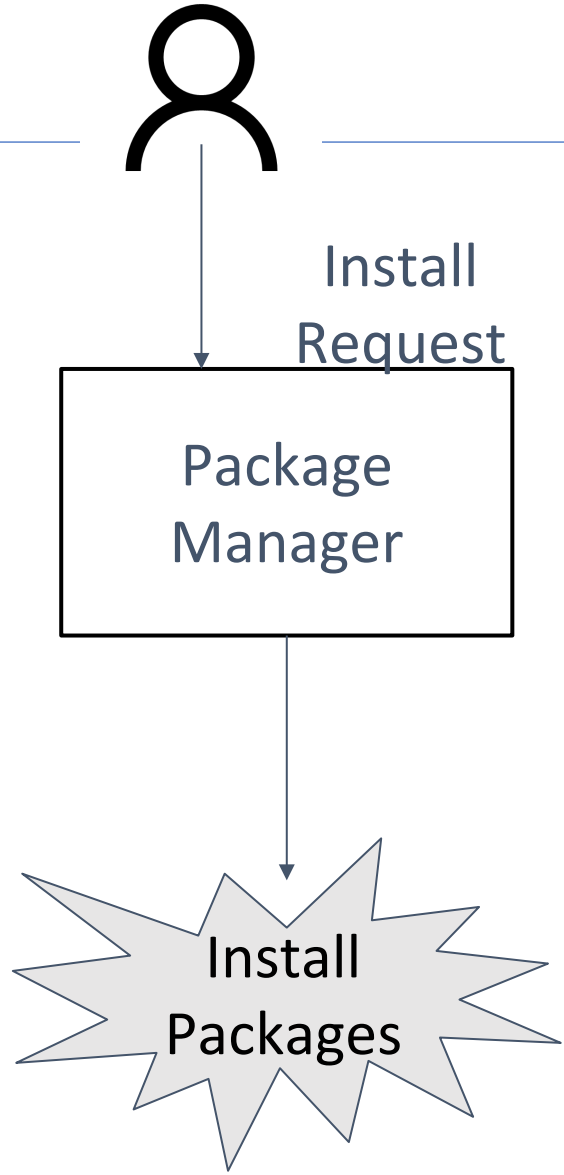
Package Managers Manage the Transitive Dependencies





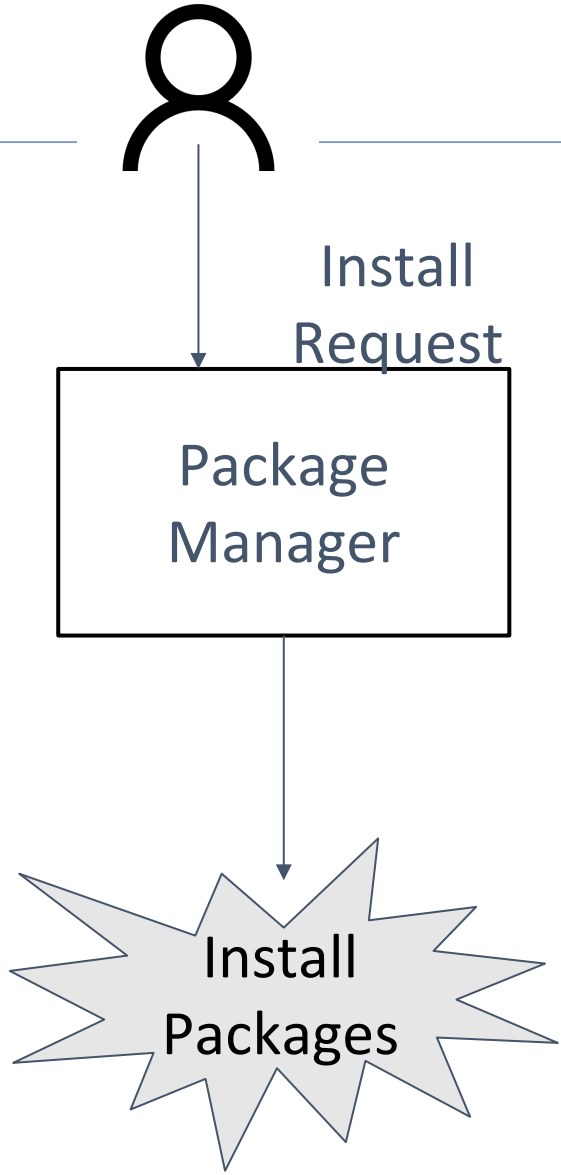
What Can Go Wrong?

- Dependency solving can fail
  - Conflicting constraints
  - Weaknesses in solving algorithms (old Pip, current NPM)



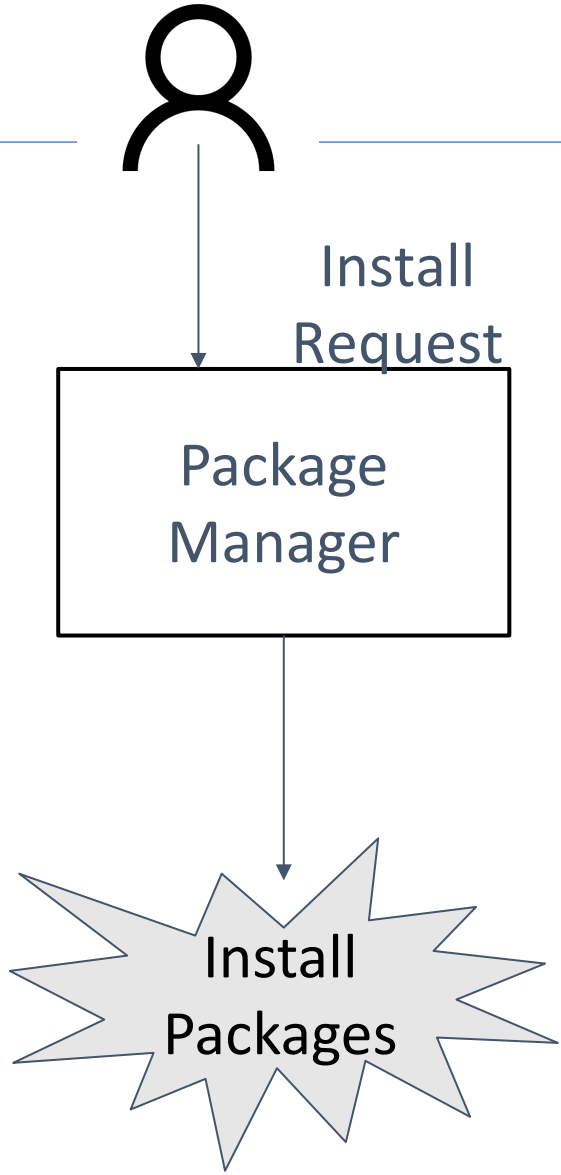
What Can Go Wrong?

- Dependency solving can fail
  - Conflicting constraints
  - Weaknesses in solving algorithms (old Pip, current NPM)
- Dependency solutions can induce code failures
  - Build failures, runtime crashes, runtime bugs, etc.



What Can Go Wrong?

- Dependency solving can fail
  - Conflicting constraints
  - Weaknesses in solving algorithms (old Pip, current NPM)
- Dependency solutions can induce code failures
  - Build failures, runtime crashes, runtime bugs, etc.
- Low-quality dependency solutions
  - Security vulnerabilities
  - Large code size
  - Old versions of packages



Semantic Versioning Can Help Keep Track of Breaking Changes

- Given a version number MAJOR.MINOR.PATCH, increment the:
  - MAJOR version when you make incompatible API changes
  - MINOR version when you add functionality in a backward compatible manner
  - PATCH version when you make backward compatible bug fixes
- Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

https://semver.org/

Semantic Versioning takes effort

"It’s hard to follow semantic versioning—it takes significant effort to make backward-compatible changes, backward-compatible bug fixes, and to backport security patches to old release numbers. However, following semantic versioning is the best way to spread joy to your downstream users."

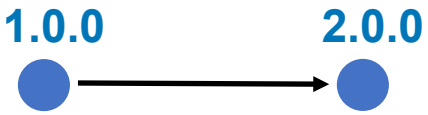
https://semver.org/

Characterizing Updates

Semver Increment

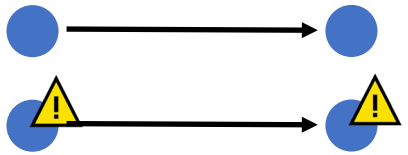
Bug (Patch)

Minor

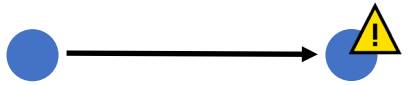


Major

Security Effect



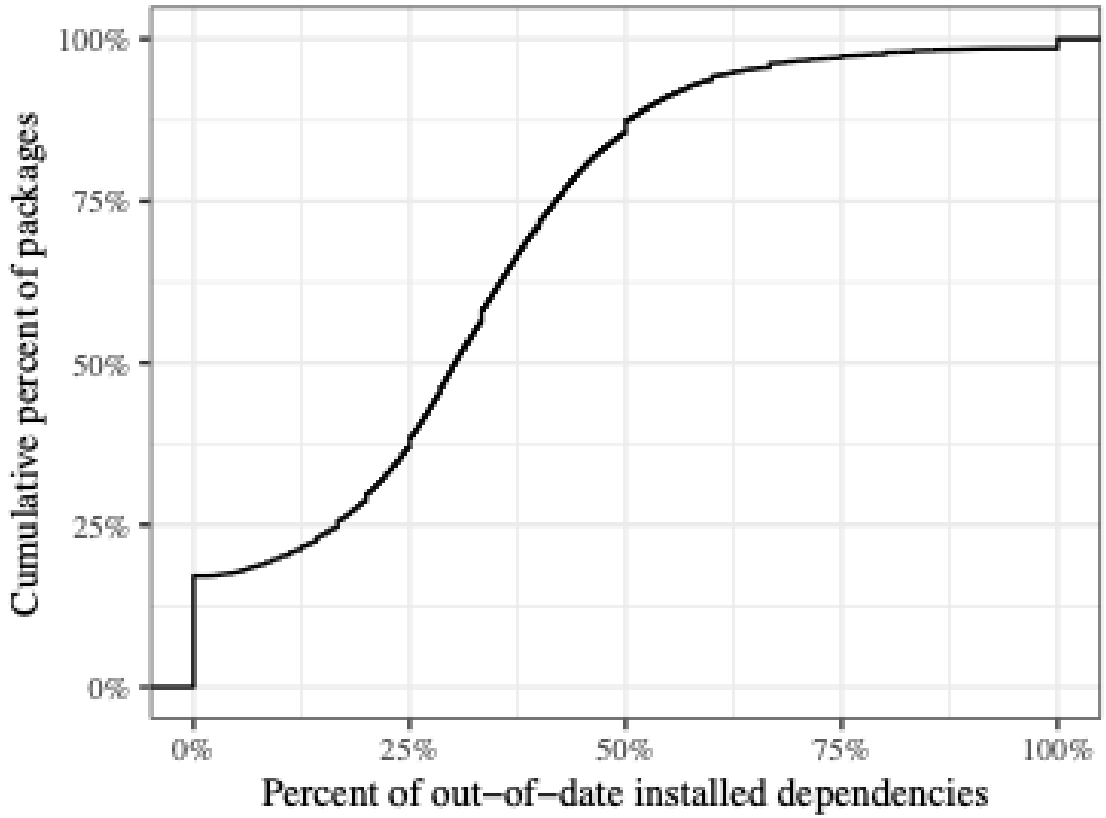
No Effect



Intro Vuln

Patch Vuln

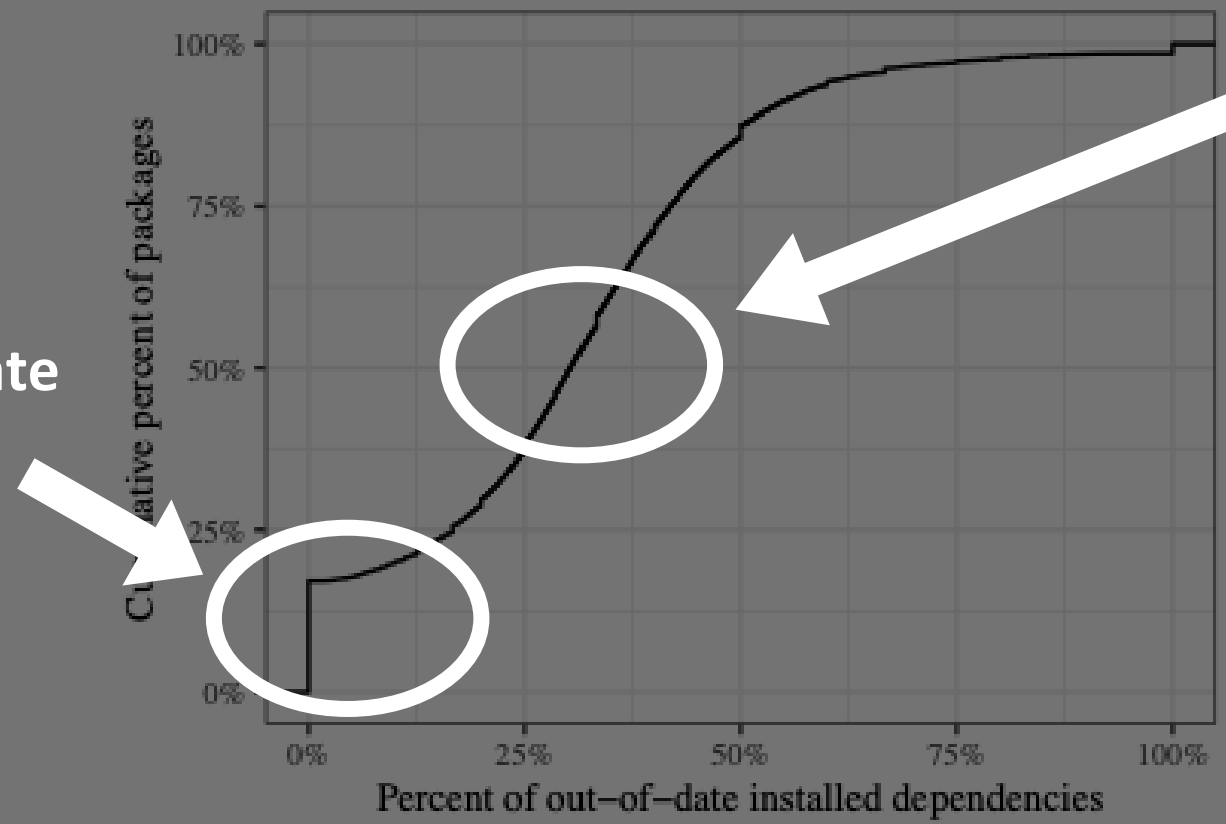
Most Packages Have Out-of-Date Dependencies



Most Packages Have Out-of-Date Dependencies

50% of packages have 31% or more of their dependencies out-of-date

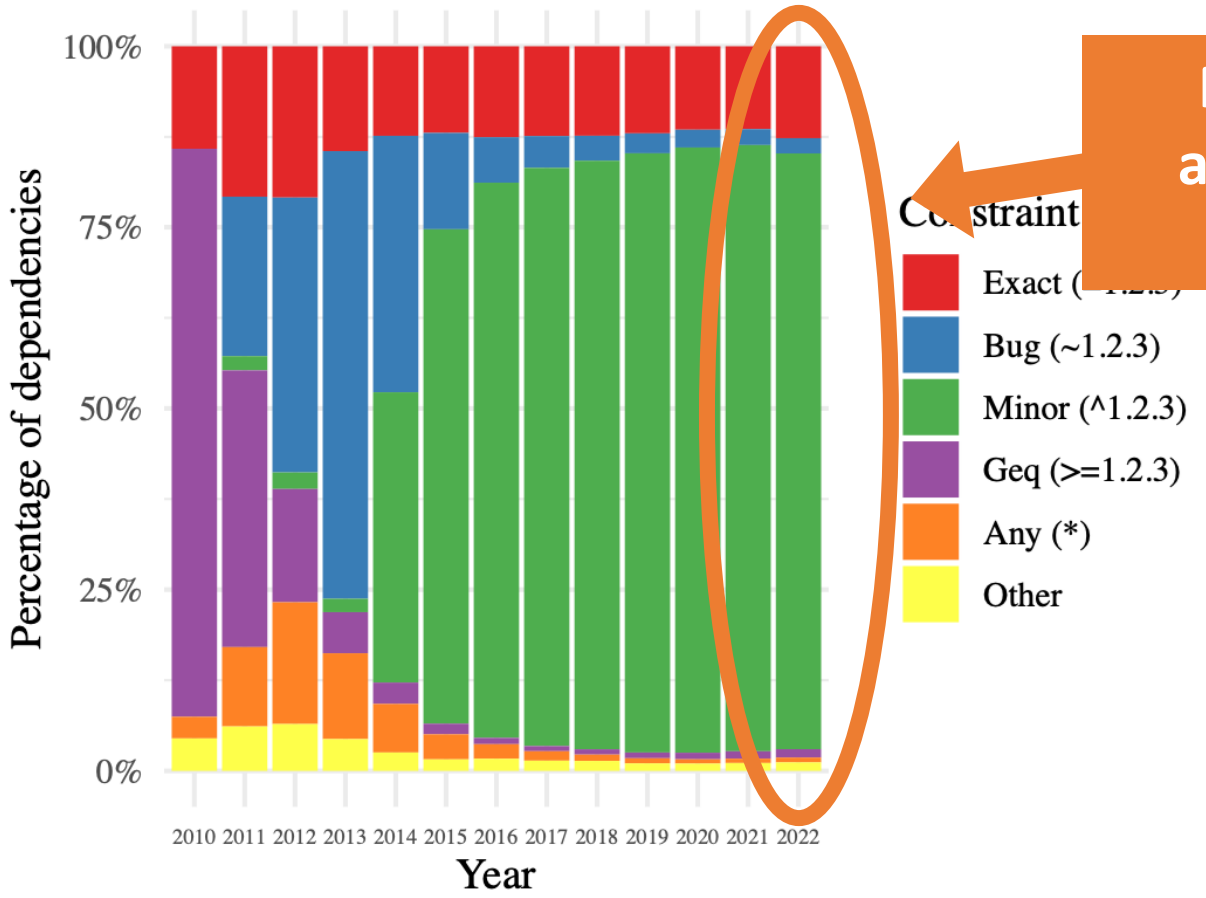
17% of packages have fully up-to-date dependencies



Semantic Versioning starts with your package.json

```
"dependencies": {
    "react": "18.2.0", // Exactly this version
    // Any patch updates (7.2.5, 7.2.6, etc.)
    "protobufjs": "~7.2.5",
    // Any patch or minor updates (4.17.21, 4.18.0, etc.)
    "lodash": "^4.17.21",
    // Arbitrary conjunctions and disjunctions
    "moment": ">=1.0.0 <1.2.0 || ^2.3.1"
}
```

Developers Rarely Distinguish Bug vs. Minor Updates



Most constraints are either exact or minor-flexible

Probably because npm --save defaults to "^"

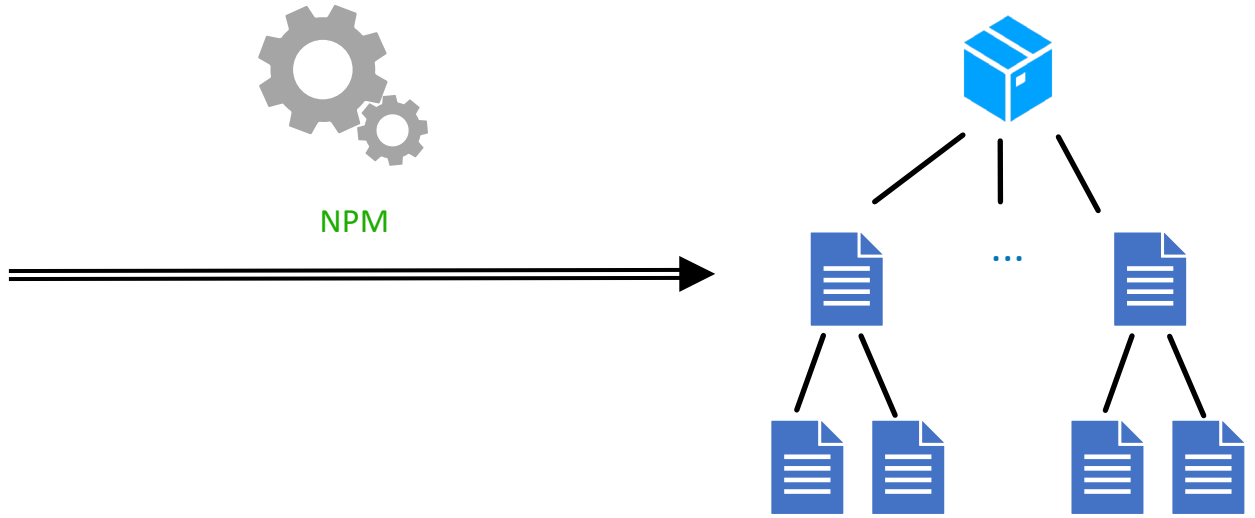
```
"dependencies": {
  "@chakra-ui/next-js": "^2.2.0",
  "@chakra-ui/react": "^2.8.2",
  "@emotion/styled": "^11.11.0",
  "framer-motion": "^11.0.3",
  "jsdoc": "^4.0.2",
  "next": "14.1.0",
  "react": "^18",
  "react-dom": "^18",
  "react-icons": "^5.0.1",
  "what-props-changed": "^1.0.2"
},
```

Implications For Developers & Researchers

- Consider using ~ constraints (bug updates) instead of ^ (bug + minor updates)
  - At the cost of technical lag
  - And forcing the technical lag on clients
- Alternatively, allow developers to specify preferences outside of constraints
  - what would that even mean? 🤔

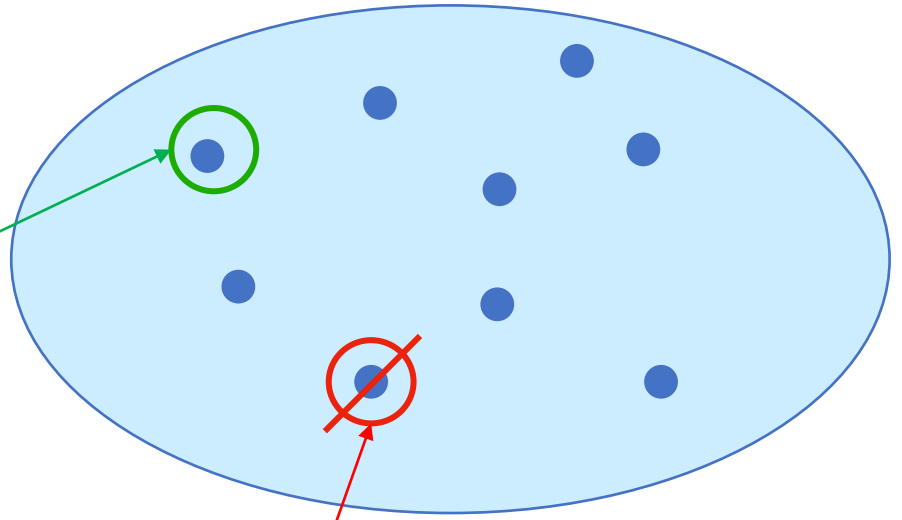
What is dependency solving?

```
"dependencies": {
  "commander": "^2.8.1”,
  "convert-source-map": "^1.1.0",
  "fs-readdir-recursive": "^1.1.0",
  "glob": "^7.0.0",
  "lodash": "^4.17.10",
  "mkdirp": "^0.5.1",
  "output-file-sync": "^2.0.0",
  "slash": "^2.0.0",
  "source-map": "^0.5.0”
}
```



Consider the solution space

```
"dependencies": {
  "commander": "^2.8.1”,
  "convert-source-map": "^1.1.0",
  "fs-readdir-recursive": "^1.1.0",
  "glob": "^7.0.0",
  "lodash": "^4.17.10",
  "mkdirp": "^0.5.1",
  "output-file-sync": "^2.0.0",
  "slash": "^2.0.0",
  "source-map": "^0.5.0”
}
```



```
{"commander" @ "^2.8.1”,
 "convert-source-map": @ "^1.1.1",
  ...
}
```

```
{"commander" @ "^2.8.1”,
 "convert-source-map": @ "^2.1.1",
  ...
}
```

Versions, Constraints, and Constraint Semantics

$$\mathcal{V} := (x\ y\ z) \qquad \text{Version numbers}$$

$$\begin{array}{lll} \mathscr{C} := & (= \ x\ y\ z) & \text{Exact} \\ \mid & (<= \ x\ y\ z) & \text{At most} \\ \mid & (>= \ x\ y\ z) & \text{At least} \\ \mid & (\texttt{and}\ \mathscr{C}_1\ \mathscr{C}_2) & \text{Conjunction} \end{array}$$

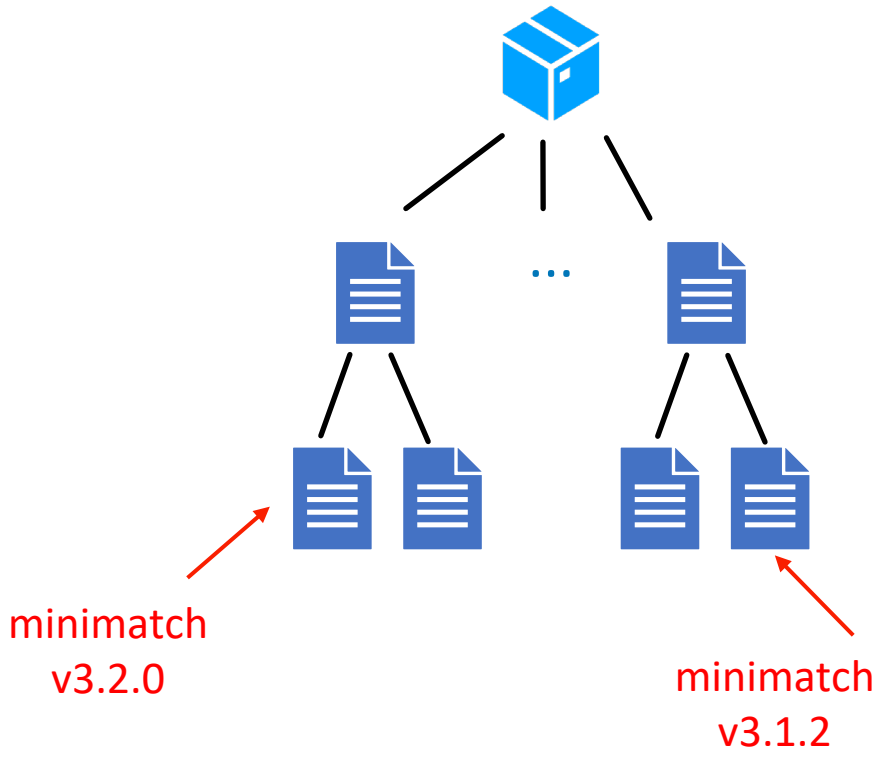
$$sat \ : \ \mathscr{C} \rightarrow \mathcal{V} \rightarrow \textbf{Bool}$$

Does version v satisfy constraint c?

```
(define (sat c v)
  (match `(,v ,c)
    [`((,x ,y ,z) (= ,x ,y ,z))          #true]
    [`((,x ,y ,z1) (<= ,x ,y ,z2))       (<= z1 z2)]
    [`((,x ,y1 ,z1) (<= ,x ,y2 ,z2))     (< y1 y2)]
    [`((,x1 ,y1 ,z1) (<= ,x2 ,y2 ,z2)) (< x1 x2)]
    [`(,_ (and ,c1 ,c2))
      (and (sat c1 v) (sat c2 v))]
    [`((,x1 ,y1, z1) (>= ,x2 ,y2 ,z2))
      (sat `(,x2 ,y2 ,z2) `(<= ,x1 ,y1 ,z1))]
    [_                                   #false]))
```

Are Multiple Versions of a Package Allowed?

- NPM: Yes
- PIP: No
- Cargo: Partially



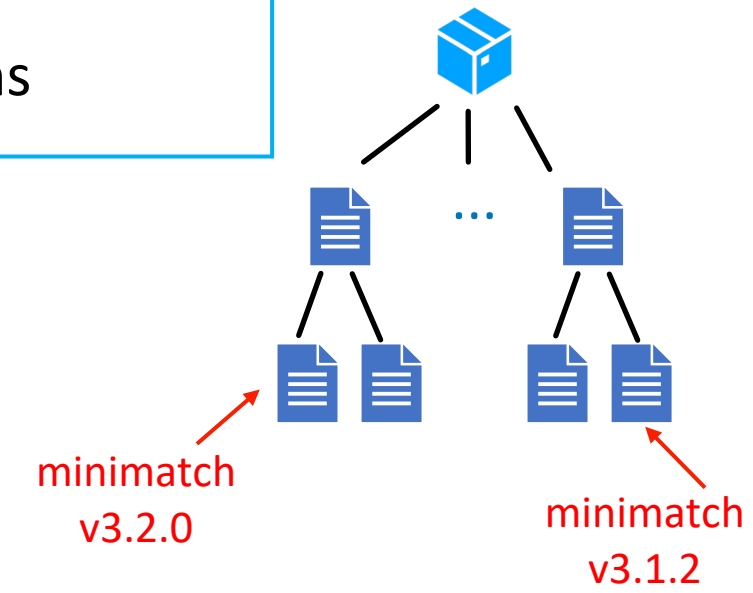
27

If multiple versions are allowed, which ones are consistent with each other?

$$consistent \quad : \quad \mathcal{V} \rightarrow \mathcal{V} \rightarrow \mathbf{Bool}$$

Different package managers may allow different combinations

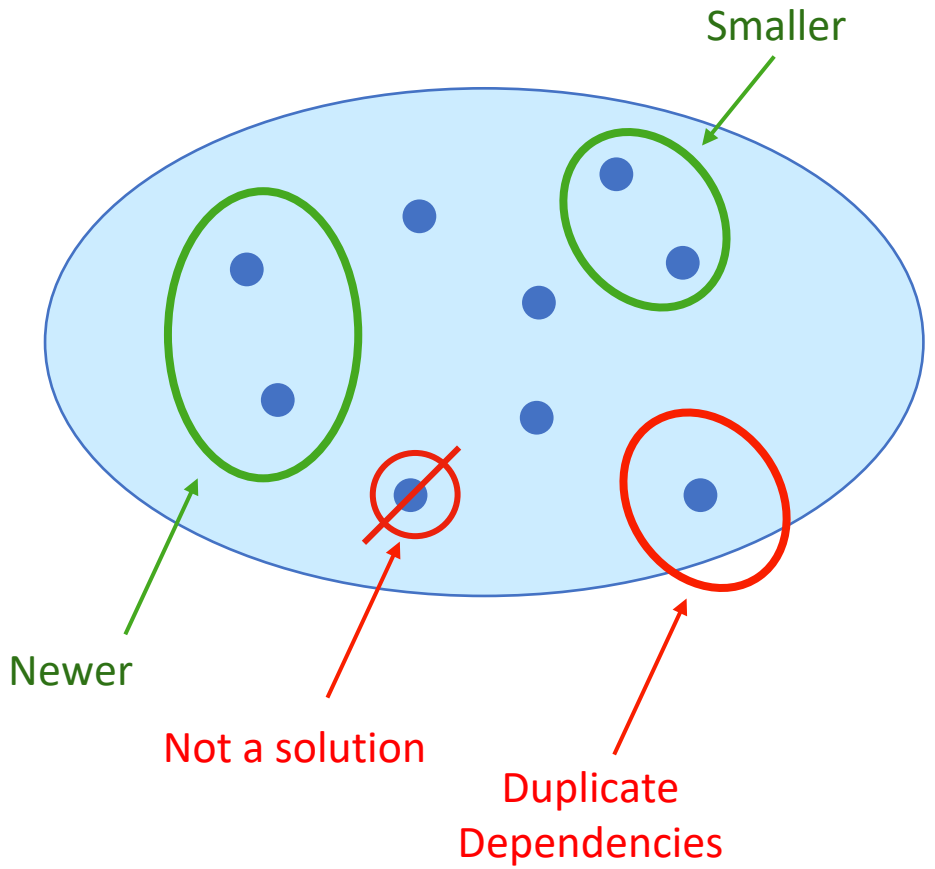
```
(define (npm-consistent v1 v2)
  #true)

(define (pip-consistent v1 v2)
  (equal? v1 v2))

(define (cargo-consistent v1 v2)
  (match `(,v1 ,v2)
    [`((0 0 ,z1) (0 0 ,z2))       #true]
    [`((0 ,y ,z1) (0 ,y ,z2))     (= z1 z2)]
    [`((0 ,y1 ,z1) (0 ,y2 ,z2))   #true]
    [`((x ,y1 ,z1) (x ,y2 ,z2))   (and (= y1 y2) (= z1 z2))]
    [_                            #true]))
```



28

If there are many possible solutions, which one should we choose?

```
"dependencies": {
  "commander": "^2.8.1”,
  "convert-source-map": "^1.1.0",
  "fs-readdir-recursive": "^1.1.0",
  "glob": "^7.0.0",
  "lodash": "^4.17.10",
  "mkdirp": "^0.5.1",
  "output-file-sync": "^2.0.0",
  "slash": "^2.0.0",
  "source-map": "^0.5.0”
}
```



$$minGoal \quad : \quad \mathcal{G} \rightarrow \mathbb{R}^n$$

So this is an optimization problem!

Minimize # of Dependencies

```
(define (minGoal-num-deps g)
  (length (graph-nodes g)))
```

Different package managers may have different optimization goals

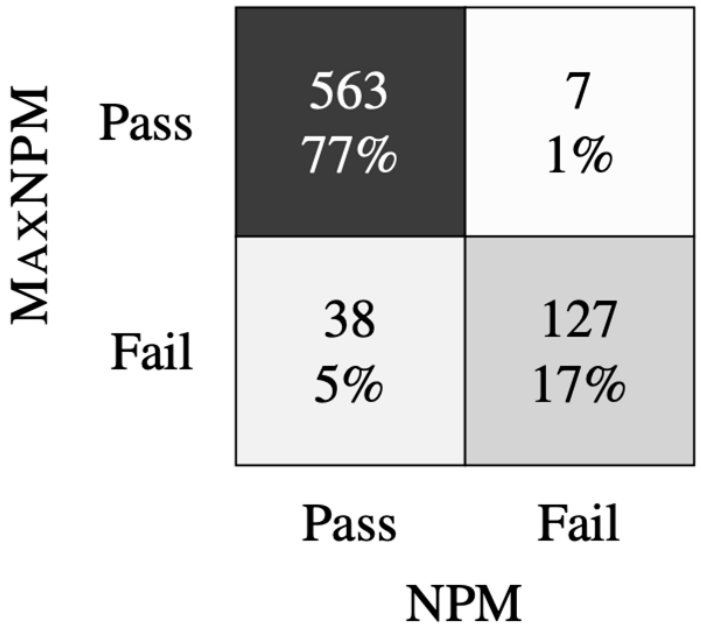
Prefer Newer Versions

```
(define (minGoal-oldness g)
  (apply +
    (map
      (lambda (n)
        (get-oldness
          (node-package n)
          (node-version n)))
      (graph-nodes g))))

(define (get-oldness p v)
  ; The get-sorted-versions retrieve
  ; a list of all versions of p
  (define all-vs
    (get-sorted-versions p))
  (if (= (length all-vs) 1)
      0
      (/ (index-of all-vs v)
          (sub1 (length all-vs)))))
```

Tunable knobs for a package manager

sat	:	$\mathscr{C} \to \mathcal{V} \to \textbf{Bool}$	Constraint satisfaction semantics
$consistent$	:	$\mathcal{V} \to \mathcal{V} \to \textbf{Bool}$	Version consistency versions
$cycles\_ok$	$\in$	**Bool**	If cycles are permitted in solution graphs
$minGoal$	:	$\mathcal{G} \to \mathbb{R}^n$	Objective functions

Luckily, most projects are robust to different dependency solutions



Learning Objectives for this Module

- You should now be able to:
  - Explain why you need dependencies
  - Explain the major risks of dependencies
  - Explain the principles of semantic versioning
  - Explain what a package manager does
  - Understand that different package managers may solve dependencies differently