goto;

GOTO
AARHUS 2022



#GOTOaar

JSFIX - https://jsfix.live

How to Prevent Failures Caused by Breaking Changes in JavaScript Libraries

Benjamin Barslev & Martin Torp - June 16, 2022 GOTO Aarhus

Agenda

- Our background
- Dependency management in npm
- Automating dependency upgrades (using JSFIX)
- Ongoing and future work



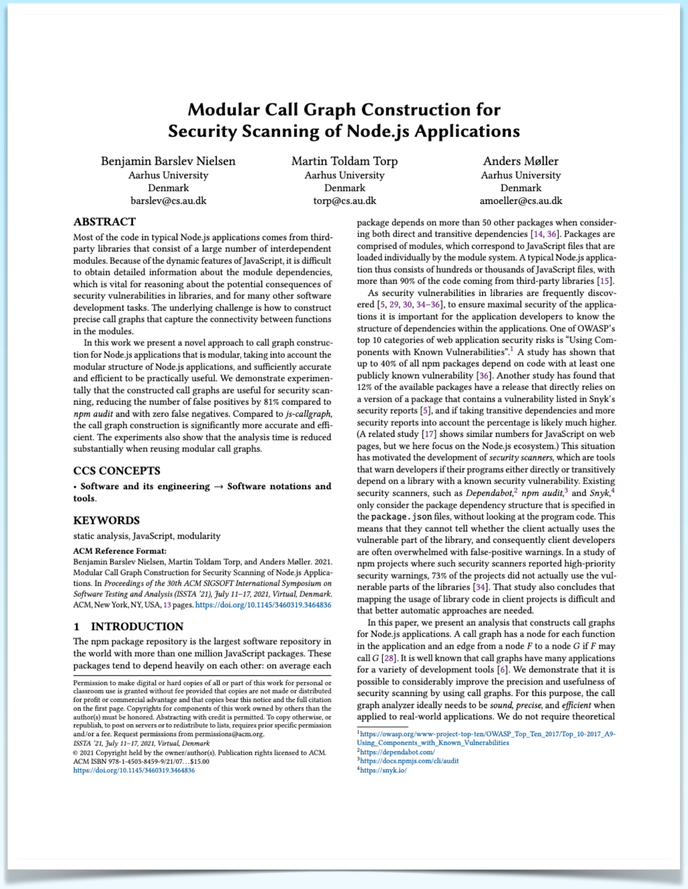
Martin Torp
PhD CS
martin@coana.tech

Benjamin Barslev
PhD CS
barslev@coana.tech



Anders Møller
Professor in CS
amoeller@cs.au.dk

Research background



Modular Call Graph Construction for Security Scanning of Node.js Applications

Benjamin Barslev Nielsen
Aarhus University
Denmark
barslev@cs.au.dk

Martin Toldam Torp
Aarhus University
Denmark
torp@cs.au.dk

Anders Møller
Aarhus University
Denmark
amoeller@cs.au.dk

ABSTRACT

Most of the code in typical Node.js applications comes from third-party libraries that consist of a large number of interdependent modules. Because of the dynamic features of JavaScript, it is difficult to obtain detailed information about the module dependencies, which is vital for reasoning about the potential consequences of security vulnerabilities in libraries, and for many other software development tasks. The underlying challenge is how to construct precise call graphs that capture the connectivity between functions in the modules.

In this work we present a novel approach to call graph construction for Node.js applications that is modular, taking into account the modular structure of Node.js applications, and sufficiently accurate and efficient to be practically useful. We demonstrate experimentally that the constructed call graphs are useful for security scanning, reducing the number of false positives by 81% compared to *npm audit* and with zero false negatives. Compared to *js-callgraph*, the call graph construction is significantly more accurate and efficient. The experiments also show that the analysis time is reduced substantially when reusing modular call graphs.

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**.

KEYWORDS

static analysis, JavaScript, modularity

ACM Reference Format:
Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular Call Graph Construction for Security Scanning of Node.js Applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464836

1 INTRODUCTION

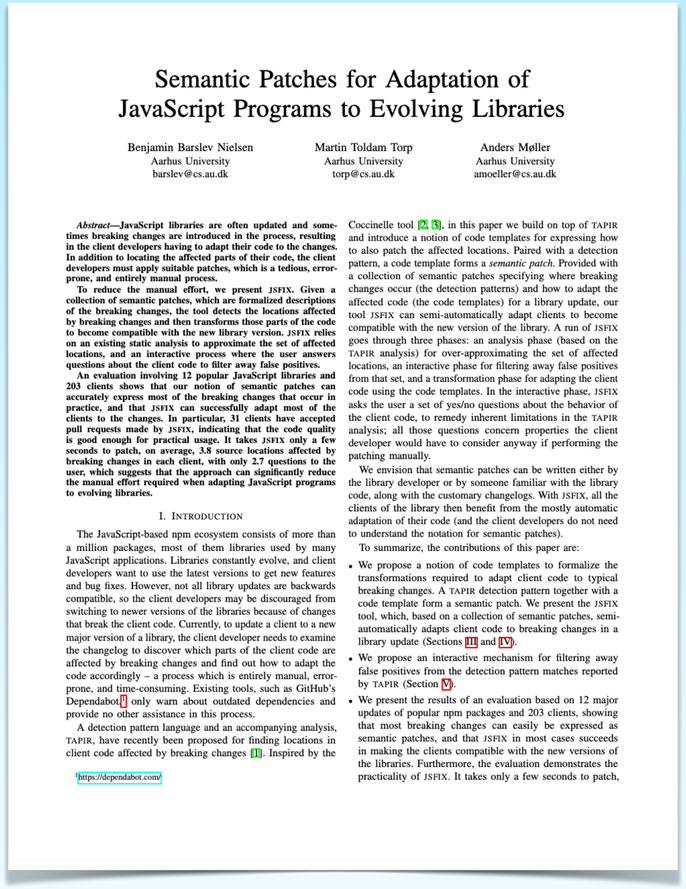
The npm package repository is the largest software repository in the world with more than one million JavaScript packages. These packages tend to depend heavily on each other: on average each package depends on more than 50 other packages when considering both direct and transitive dependencies [14, 36]. Packages are comprised of modules, which correspond to JavaScript files that are loaded individually by the module system. A typical Node.js application thus consists of hundreds or thousands of JavaScript files, with more than 90% of the code coming from third-party libraries [15].

As security vulnerabilities in libraries are frequently discovered [5, 29, 30, 34–36], to ensure maximal security of the applications it is important for the application developers to know the structure of dependencies within the applications. One of OWASP's top 10 categories of web application security risks is "Using Components with Known Vulnerabilities".[1] A study has shown that up to 40% of all npm packages depend on code with at least one publicly known vulnerability [36]. Another study has found that 12% of the available packages have a release that directly relies on a version of a package that contains a vulnerability listed in Snyk's security reports [5], and if taking transitive dependencies and more security reports into account the percentage is likely much higher. (A related study [17] shows similar numbers for JavaScript on web pages, but we here focus on the Node.js ecosystem.) This situation has motivated the development of *security scanners*, which are tools that warn developers if their programs either directly or transitively depend on a library with a known security vulnerability. Existing security scanners, such as *Dependabot*,[2] *npm audit*,[3] and *Snyk*,[4] only consider the package dependency structure that is specified in the `package.json` files, without looking at the program code. This means that they cannot tell whether the client actually uses the vulnerable part of the library, and consequently client developers are often overwhelmed with false-positive warnings. In a study of npm projects where such security scanners reported high-priority security warnings, 73% of the projects did not actually use the vulnerable parts of the libraries [34]. That study also concludes that mapping the usage of library code in client projects is difficult and that better automatic approaches are needed.

In this paper, we present an analysis that constructs call graphs for Node.js applications. A call graph has a node for each function in the application and an edge from a node F to a node G if F may call G [28]. It is well known that call graphs have many applications for a variety of development tools [6]. We demonstrate that it is possible to considerably improve the precision and usefulness of security scanning by using call graphs. For this purpose, the call graph analyzer ideally needs to be *sound*, *precise*, and *efficient* when applied to real-world applications. We do not require theoretical

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '21, July 11–17, 2021, Virtual, Denmark
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464836

[1] https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A9-Using_Components_with_Known_Vulnerabilities
[2] https://dependabot.com/
[3] https://docs.npmjs.com/cli/audit
[4] https://snyk.io/



Semantic Patches for Adaptation of JavaScript Programs to Evolving Libraries

Benjamin Barslev Nielsen
Aarhus University
barslev@cs.au.dk

Martin Toldam Torp
Aarhus University
torp@cs.au.dk

Anders Møller
Aarhus University
amoeller@cs.au.dk

***Abstract*—JavaScript libraries are often updated and sometimes breaking changes are introduced in the process, resulting in the client developers having to adapt their code to the changes. In addition to locating the affected parts of their code, the client developers must apply suitable patches, which is a tedious, error-prone, and entirely manual process.**

To reduce the manual effort, we present JSFIX. Given a collection of semantic patches, which are formalized descriptions of the breaking changes, the tool detects the locations affected by breaking changes and then transforms those parts of the code to become compatible with the new library version. JSFIX relies on an existing static analysis to approximate the set of affected locations, and an interactive process where the user answers questions about the client code to filter away false positives.

An evaluation involving 12 popular JavaScript libraries and 203 clients shows that our notion of semantic patches can accurately express most of the breaking changes that occur in practice, and that JSFIX can successfully adapt most of the clients to the changes. In particular, 31 clients have accepted pull requests made by JSFIX, indicating that the code quality is good enough for practical usage. It takes JSFIX only a few seconds to patch, on average, 3.8 source locations affected by breaking changes in each client, with only 2.7 questions to the user, which suggests that the approach can significantly reduce the manual effort required when adapting JavaScript programs to evolving libraries.

I. INTRODUCTION

The JavaScript-based npm ecosystem consists of more than a million packages, most of them libraries used by many JavaScript applications. Libraries constantly evolve, and client developers want to use the latest versions to get new features and bug fixes. However, not all library updates are backwards compatible, so the client developers may be discouraged from switching to newer versions of the libraries because of changes that break the client code. Currently, to update a client to a new major version of a library, the client developer needs to examine the changelog to discover which parts of the client code are affected by breaking changes and find out how to adapt the code accordingly – a process which is entirely manual, error-prone, and time-consuming. Existing tools, such as GitHub's Dependabot,[1] only warn about outdated dependencies and provide no other assistance in this process.

A detection pattern language and an accompanying analysis, TAPIR, have recently been proposed for finding locations in client code affected by breaking changes [1]. Inspired by the Coccinelle tool [2, 3], in this paper we build on top of TAPIR and introduce a notion of code templates for expressing how to also patch the affected locations. Paired with a detection pattern, a code template forms a *semantic patch*. Provided with a collection of semantic patches specifying where breaking changes occur (the detection patterns) and how to adapt the affected code (the code templates) for a library update, our tool JSFIX can semi-automatically adapt clients to become compatible with the new version of the library. A run of JSFIX goes through three phases: an analysis phase (based on the TAPIR analysis) for over-approximating the set of affected locations, an interactive phase for filtering away false positives from that set, and a transformation phase for adapting the client code using the code templates. In the interactive phase, JSFIX asks the user a set of yes/no questions about the behavior of the client code, to remedy inherent limitations in the TAPIR analysis; all those questions concern properties the client developer would have to consider anyway if performing the patching manually.

We envision that semantic patches can be written either by the library developer or by someone familiar with the library code, along with the customary changelogs. With JSFIX, all the clients of the library then benefit from the mostly automatic adaptation of their code (and the client developers do not need to understand the notation for semantic patches).

To summarize, the contributions of this paper are:

- We propose a notion of code templates to formalize the transformations required to adapt client code to typical breaking changes. A TAPIR detection pattern together with a code template form a semantic patch. We present the JSFIX tool, which, based on a collection of semantic patches, semi-automatically adapts client code to breaking changes in a library update (Sections III and IV).
- We propose an interactive mechanism for filtering away false positives from the detection pattern matches reported by TAPIR (Section V).
- We present the results of an evaluation based on 12 major updates of popular npm packages and 203 clients, showing that most breaking changes can easily be expressed as semantic patches, and that JSFIX in most cases succeeds in making the clients compatible with the new versions of the libraries. Furthermore, the evaluation demonstrates the practicality of JSFIX. It takes only a few seconds to patch,

[1] https://dependabot.com/



Detecting Locations in JavaScript Programs Affected by Breaking Library Changes

ANDERS MØLLER, Aarhus University, Denmark
BENJAMIN BARSLEV NIELSEN, Aarhus University, Denmark
MARTIN TOLDAM TORP, Aarhus University, Denmark

187

JavaScript libraries are widely used and evolve rapidly. When adapting client code to non-backwards compatible changes in libraries, a major challenge is how to locate affected API uses in client code, which is currently a difficult manual task. In this paper we address this challenge by introducing a simple pattern language for expressing API access points and a pattern-matching tool based on lightweight static analysis.

Experimental evaluation on 15 popular npm packages shows that typical breaking changes are easy to express as patterns. Running the static analysis on 265 clients of these packages shows that it is accurate and efficient: it reveals usages of breaking APIs with only 14% false positives and no false negatives, and takes less than a second per client on average. In addition, the analysis is able to report its confidence, which makes it easier to identify the false positives. These results suggest that the approach, despite its simplicity, can reduce the manual effort of the client developers.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**.

Additional Key Words and Phrases: software evolution, software maintenance, breaking changes

ACM Reference Format:
Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting Locations in JavaScript Programs Affected by Breaking Library Changes. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 187 (November 2020), 25 pages. https://doi.org/10.1145/3428255

1 INTRODUCTION

Modern JavaScript applications heavily rely on third-party libraries. The npm registry contains more than 1.2 million packages,[1] and an average package depends on around 80 other packages [Zimmermann et al. 2019]. Many libraries, especially the most widely used ones, are frequently updated. New features are added, security flaws and other bugs are fixed, and outdated functionality is removed. The npm system uses semantic versioning,[2] which is a version numbering scheme that distinguishes between major updates that may contain breaking changes and minor and patch updates that usually can be adapted immediately. For most libraries, a changelog is maintained, documenting the changes, especially the ones that may require attention from the client developers.

Previous work has shown that there is typically a considerable delay, called a technical lag, between a release of a new version of a library and the corresponding update of a client [Zerouali et al. 2019]. Client developers are of course interested in the updates, especially the security-related

[1] http://modulecounts.com/ (April 2020)
[2] http://semver.org/

Authors' addresses: Anders Møller, Aarhus University, Denmark, amoeller@cs.au.dk; Benjamin Barslev Nielsen, Aarhus University, Denmark, barslev@cs.au.dk; Martin Toldam Torp, Aarhus University, Denmark, torp@cs.au.dk.

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/11-ART187
https://doi.org/10.1145/3428255

Proc. ACM Program. Lang., Vol. 4, No. OOPSLA, Article 187. Publication date: November 2020.

CENTER FOR ADVANCED SOFTWARE ANALYSIS

http://casa.au.dk/

AARHUS UNIVERSITET

From research to startup

The Coana startup

- Funded by Innovationsfonden and European Research Council
- 6 student developers

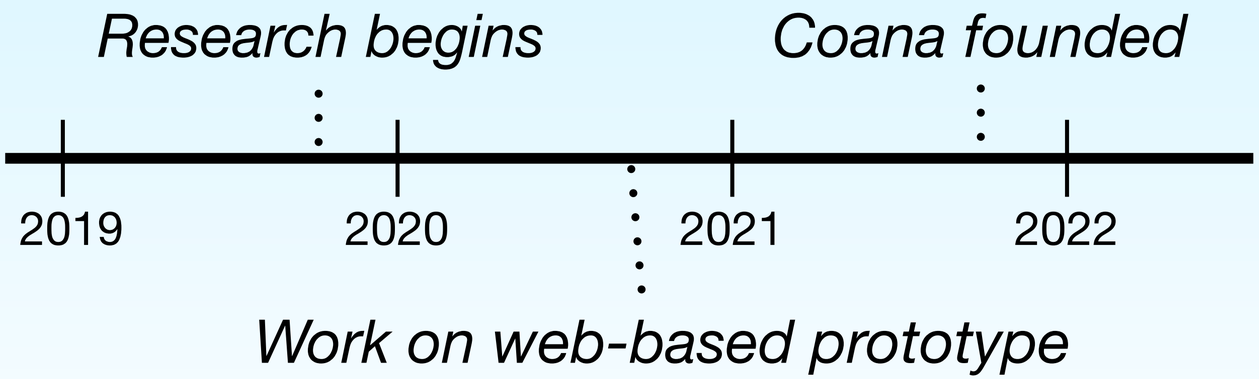
European Research Council
Established by the European Commission

Innovationsfonden



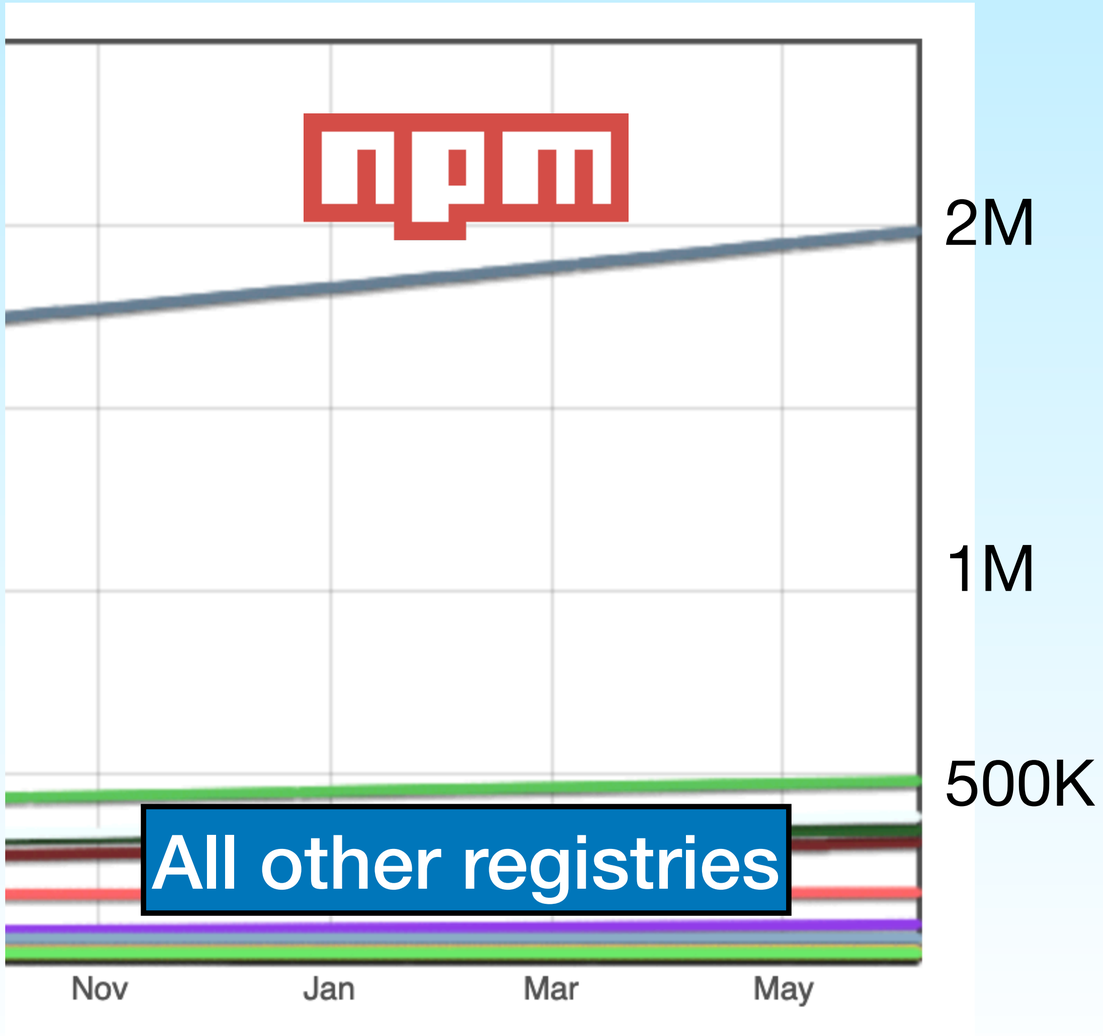






Node package manager (npm) registry

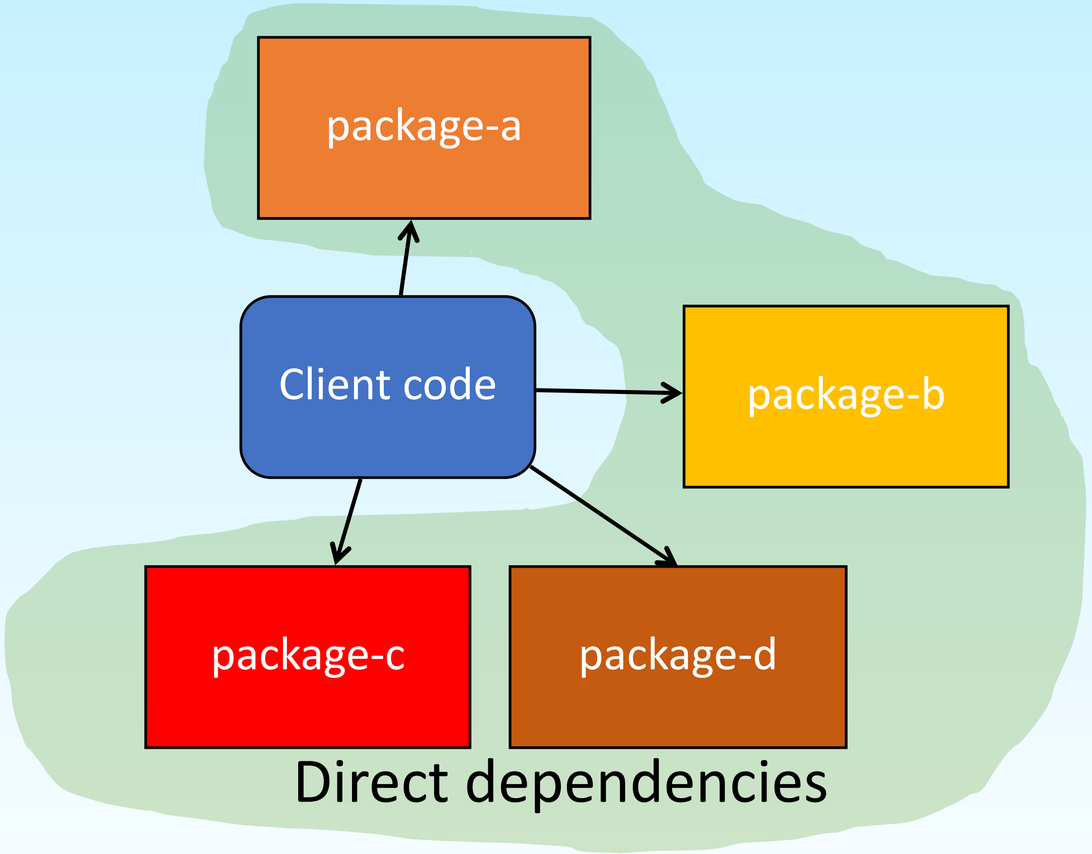
- Package management system for JavaScript/TypeScript
- Around 2M packages
- 1B package installations per day
- Used for Node.js, Angular, React, Vue.js, etc.




Client code

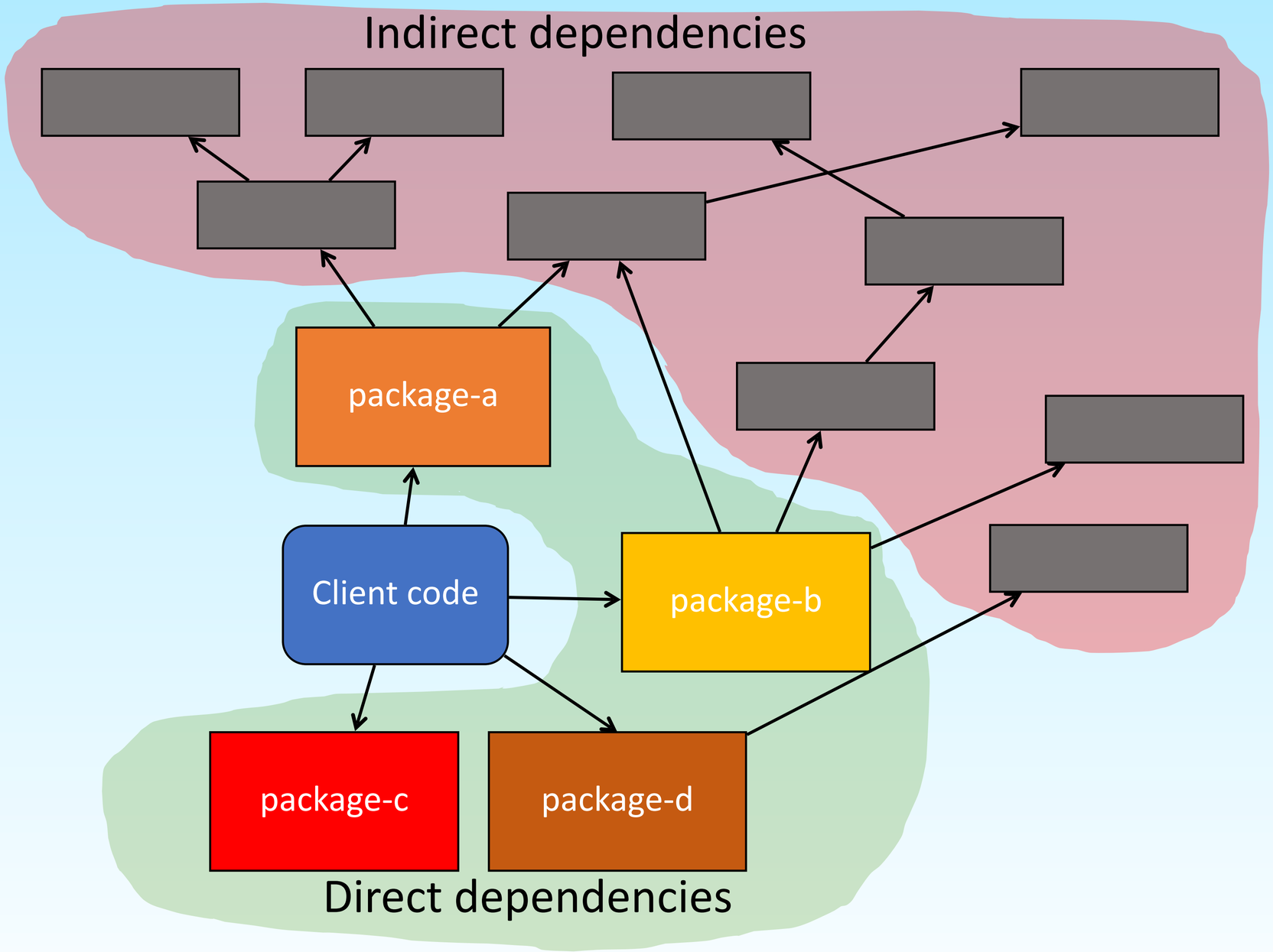
package.json

```
"dependencies" : {
   "package-a":^1.0.0,
   "package-b":^4.0.0,
   "package-c":^2.3.0,
   "package-d":^1.2.0,
 }
```



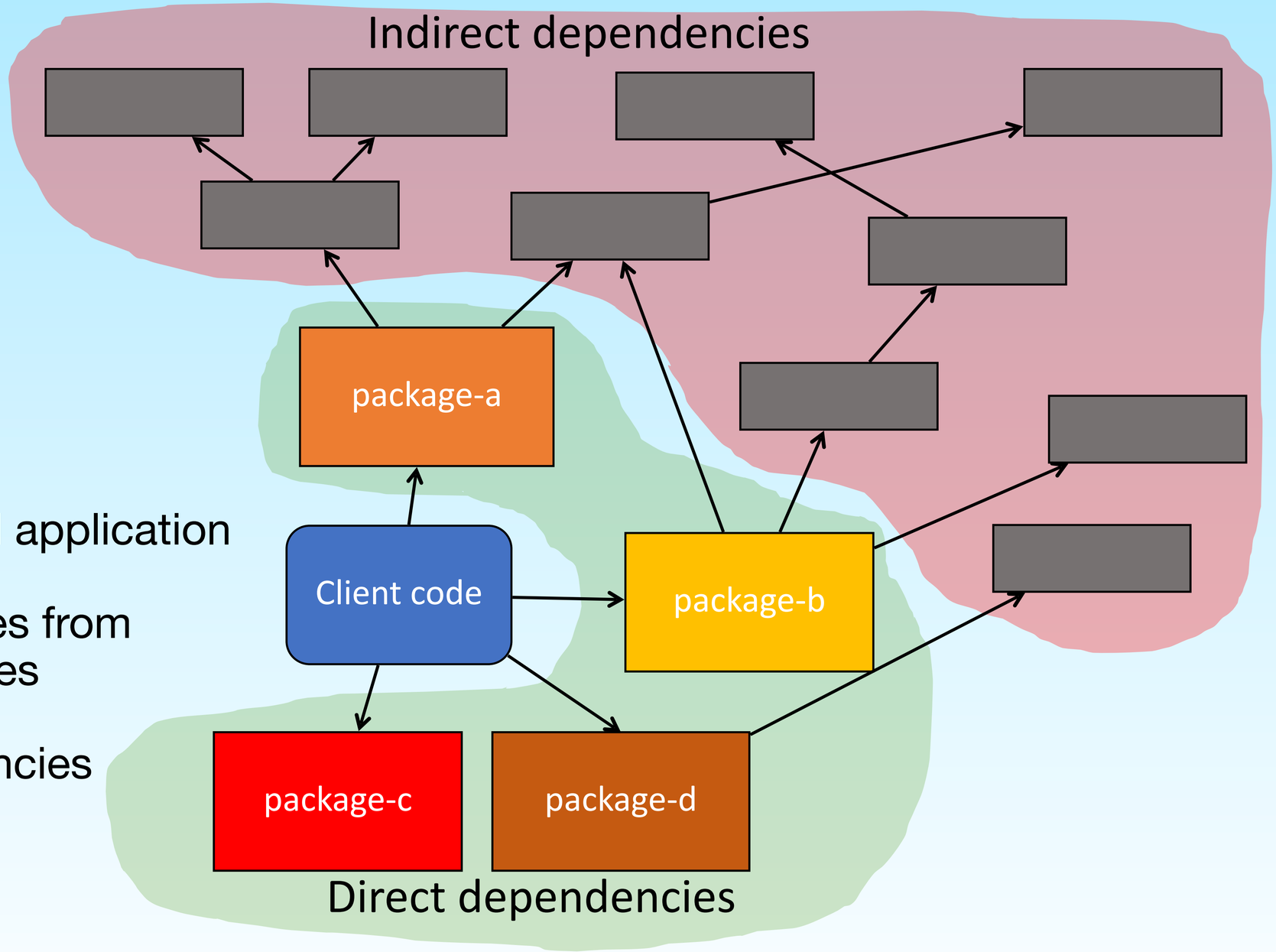
package.json

```
"dependencies" : {
   "package-a":^1.0.0,
   "package-b":^4.0.0,
   "package-c":^2.3.0,
   "package-d":^1.2.0,
 }
```



package.json

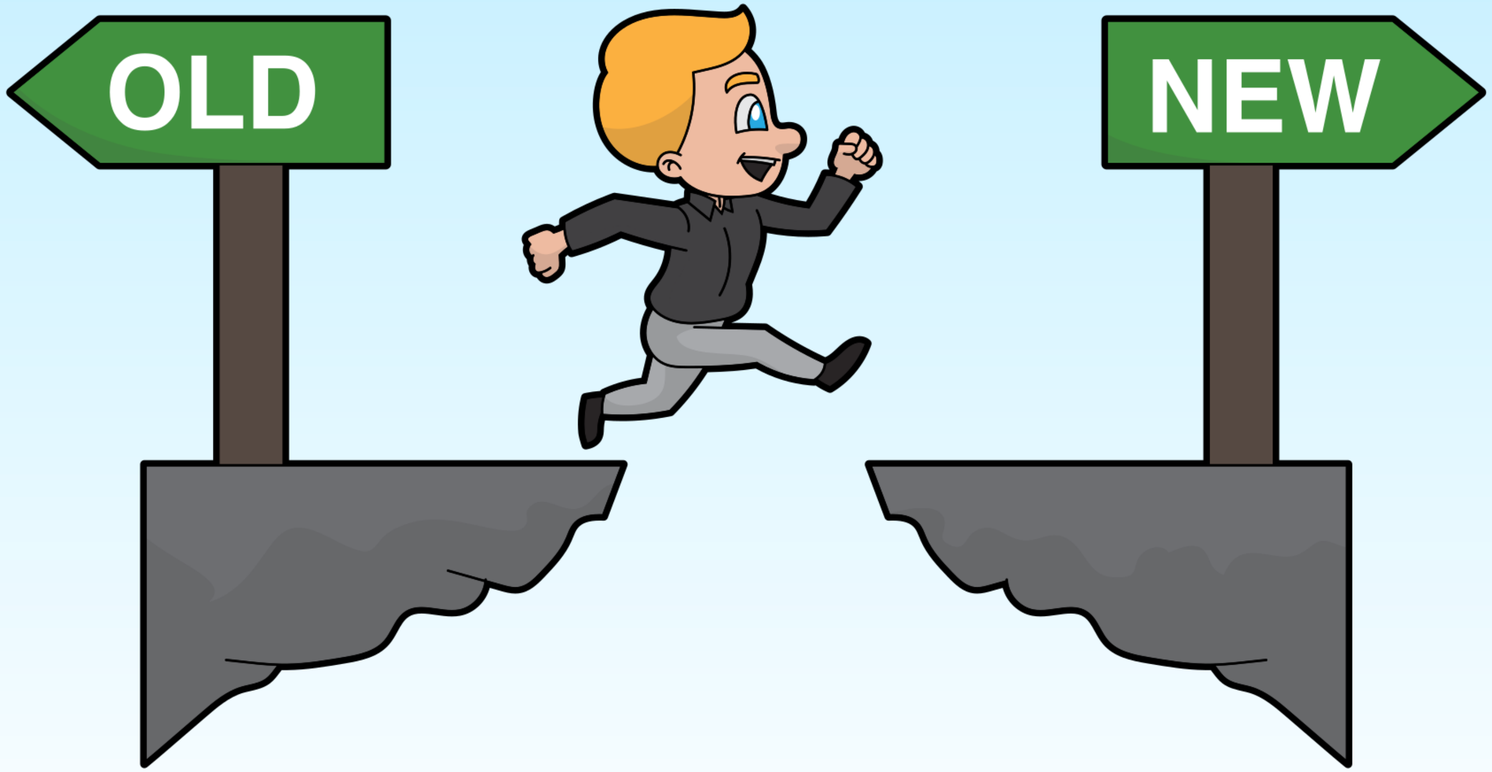
```
"dependencies" : {
   "package-a":^1.0.0,
   "package-b":^4.0.0,
   "package-c":^2.3.0,
   "package-d":^1.2.0,
 }
```



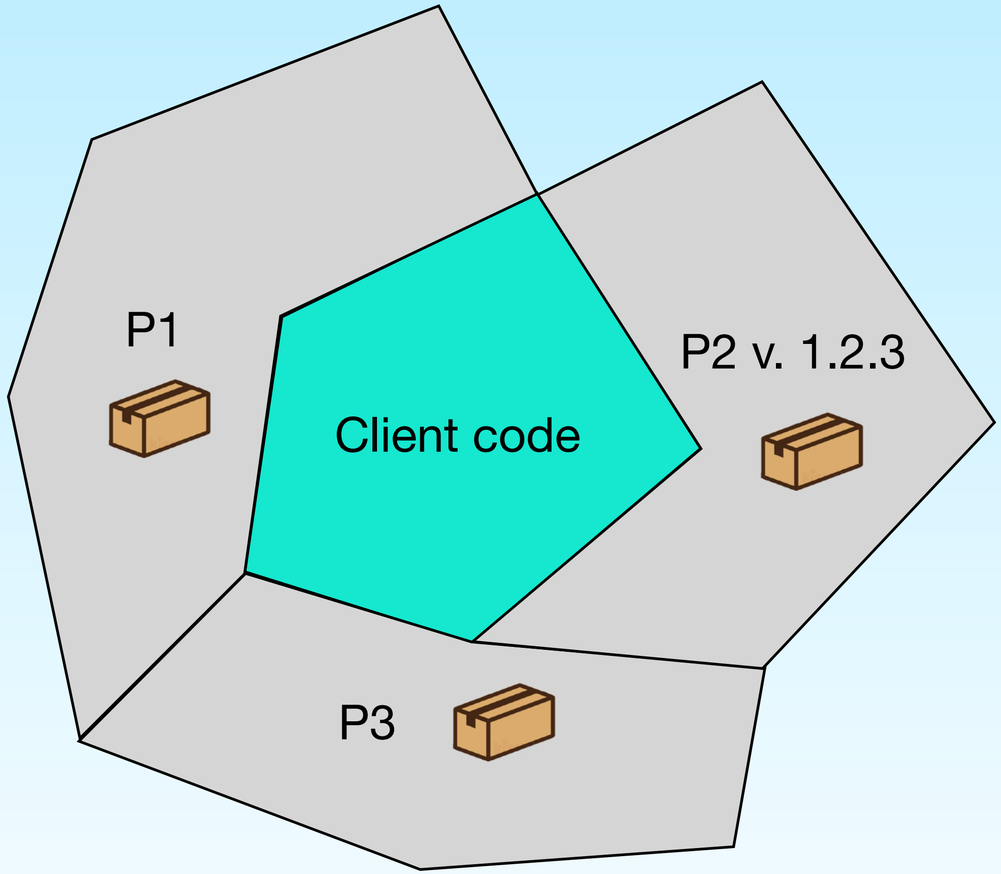
Average npm-based application

- 90% of code comes from third-party packages
- 45 direct dependencies

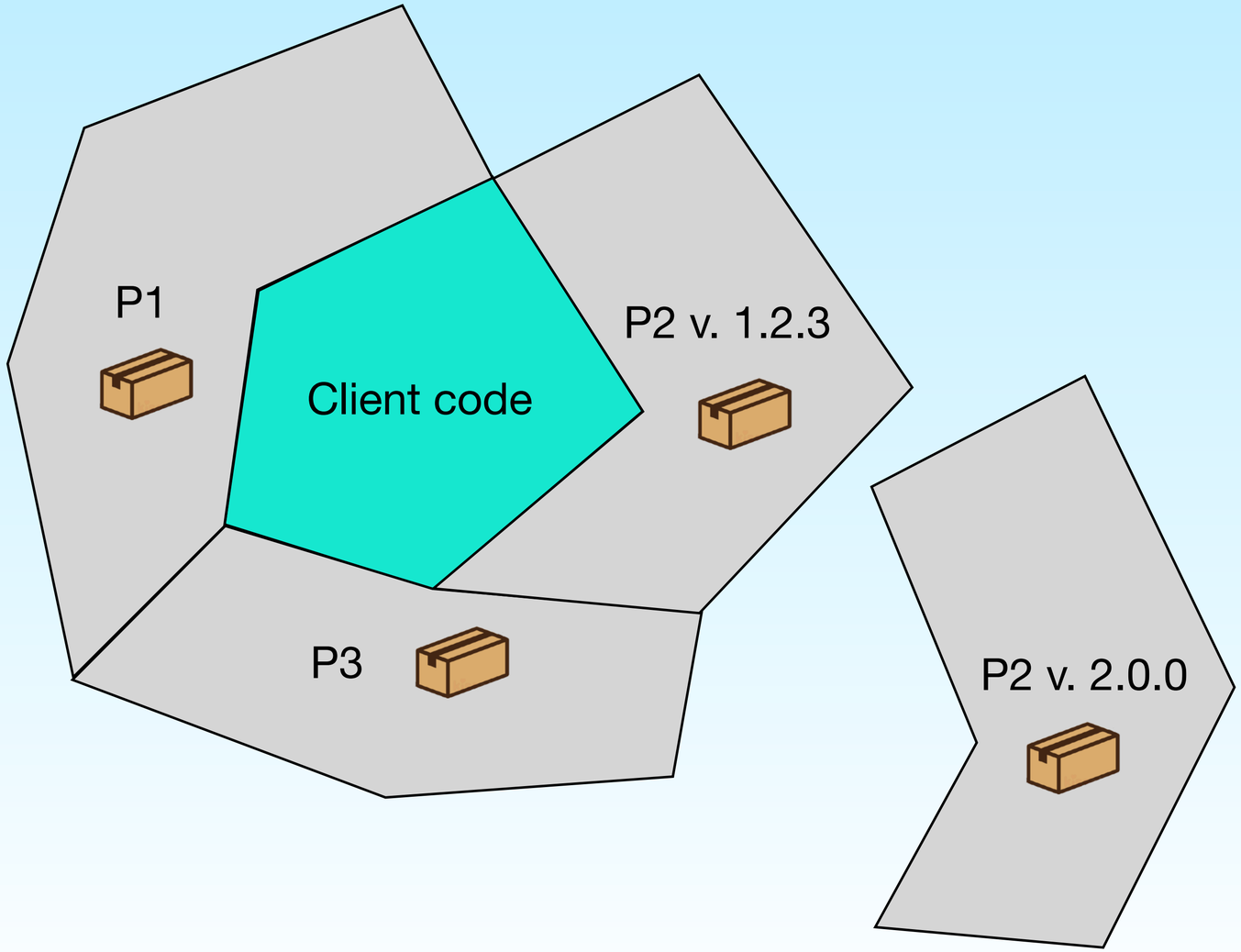
Package upgrades



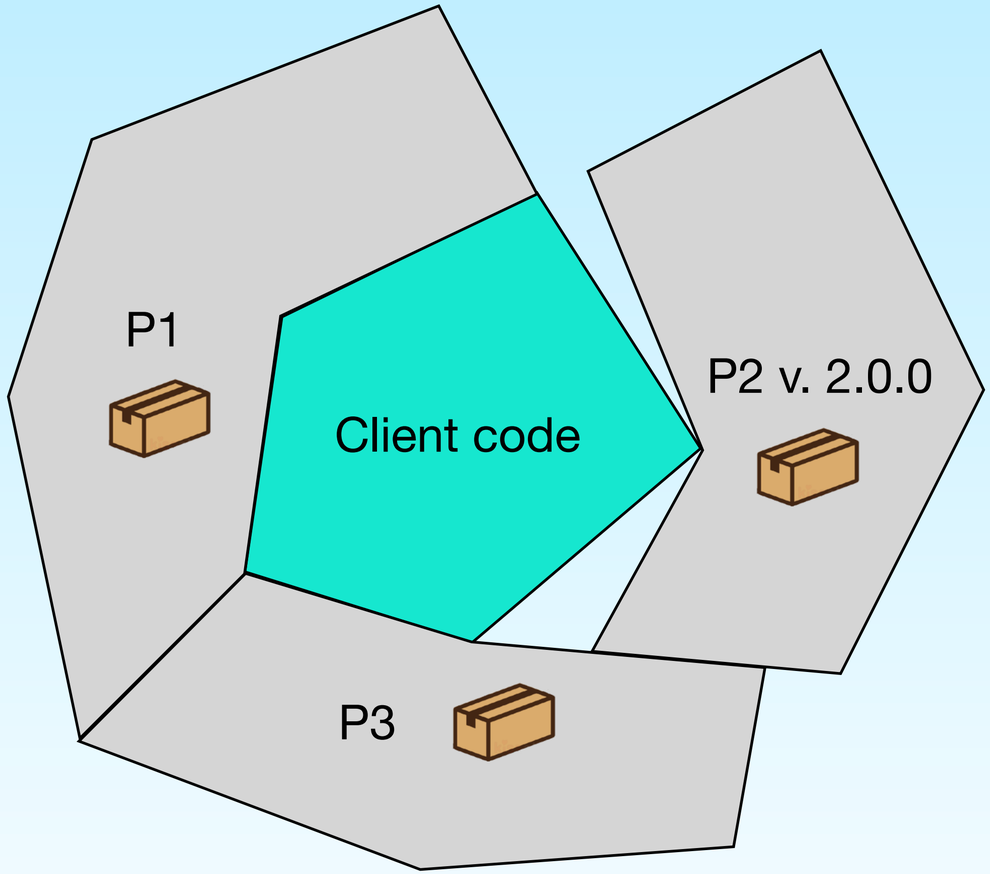
The problem



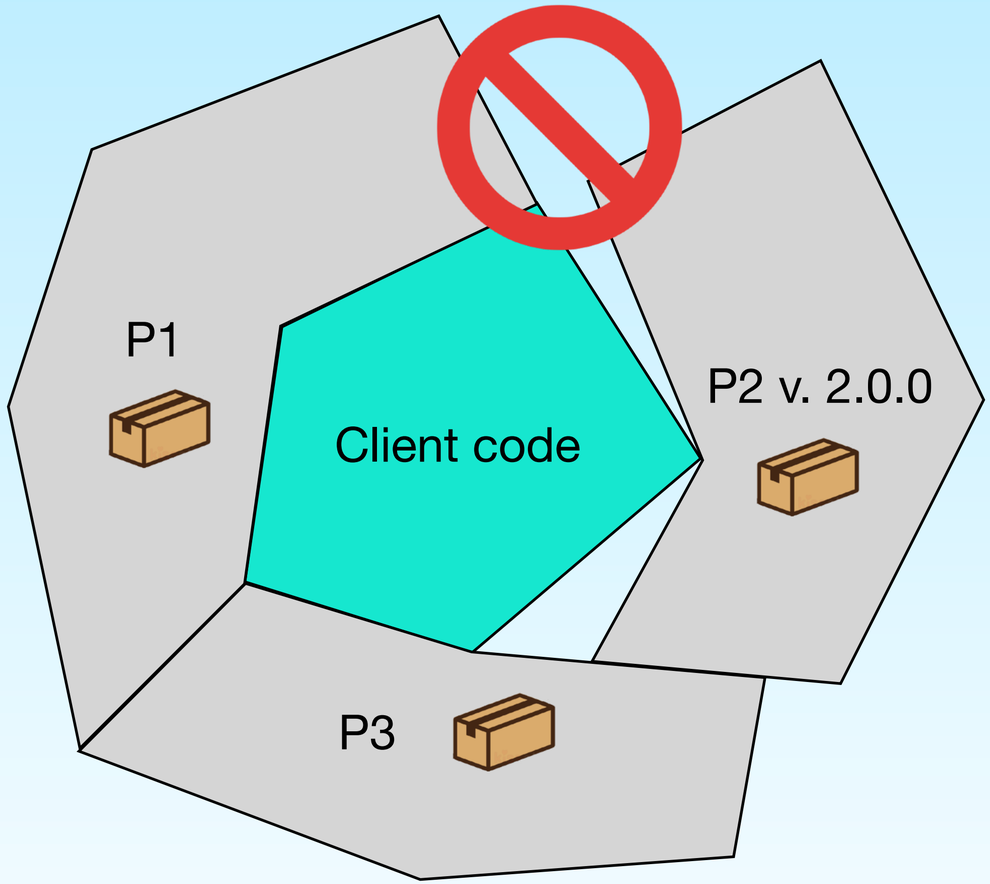
The problem



The problem

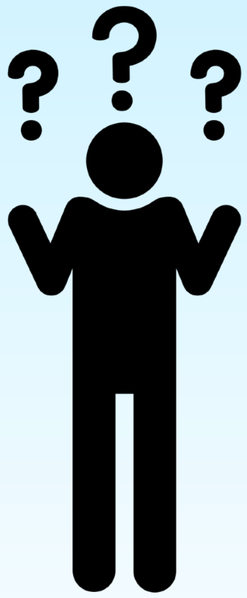


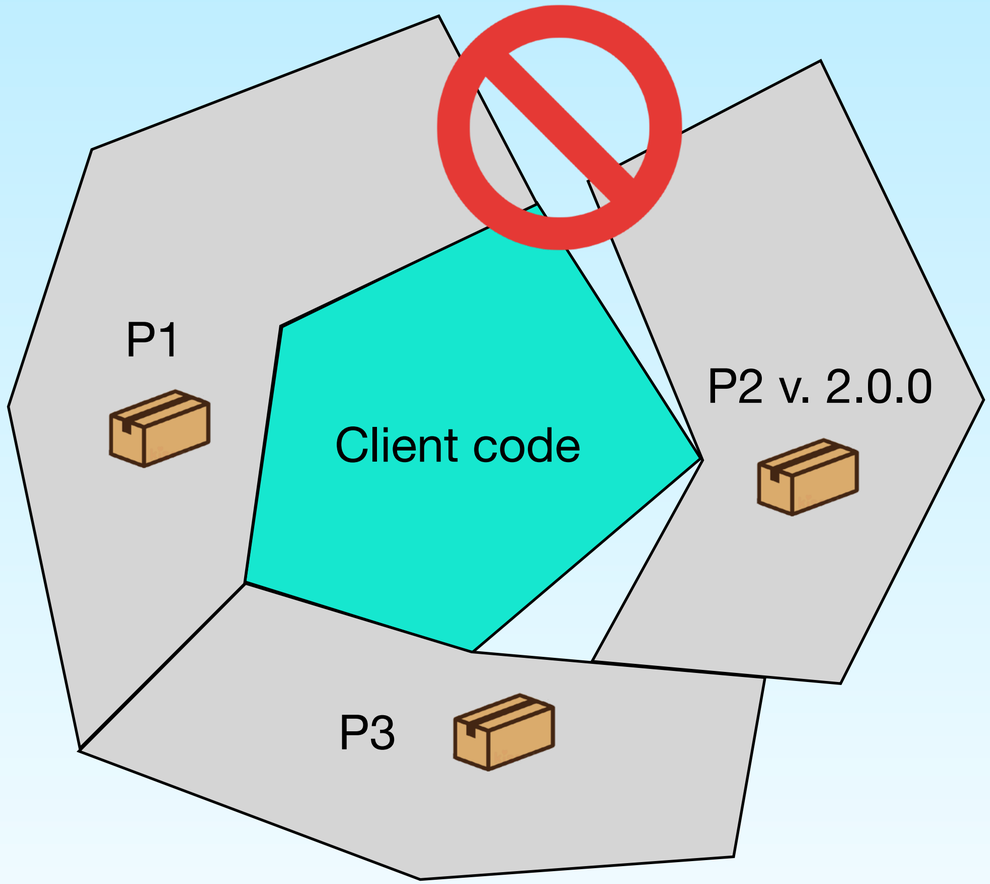
The problem



The problem

How should the developer adapt the client code?





Why keep dependencies up-to-date?



- Reduce vulnerability space
(according to OWASP 6th biggest security threat for web)
- Benefit from new features and bug fixes
- Adhere to peer dependency constraints

Semantic versioning

MAJOR.MINOR.PATCH

Backwards incompatible changes

Backwards compatible changes

Semantic versioning

MAJOR.MINOR.PATCH

1.2.3 ➜ 1.2.7

Backwards incom **le changes**

Semantic versioning

MAJOR.MINOR.PATCH

1.2.3 ➜ 1.3.0

Backwards incom **le changes**

Semantic versioning

MAJOR.MINOR.PATCH

1.2.3 ➜ 2.0.0

Backwards incom **le changes**

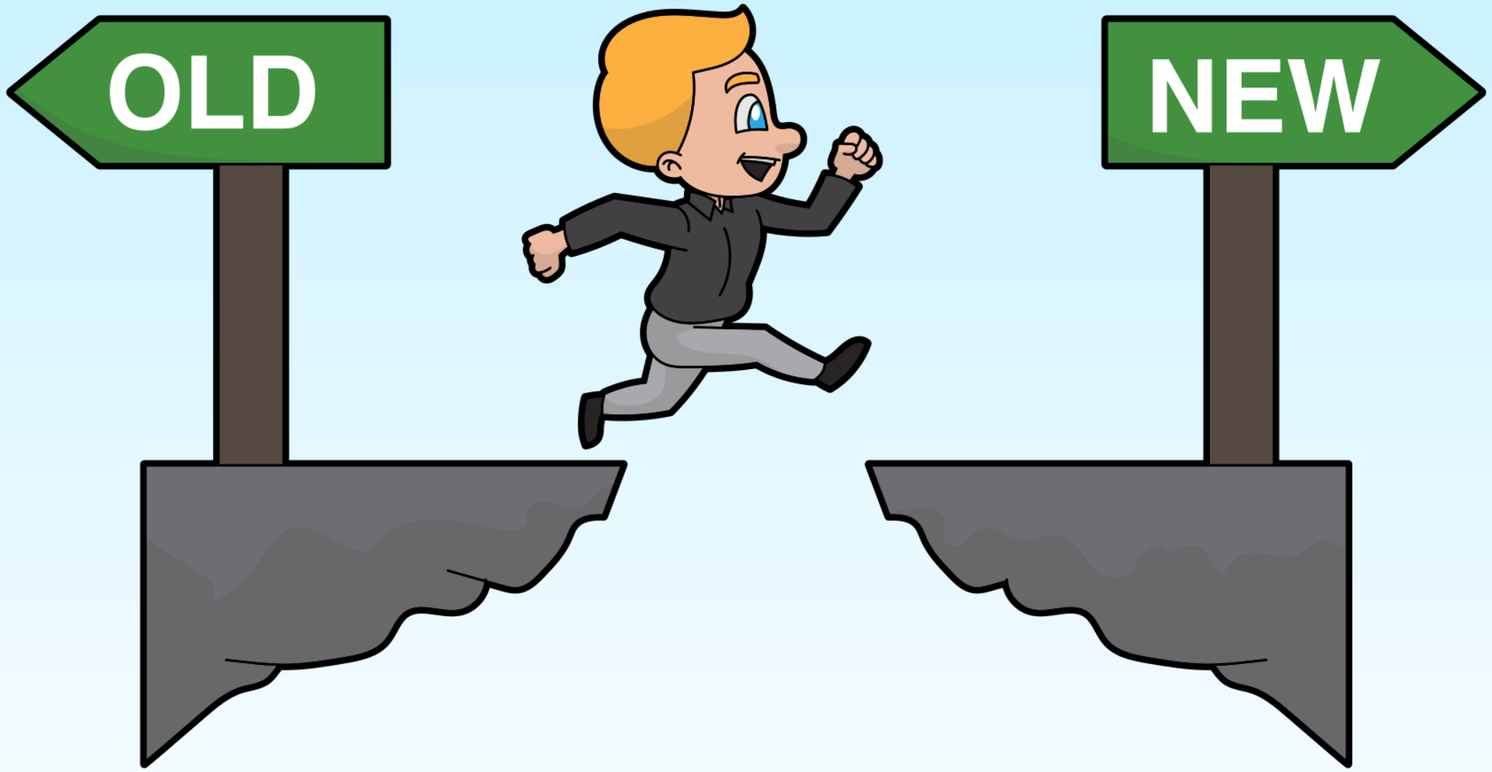
Semantic versioning

MAJOR.MINOR.PATCH

Backwards incompatible changes

Backwards compatible changes

Current upgrade practice



Current practice: entirely manual process



Requires reading complex and incomplete documentation



Time-consuming

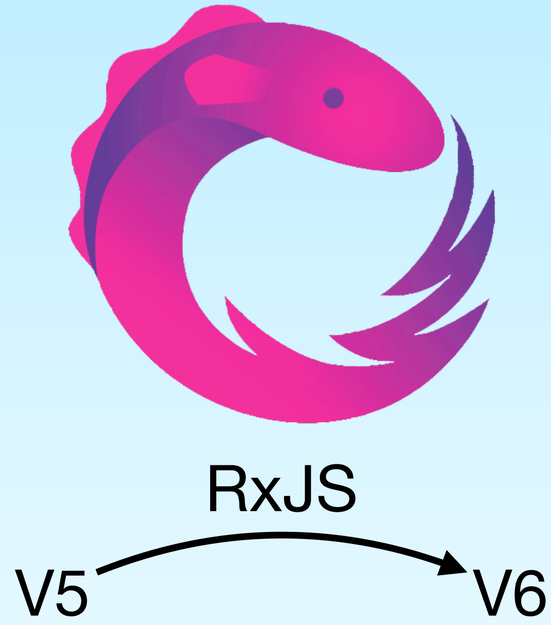


Error-prone

Example package upgrade

https://github.com/jeffbski/redux-logic

- 20,000 line application
- 137 source code changes
- 2 days of work
- First 2 commits had errors



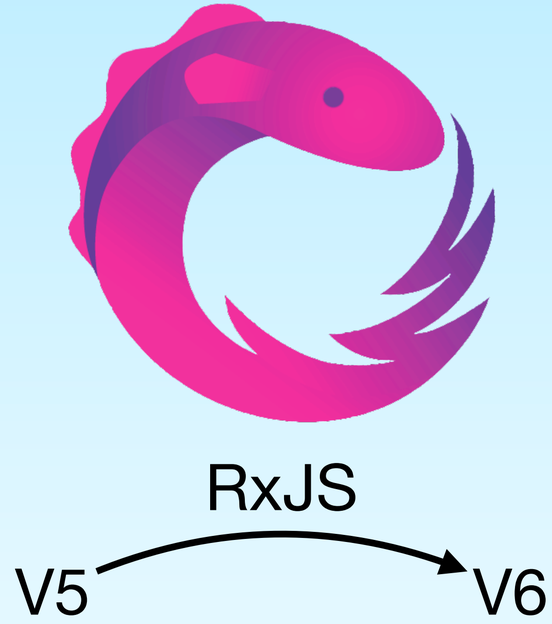
Example package upgrade

https://github

- 20,000 line
- 137 source
- 2 days of v
- First 2 com

BREAKING CHANGES

- **ajax:** will no longer execute a CORS request by default, you must opt-in with the `crossDomain` flag in the config.
- **mergeMap|concatMap|concatMapTo:** mergeMap, concatMap and concatMapTo no longer support a result selector, if you need to use a result selector, use the following pattern: `source.mergeMap(x => of(x + x).pipe(map(y => y + x))` (the pattern would be the same for `concatMap`).
- **never:** no longer exported. Use the `NEVER` constant instead.
- **bindCallback:** removes result selector, use `map` instead: `bindCallback(fn1, fn2)()` becomes `bindCallback(fn1)().pipe(map(fn2))`
- **Rx.ts:** importing from `rxjs/Rx` is no longer available. Upcoming backwards compat solution will allow that
- **Symbol.iterator:** We are no longer polyfilling `Symbol.iterator`. That would be done by a proper polyfilling library
- **Observable.if:** TypeScript users using `Observable.if` will have to cast `Observable` as any to get to `if`. It is a better idae to just use `iif` directly via `import { iif } from 'rxjs';`
- **bindNodeCallback:** resultSelector removed, use `map` instead: `bindNodeCallback(fn1, fn2)()` becomes `bindNodeCallback(fn1)().pipe(map(fn2))`
- **Symbol.observable:** RxJS will no longer be polyfilling Symbol.observable. That should be done by an actual polyfill library. This is to prevent duplication of code, and also to prevent having modules with side-effects in rxjs.
- **fromEvent:** result selector removed, use `map` instead: `fromEvent(target, 'click', fn)` becomes `fromEvent(target, 'click').pipe(map(fn))`
- **last:** no longer accepts `resultSelector` argument. To get this same functionality, use `map`.
- **first:** no longer supports `resultSelector` argument. The same functionality can be achieved by simply mapping either before or after `first` depending on your use case.
- **exhaustMap:** `resultSelector` no longer supported, to get this functionality use: `source.pipe(exhaustMap(x => of(x + x).pipe(map(y => x + y))))`
- **switchMap|switchMapTo:** `switchMap` and `switchMapTo` no longer take `resultSelector` arguments, to get the same functionality use `switchMap` and `map` in combination: `source.pipe(switchMap(x => of(x + x).pipe(y => x + y)))`.
- **mergeMapTo:** `mergeMapTo` no longer accepts a resultSelector, to get this functionality, you'll want to use `mergeMap` and `map` together: `source.pipe(mergeMap(() => inner).pipe(map(y => x + y)))`
- **fromEventPattern:** no longer supports a result selector, use `map` instead: `fromEventPattern(fn1, fn2, fn3)` becomes `fromEventPattern(fn1, fn2).pipe(map(fn3))`



Example package upgrade

https://github

- 20,000 line
- 137 source
- 2 day
- First 2

BREAKING CHANGES

- **ajax:** will no longer execute a CORS request by default, you must opt-in with the `crossDomain` flag in the config.
- **mergeMap|concatMap|concatMapTo:** mergeMap, concatMap and concatMapTo no longer support a result selector, if you need to use a result selector, use the following pattern: `source.mergeMap(x => of(x + x).pipe(map(y => y + x))` (the pattern would be the same for `concatMap`).
- **never:** no longer exported. Use the `NEVER` constant instead.
- **bindCallback:** removes result selector, use `map` instead: `bindCallback(fn1, fn2)()` becomes `bindCallback(fn1)().pipe(map(fn2))`
- **Rx.ts:** importing from `rxjs/Rx` is no longer available. Upcoming backwards compat solution will allow that
- **Symbol.iterator:** We are no longer polyfilling `Symbol.iterator`. That would be done by a proper polyfilling libr
- **Observable.if:** TypeScript users using `Observable.if` will have to cast `Observ`
  use `iif` directly via `import { iif } from 'rxjs'`
- **bindNodeCallback:** re

same functionality can be achieved by simply mapping either before or
use case.

- **exhaustMap:** `resultSelector` no longer supported, to get this functionality use: `source.pipe(exhaustMap(x => of(x + x).pipe(map(y => x + y))))`
- **switchMap|switchMapTo:** `switchMap` and `switchMapTo` no longer take `resultSelector` arguments, to get the same functionality use `switchMap` and `map` in combination: `source.pipe(switchMap(x => of(x + x).pipe(y => x + y)))`.
- **mergeMapTo:** `mergeMapTo` no longer accepts a resultSelector, to get this functionality, you'll want to use `mergeMap` and `map` together: `source.pipe(mergeMap(() => inner).pipe(map(y => x + y)))`
- **fromEventPattern:** no longer supports a result selector, use `map` instead: `fromEventPattern(fn1, fn2, fn3)` becomes `fromEventPattern(fn1, fn2).pipe(map(fn3))`

JSFIX demo of this upgrade later



RxJS

V5 → V6

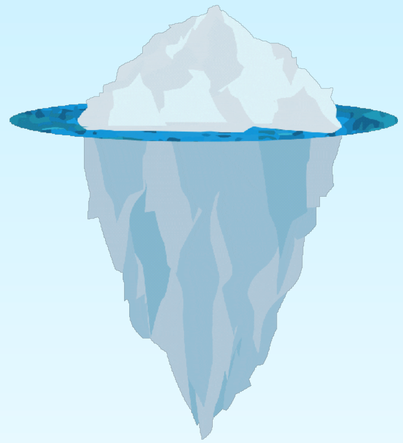
Which packages do developers use? How do they maintain dependencies?

- 15,000 GitHub projects using npm
- 297,000 direct dependencies



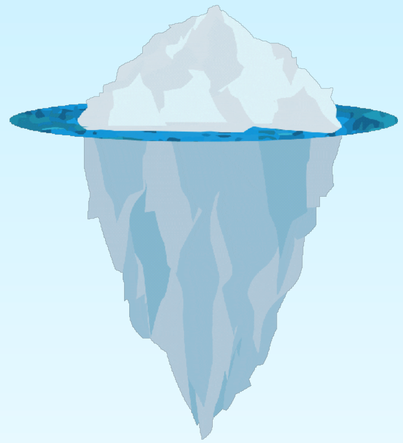
Experiment findings

- 50% of all dependencies are from top 500
- 35% of all dependencies are at least one major version outdated
- > 50% of all projects have a major-outdated dependency



Experiment findings

- 50% of all dependencies are from top 500
- 35% of all dependencies are at least one major version outdated
- > 50% of all projects have a major-outdated dependency



Conclusions

1. Few packages are used a lot (most packages are used a little or not at all)
2. No clear practice for keeping dependencies upgraded

Upgrade assistance tools

General tools

- Works for all updates
- Do not assist with code changes



snyk



Dependabot

Package-specific tools

- May include code mods
- Sometimes make assumptions about coding style
- Are time-consuming to create
- Typically only for heavily used packages

jscodeshift

https://update.angular.io/

JSFIX: automating package upgrades



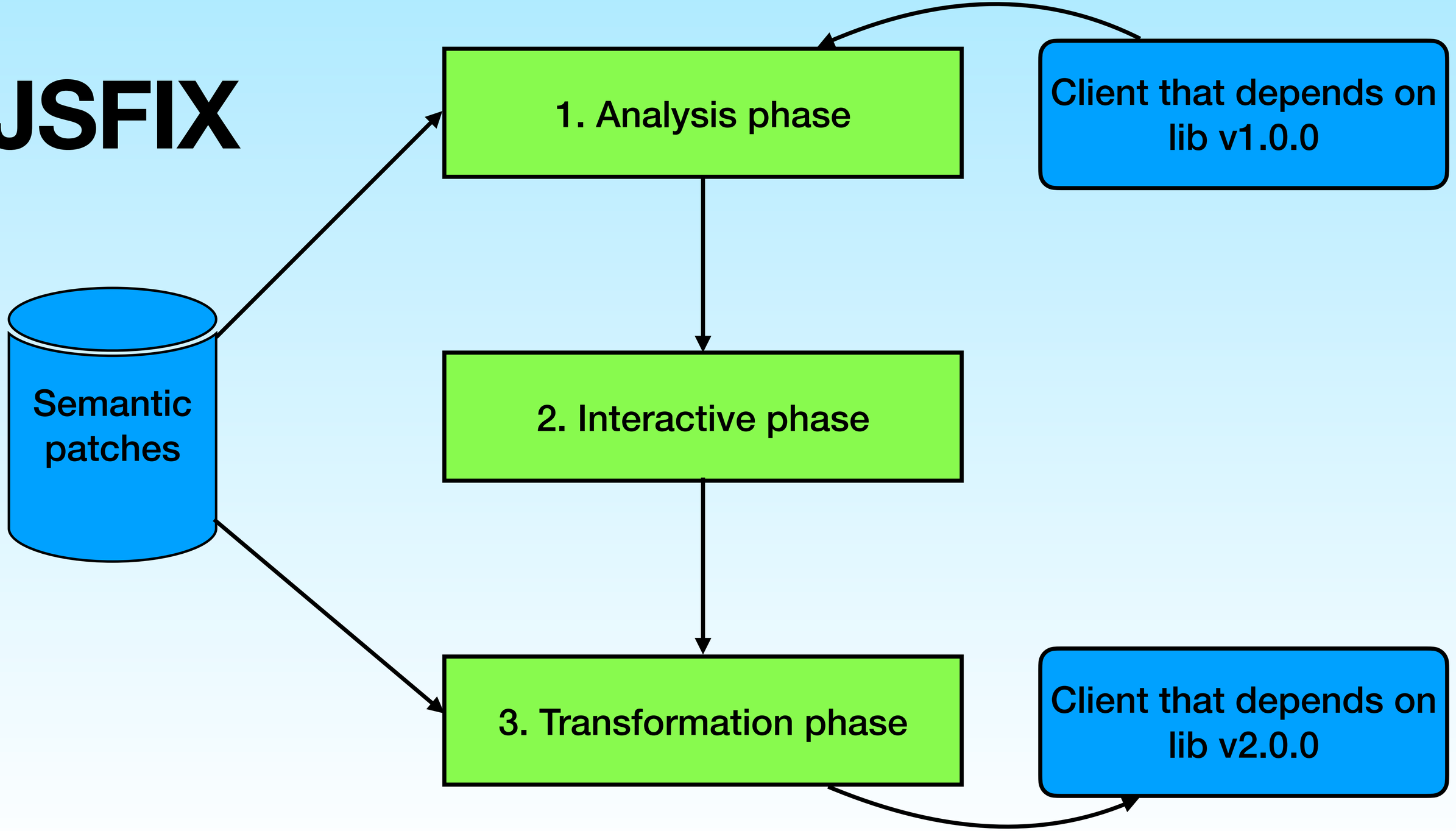
Demo - https://jsfix.live

Accepted pull request: https://github.com/mWater/minimongo/pull/81

How JSFIX works



JSFIX


1. Analysis phase
Client that depends on lib v1.0.0
Semantic patches
2. Interactive phase
3. Transformation phase
Client that depends on lib v2.0.0

Breaking change: *Chaining operators have been replaced with pipe operators*

```
import { fromEvent } from 'rxjs';

const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);
```



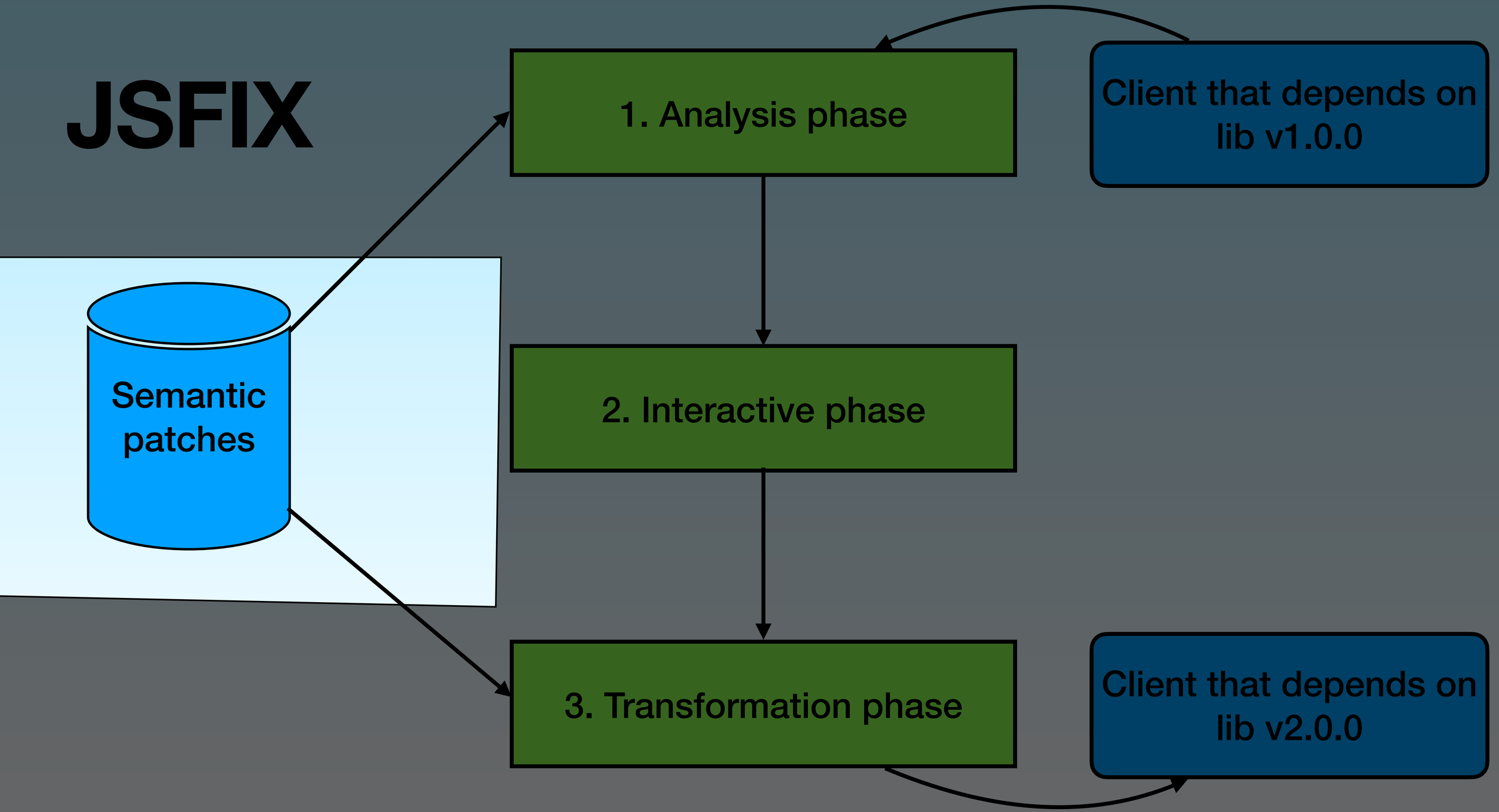
RxJS

V5 → V6

```
import { fromEvent } from ‘rxjs';
import { map, filter } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
clicks.pipe(map(ev => ev.clientX), filter(x => x > 300));
```

JSFIX

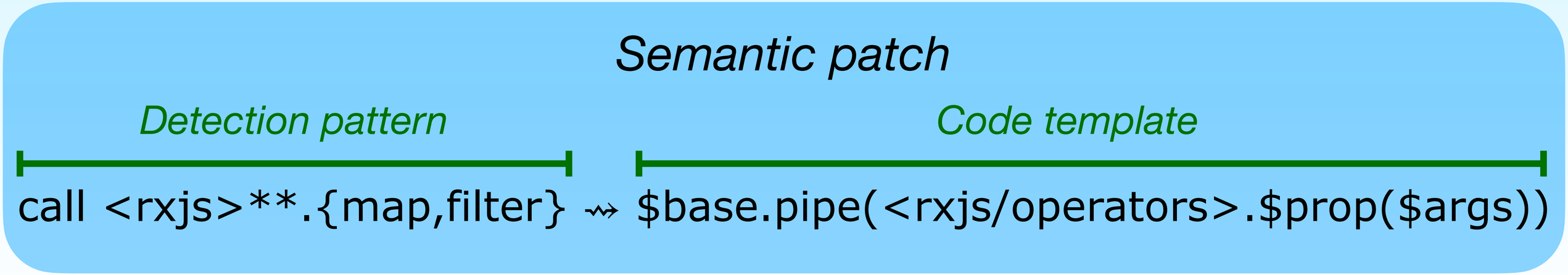

1. Analysis phase
Client that depends on lib v1.0.0
Semantic patches
2. Interactive phase
3. Transformation phase
Client that depends on lib v2.0.0

Semantic patches

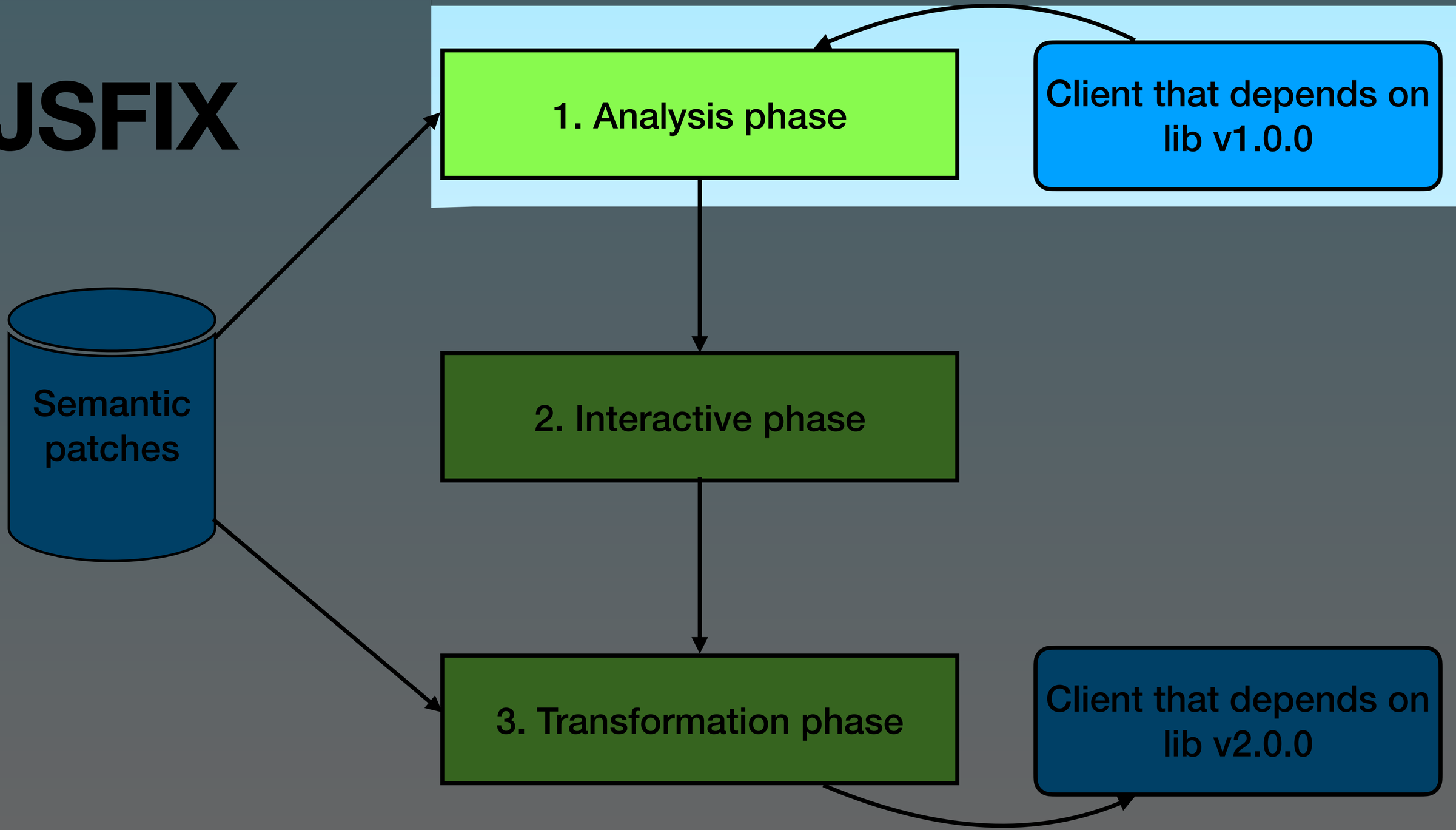
- Formalization of changelogs
- Written by us (later by anyone)
- Consists of detection pattern and code template

Detection pattern: identifies API usages affected by breaking changes

Code template: adapts API usages to the breaking changes



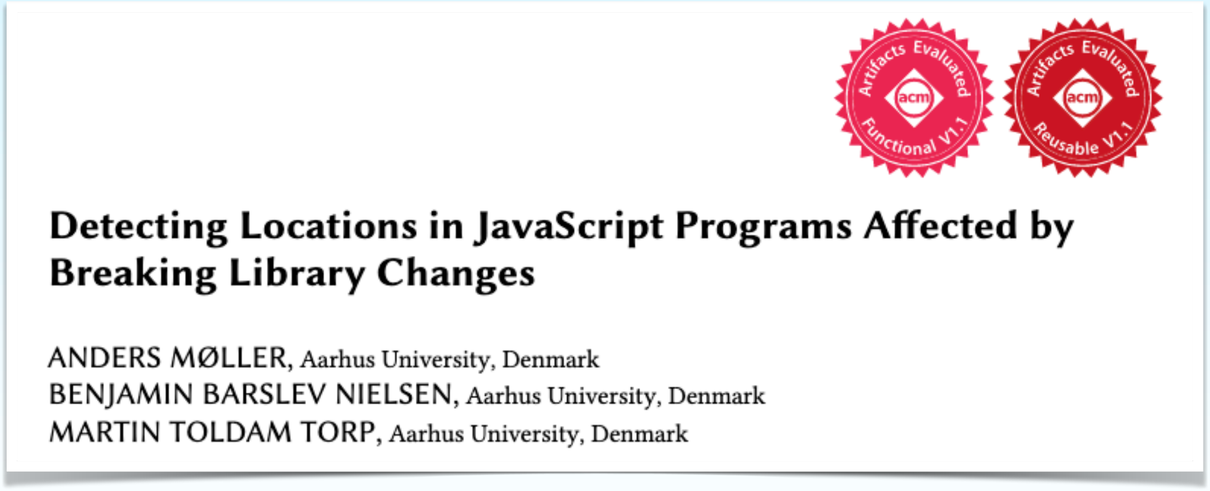
JSFIX


1. Analysis phase
Client that depends on lib v1.0.0
Semantic patches
2. Interactive phase
3. Transformation phase
Client that depends on lib v2.0.0

First phase: analysis

Computes source locations affected by breaking changes

- Static AST-based analysis (flow-insensitive, field-based)
- Extremely fast
- Aims for practical soundness
- Marks *affected locations* with *high*/*low* confidence



call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))

```
import { fromEvent } from 'rxjs';

const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);
```

call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))

```
import { fromEvent } from 'rxjs';

const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);
```

Something derived from the rxjs package

call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))

```
import { fromEvent } from 'rxjs';

const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);
```

Something derived from the rxjs package

```
call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))
```

```
import { fromEvent } from 'rxjs';

const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);
```

Matches since clicks is from rxjs

Something derived from the rxjs package

call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))

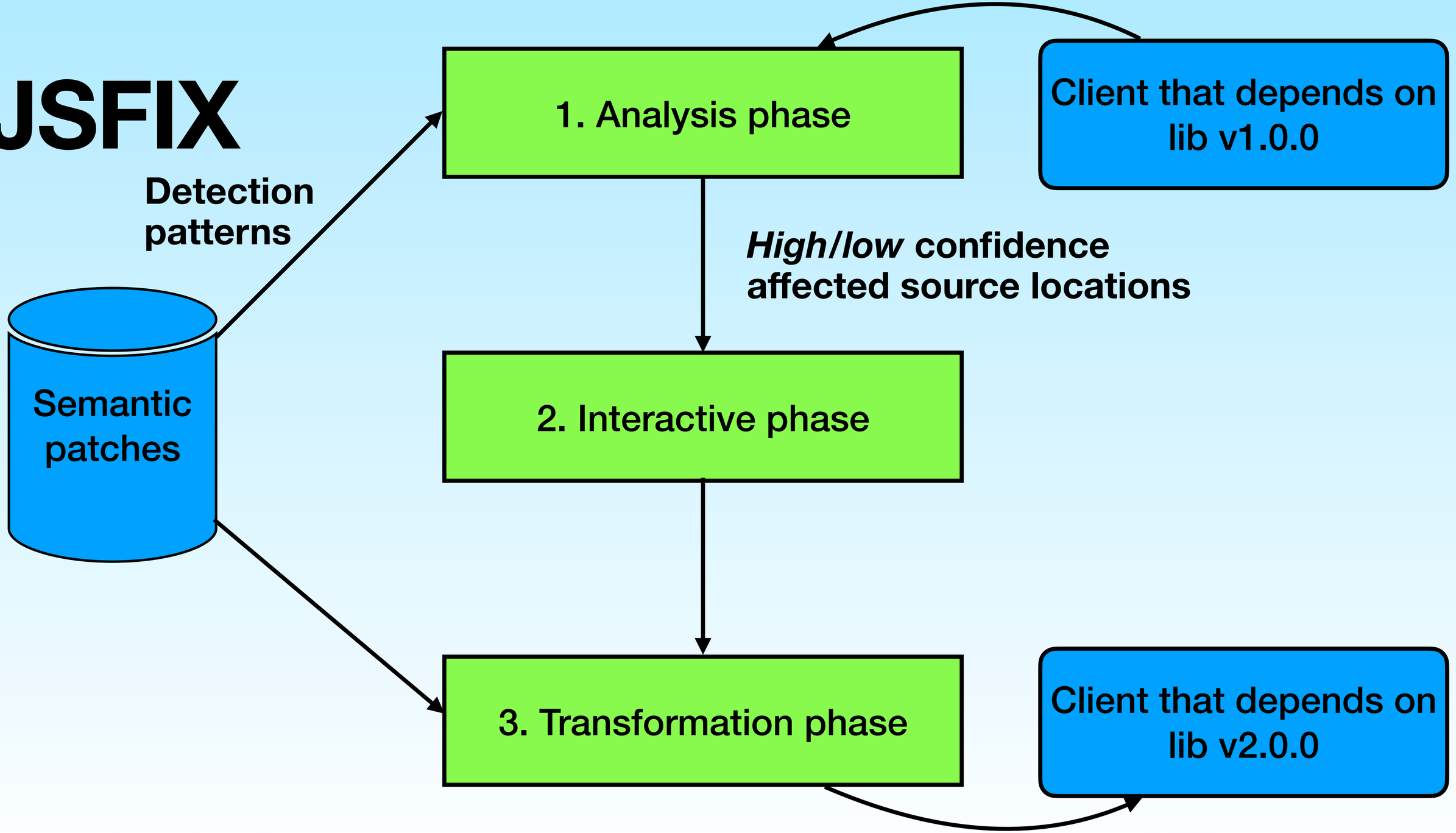
```
import { fromEvent } from 'rxjs';

const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);
```

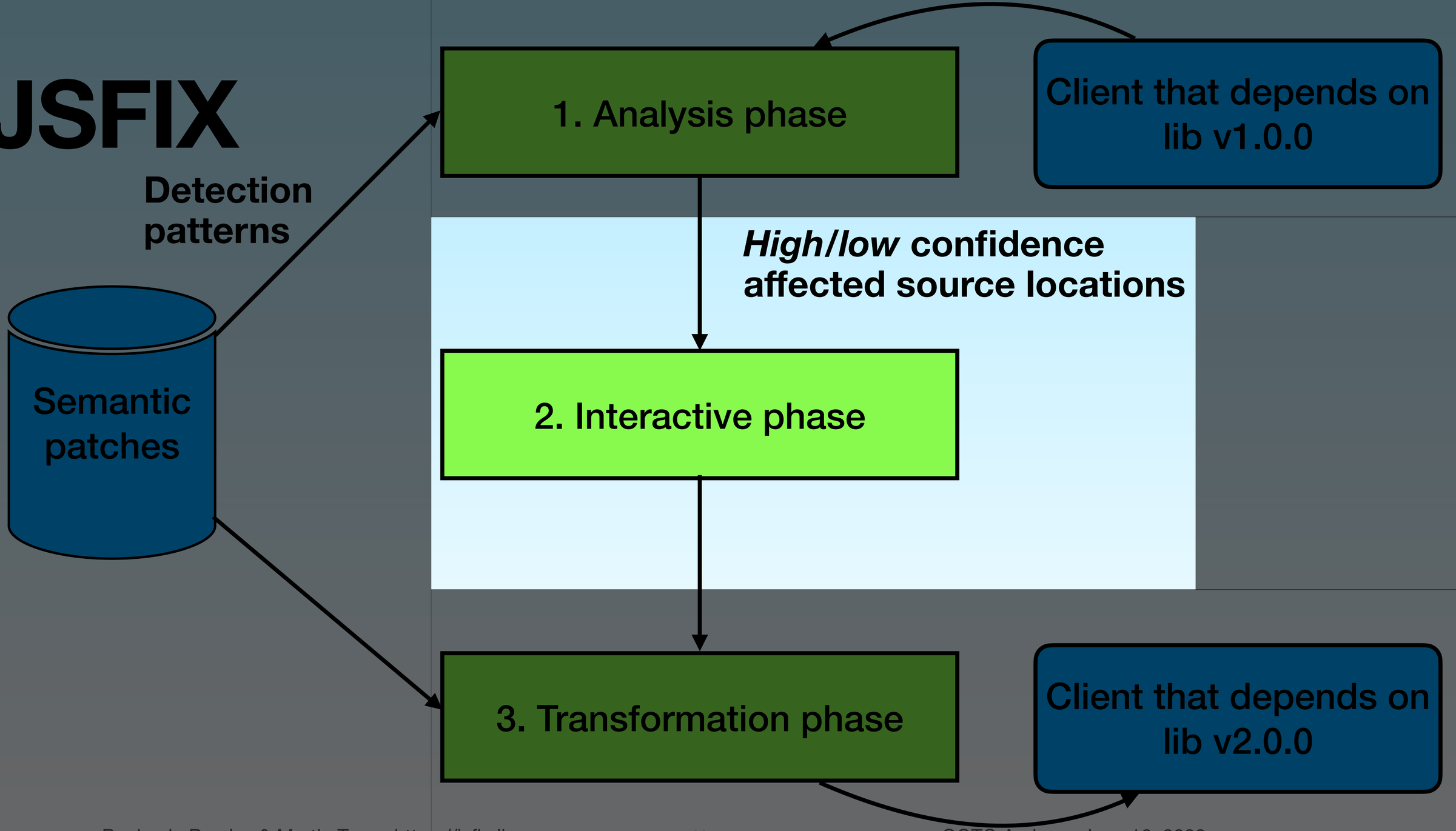
Matches since clicks is from rxjs

Matches since clicks.map(…) by extension is from rxjs

JSFIX

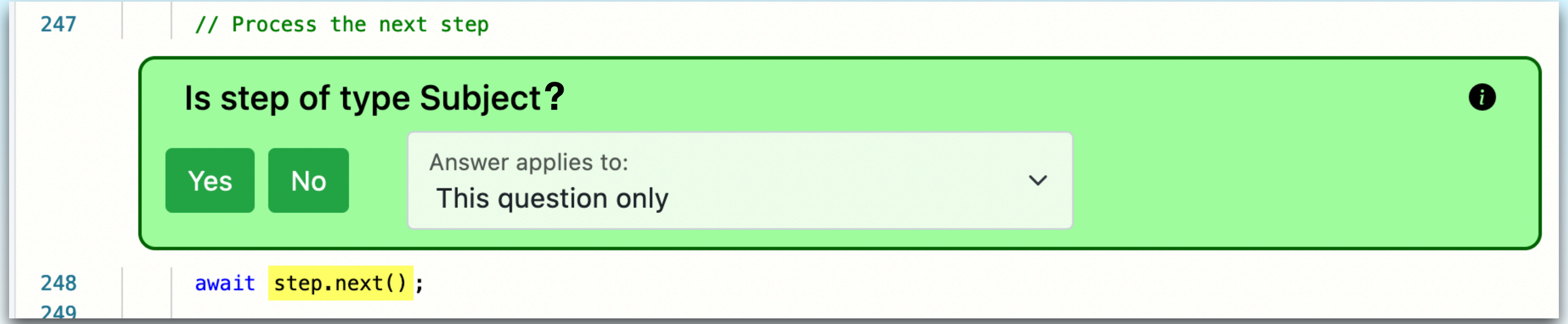

Detection patterns
1. Analysis phase
Client that depends on lib v1.0.0
High/low confidence affected source locations
Semantic patches
2. Interactive phase
3. Transformation phase
Client that depends on lib v2.0.0

JSFIX

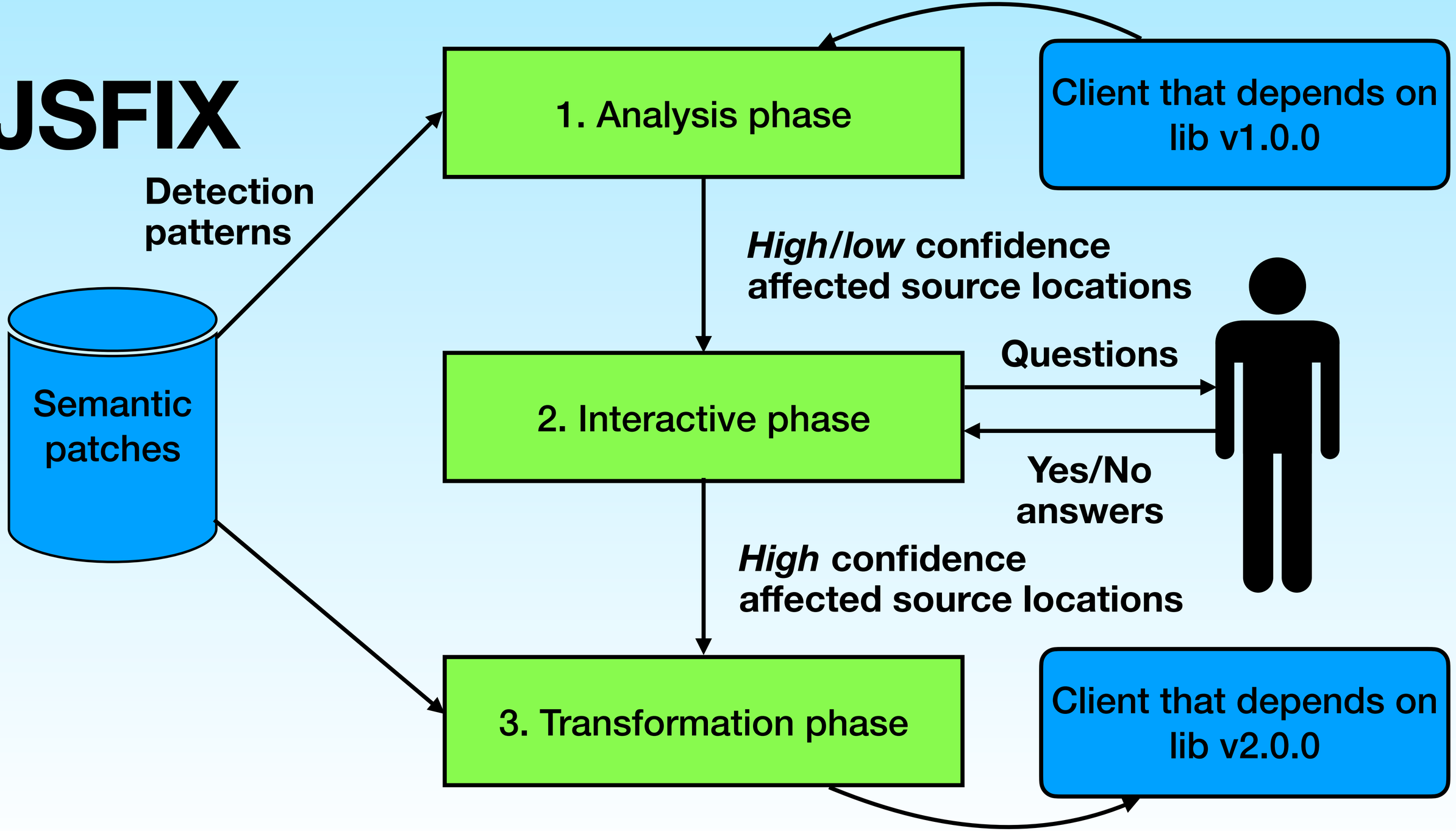

Detection patterns
Semantic patches
1. Analysis phase
Client that depends on lib v1.0.0
High/low confidence affected source locations
2. Interactive phase
3. Transformation phase
Client that depends on lib v2.0.0

Second phase: interactive

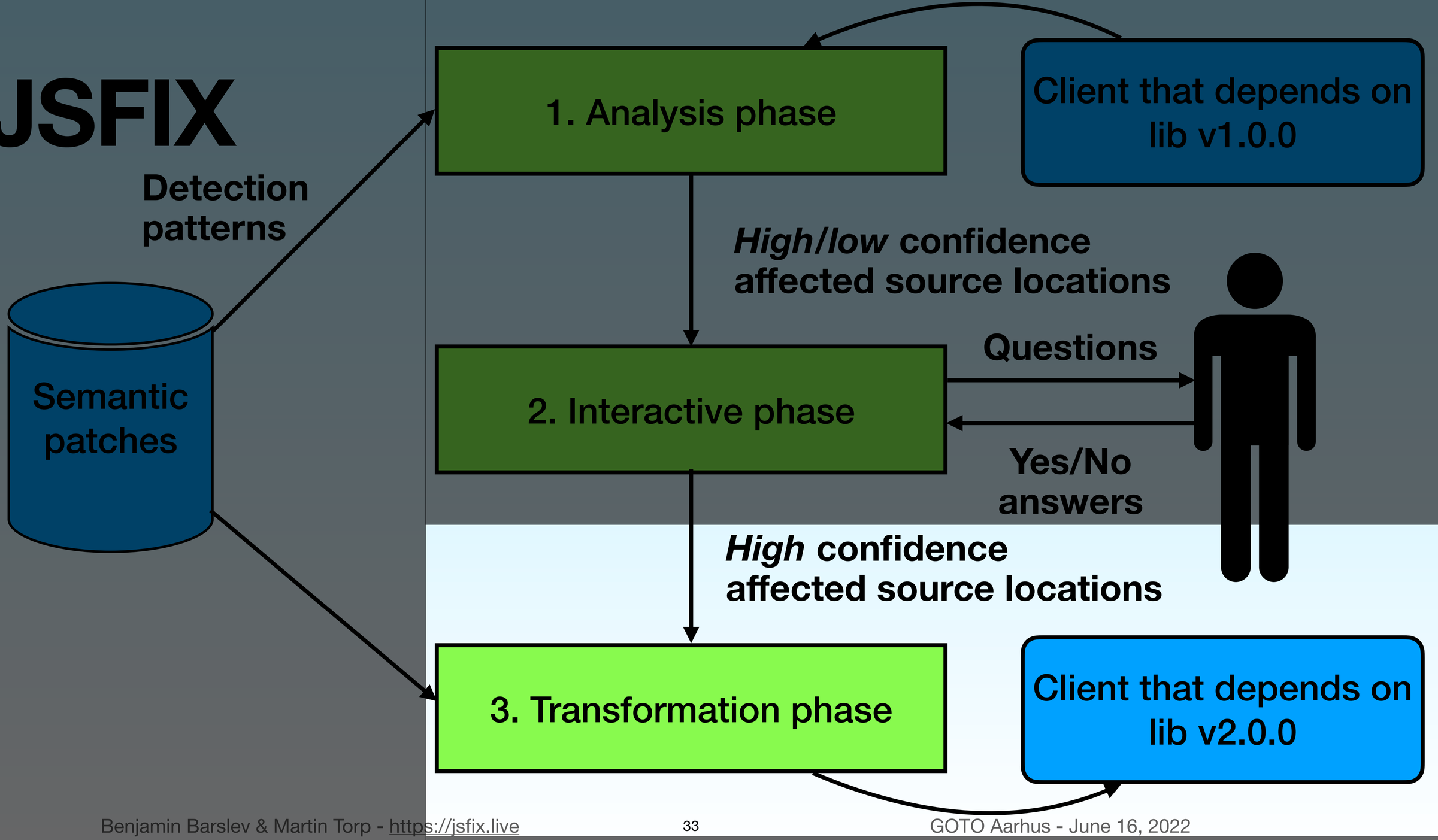
When the analysis lacks precision it asks the user *Yes/No* questions



JSFIX


Detection patterns
1. Analysis phase
Client that depends on lib v1.0.0
High/low confidence affected source locations
Semantic patches
2. Interactive phase
Questions
Yes/No answers
High confidence affected source locations
3. Transformation phase
Client that depends on lib v2.0.0

JSFIX


Detection patterns
Semantic patches
1. Analysis phase
Client that depends on lib v1.0.0
High/low confidence affected source locations
Questions
2. Interactive phase
Yes/No answers
High confidence affected source locations
3. Transformation phase
Client that depends on lib v2.0.0

Third phase: transformation

```
call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))
```

```
import { fromEvent } from 'rxjs';


const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);
```

Third phase: transformation

call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))

```
import { fromEvent } from 'rxjs';


const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);
```

Third phase: transformation

```
call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))
```

```
import { fromEvent } from 'rxjs';


const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);
```

```
clicks
```

Third phase: transformation

```
call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))
```

```
import { fromEvent } from 'rxjs';


const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);
```

```
clicks.pipe(
```

Third phase: transformation

```
call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))
```

```
import { fromEvent } from 'rxjs';

import { map } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);

clicks.pipe(map(
```

Third phase: transformation

```
call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))
```

```
import { fromEvent } from 'rxjs';

import { map } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);

clicks.pipe(map(ev => ev.clientX))
```

Third phase: transformation

call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))

```
import { fromEvent } from 'rxjs';

import { map } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);

clicks.pipe(map(ev => ev.clientX))
```

Third phase: transformation

call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))

```
import { fromEvent } from 'rxjs';

import { map, filter } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');
clicks.map(ev => ev.clientX).filter(x => x > 300);

clicks.pipe(map(ev => ev.clientX)).pipe(filter(x => x > 300))
```

Third phase: transformation

call <rxjs>**.{map,filter} ⇝ $base.pipe(<rxjs/operators>.$prop($args))

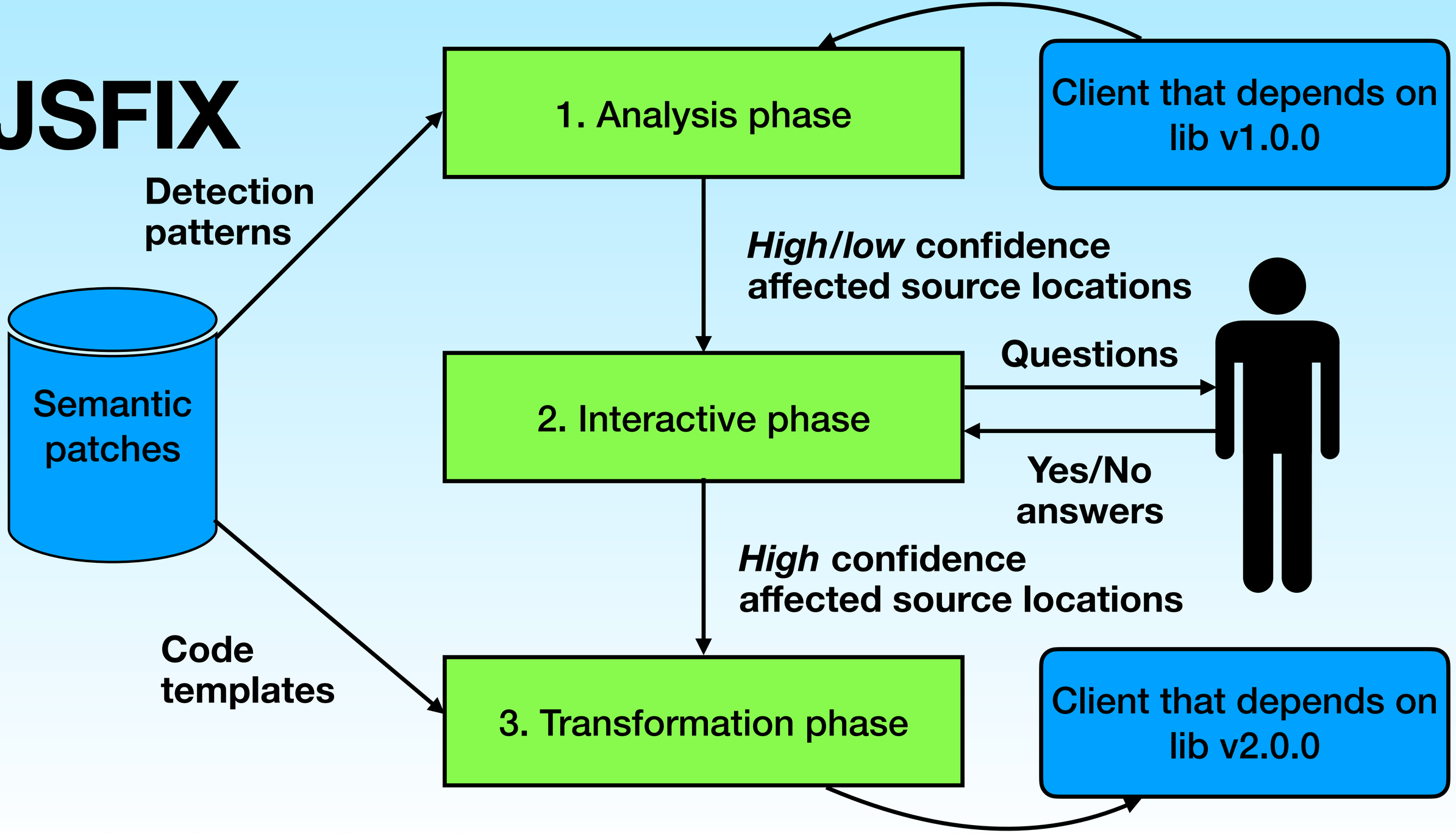
```
import { fromEvent } from 'rxjs';

import { map, filter } from 'rxjs/operators';

const clicks = fromEvent(document, 'click');


clicks.pipe(map(ev => ev.clientX)).pipe(filter(x => x > 300))
```

JSFIX


1. Analysis phase
Client that depends on lib v1.0.0
Detection patterns
High/*low* confidence affected source locations
Semantic patches
Questions
2. Interactive phase
Yes/No answers
High confidence affected source locations
Code templates
3. Transformation phase
Client that depends on lib v2.0.0

Demo - https://jsfix.live/

Collaboration with companies



Upgrading rxjs to v7 at workfeed

Upgrading *rxjs* in Workfeed's Angular app

- 809 files
- 231 changes across 105 files
- 5-10 minutes of work with JSFIX



"By using JSFIX, we were able to reduce the time spent on upgrading rxjs from several hours to just a few minutes.
JSFIX automatically identified and patched more than 100 usages of rxjs in our massive Angular application.
A process, which we would otherwise have had to do manually."

- Palle Hauge Petersen *CTO at Workfeed*

Collaboration with companies



1.
share dependency info



2.
write relevant patches



3.
test JSFIX

Collaboration with companies



1.
share dependency info



2.
write relevant patches



3.
test JSFIX

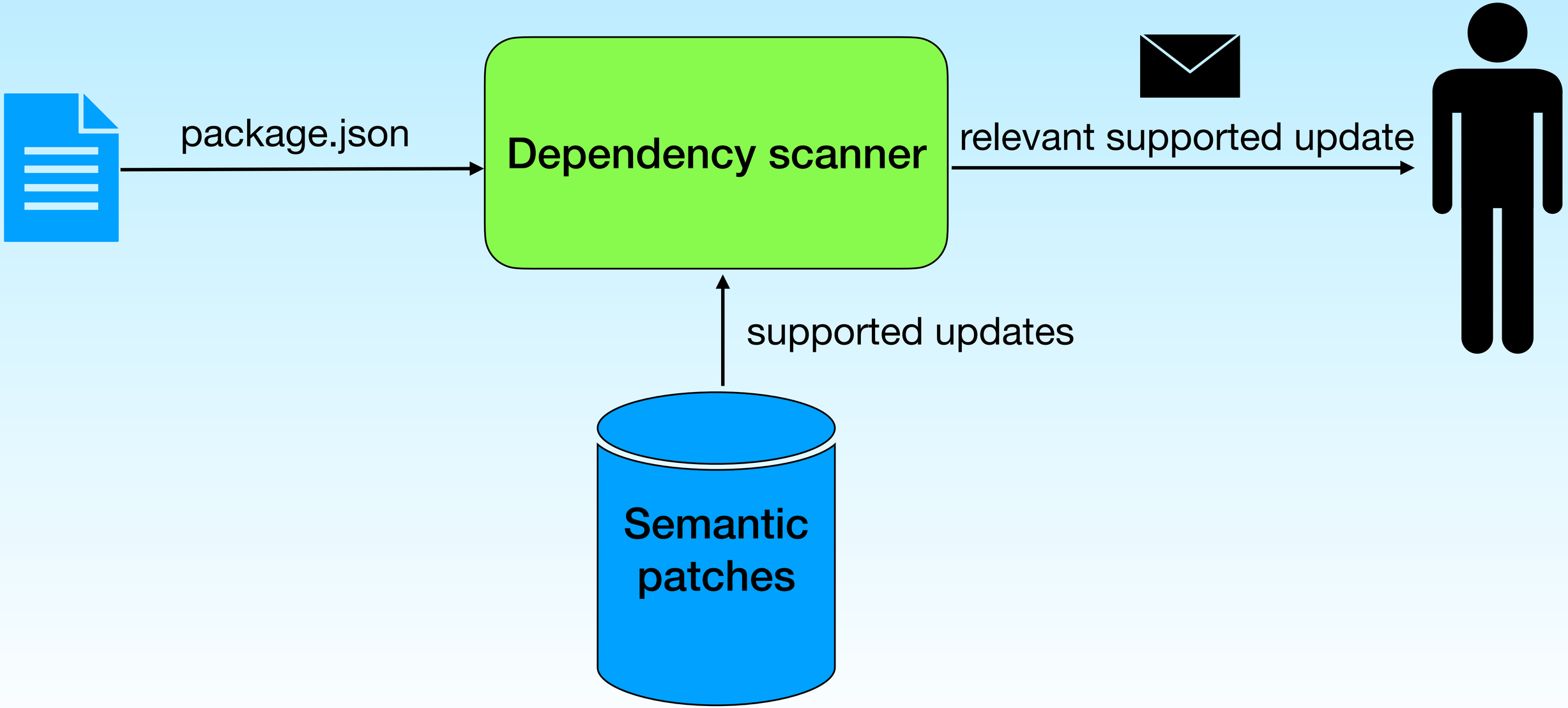
Lessons learned

- Most companies use very outdated dependencies
- Quick and dirty approach:
  Upgrade without changing client code at all, check if all tests still succeed
- Seamless integration of third-party tools required

Demo - JSFIX GitHub app

Installation link: https://github.com/apps/jsfix-updater

Dependency scanner - https://jsfix.live/scanner

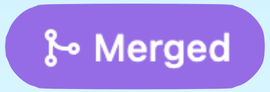


Status and next steps



Status

- Support for 32 packages upgrades (+23 in progress)
- More than 700 pull requests sent 

- Around 70 pull requests merged 









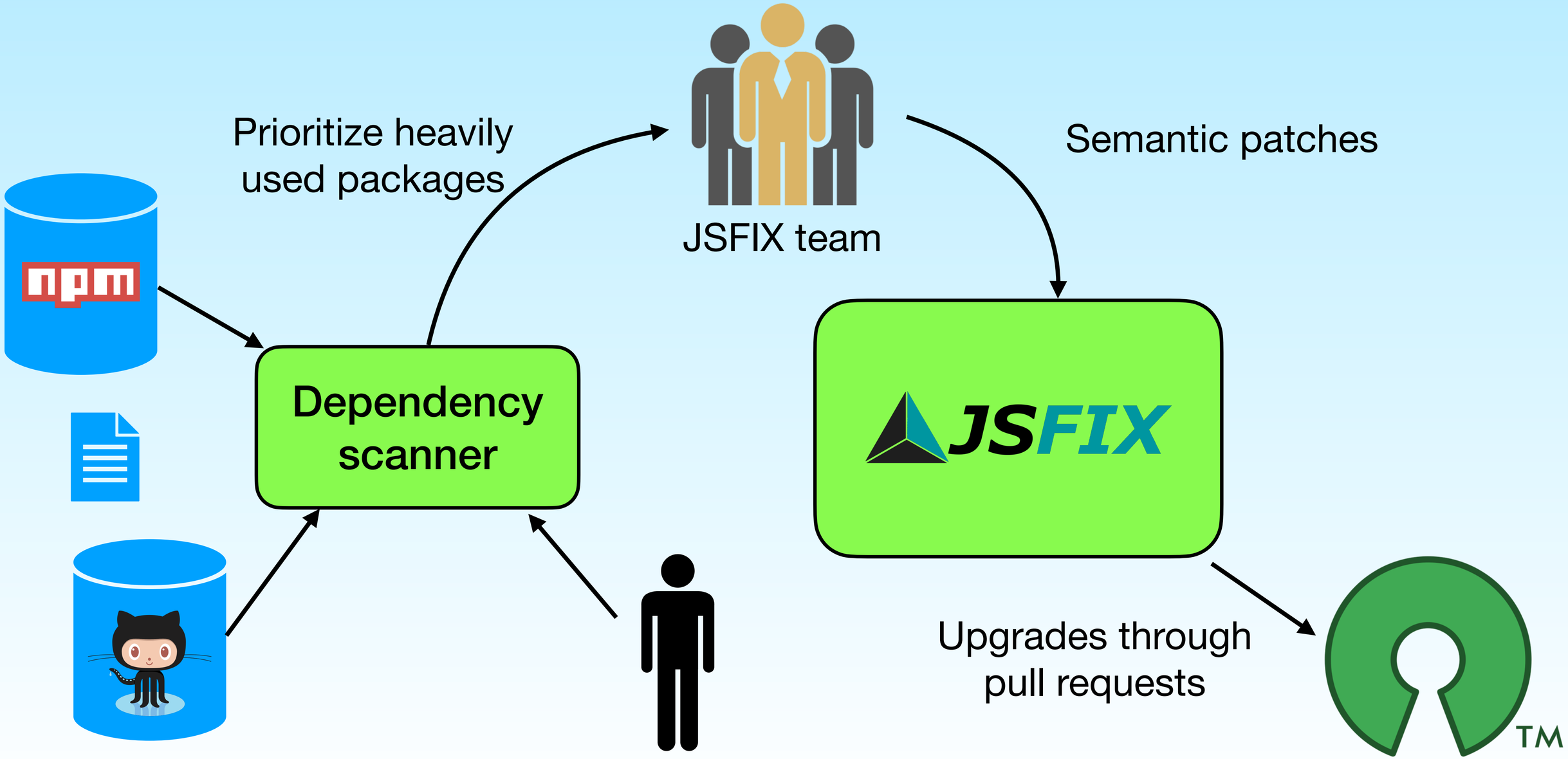








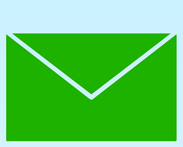
Creating new semantic patches



Ongoing and future work

- Increase automation further
- User-written semantic patches
- JSFIX engine for other usages:
  - Advanced refactoring
  - Dependency overview (usage, size, etc.)
  - Security scanning (auditing with exploit paths)

Join the JSFIX community

- How we can help you keep your npm projects up-to-date
  - Follow JSFIX through    on https://jsfix.live
  - Install Github app: https://github.com/apps/jsfix-updater
  - Signup for dependency scanner: https://jsfix.live/scanner
  - Try JSFIX: https://jsfix.live/jsfix
- Collaborate with us - contact us
    - martin@coana.tech, barslev@coana.tech



goto;

DON'T FORGET TO **RATE THE SESSIONS**

#GOTOaar

Rate a minimum of **5 sessions** and claim your **reward** at the Registration Desk at the Trifork Hall