

To Type or Not to Type: Quantifying Detectable Bugs in JavaScript

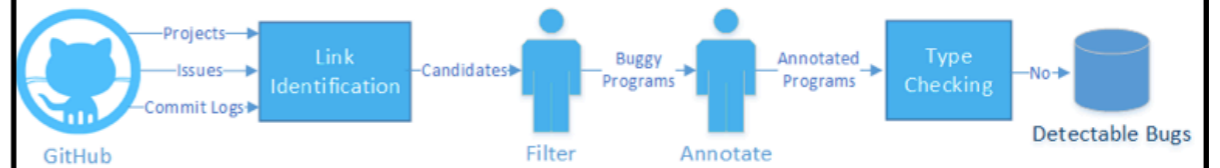
Zheng Gao*, Christian Bird*, Earl Barr*

*University College London, *Microsoft Research

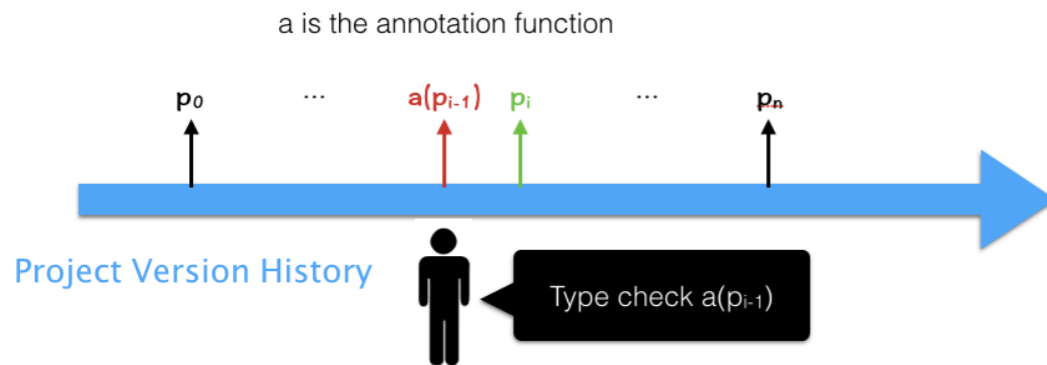
Research Question

What is the percentage of public bugs that are detectable under Flow or TypeScript?

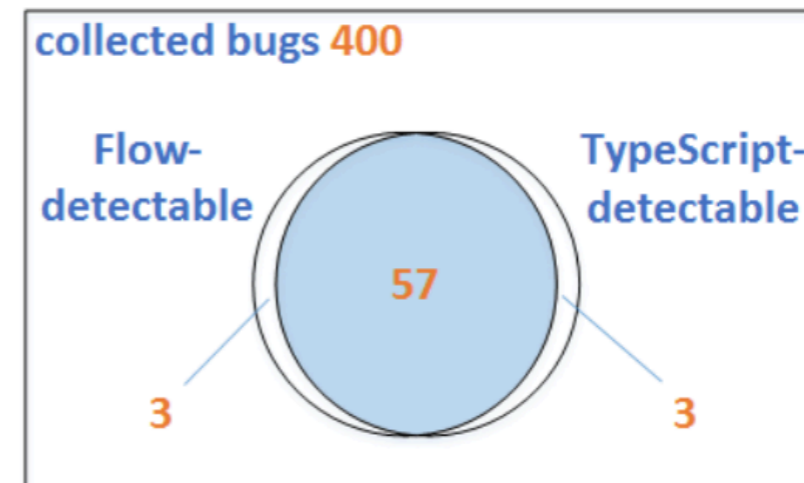
Experiment Overview



Methodology



Results

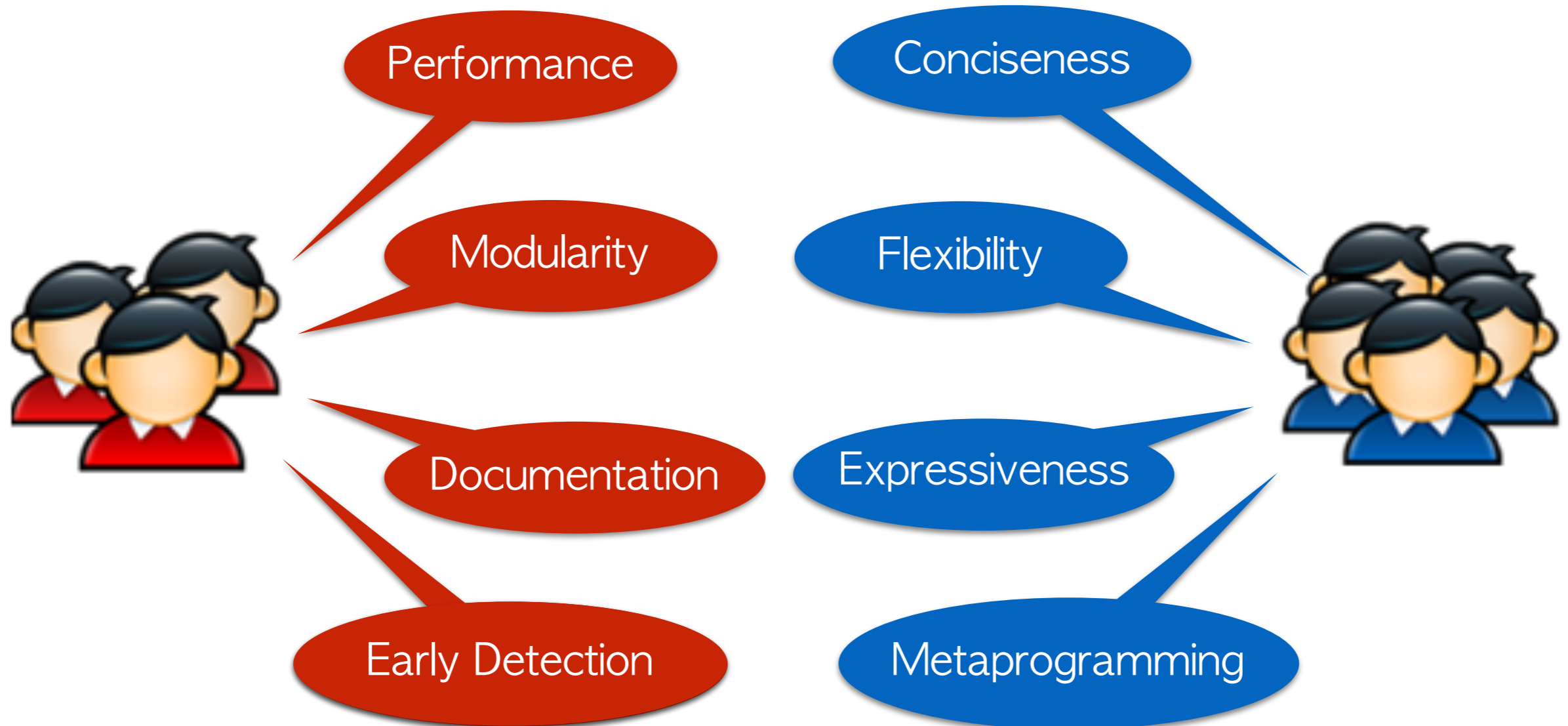


The confidence range for both Flow and TypeScript is **[11.5%,18.5%]**, at a 95% confidence level.

Static Typing vs. Dynamic Typing



Static Typing vs. Dynamic Typing



Static Typing vs. Dynamic Typing



Early Detection



Engine of the Web and



JavaScript

is dynamically typed;

has a large set of long-running projects.

Engine of the Web and



is dynamically typed;

has a large set of long-running projects.

3,599,113 JavaScript repos on GitHub

Static Typing for JavaScript

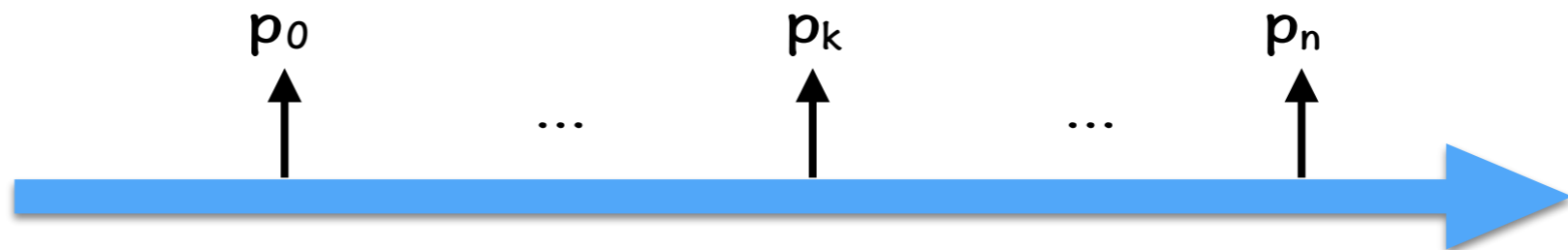
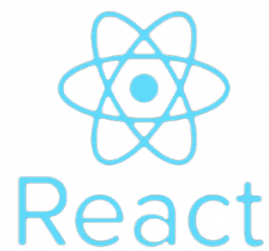
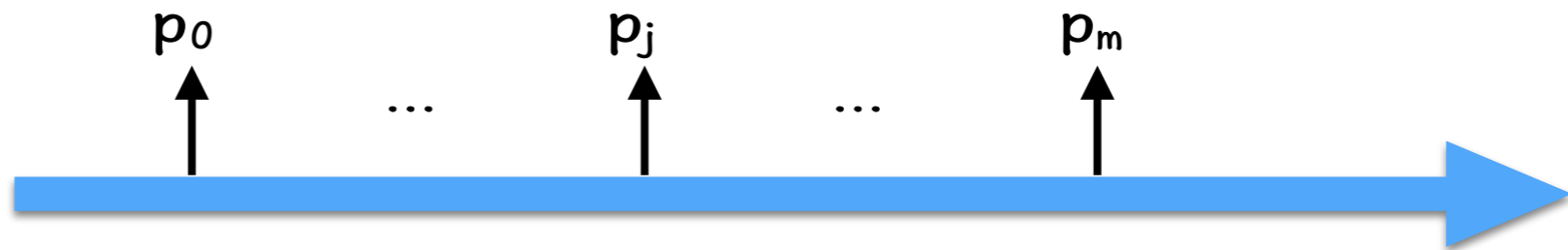
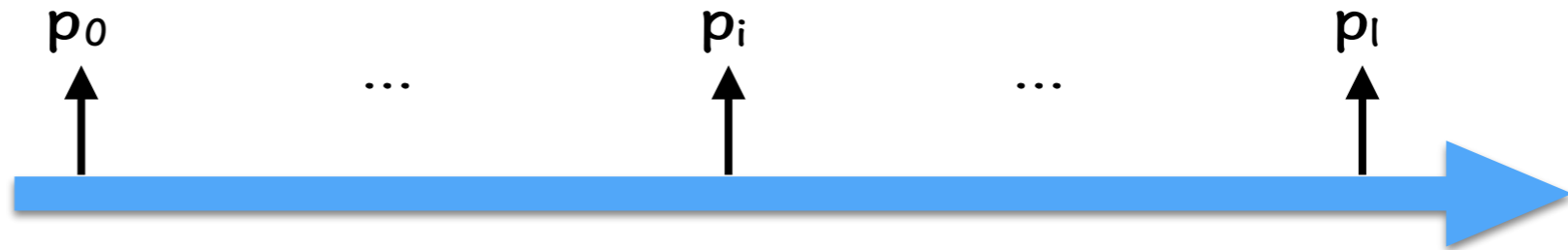


facebook

Static Typing for JavaScript

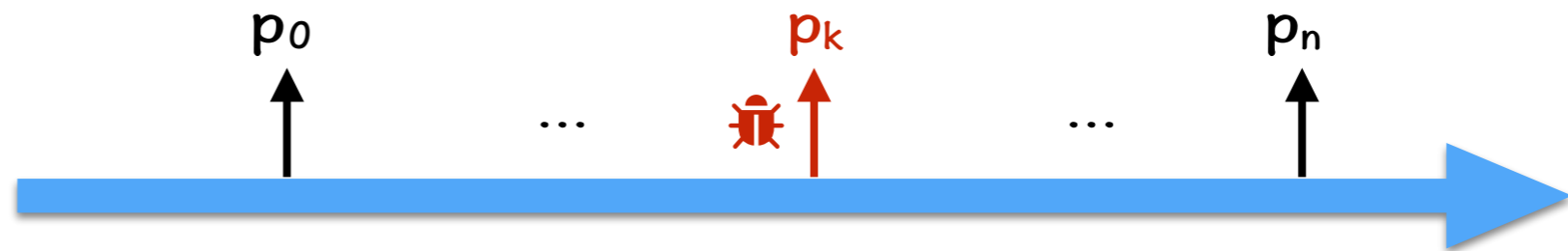
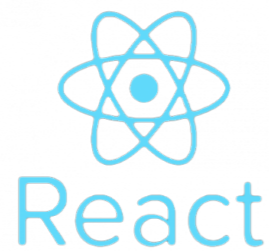
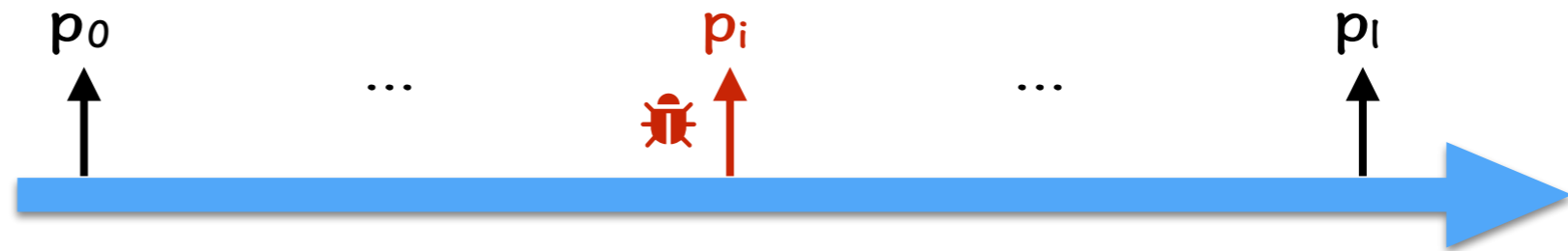


Had Static Typing been Used ...



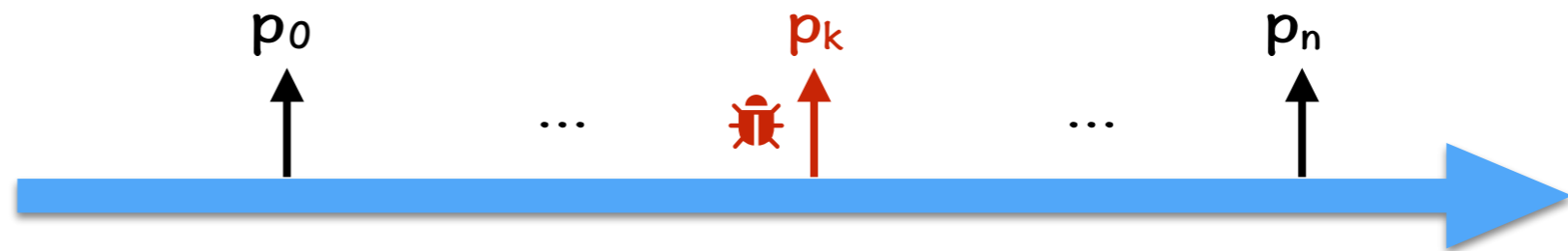
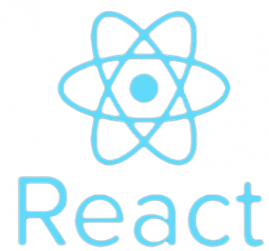
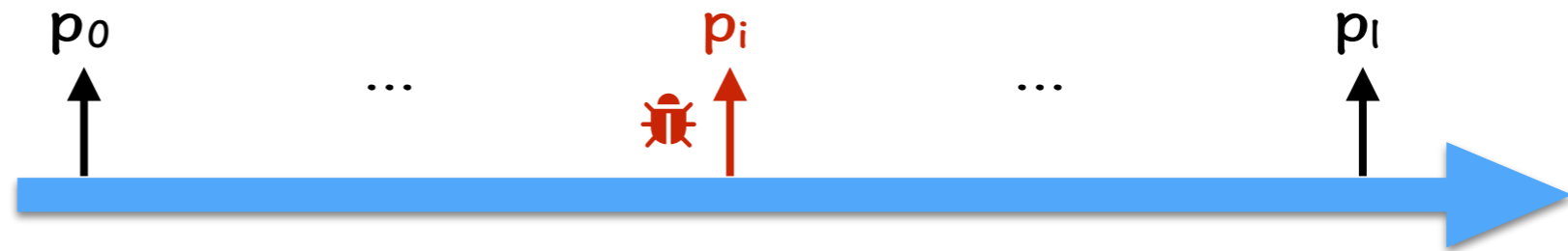
...

Had Static Typing been Used ...



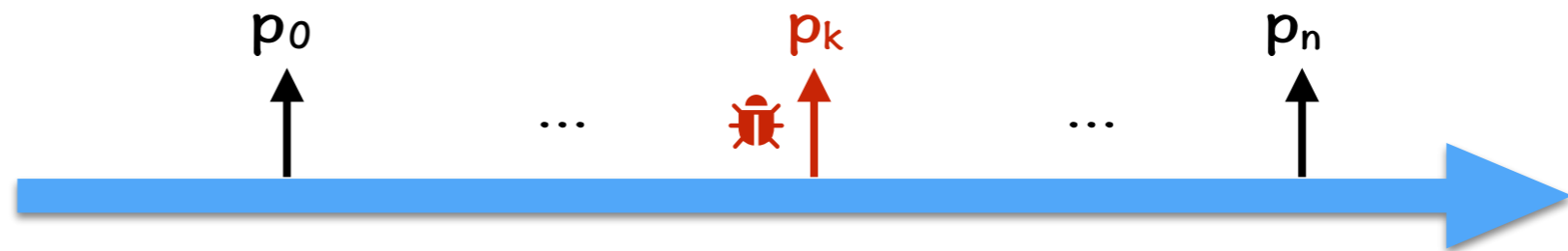
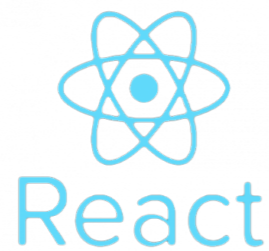
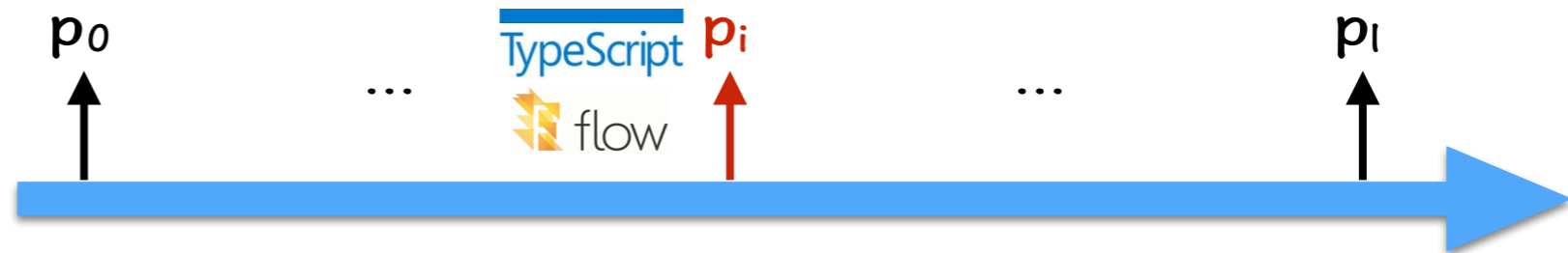
...

Had Static Typing been Used ...



...

Had Static Typing been Used ...



...

Central Finding

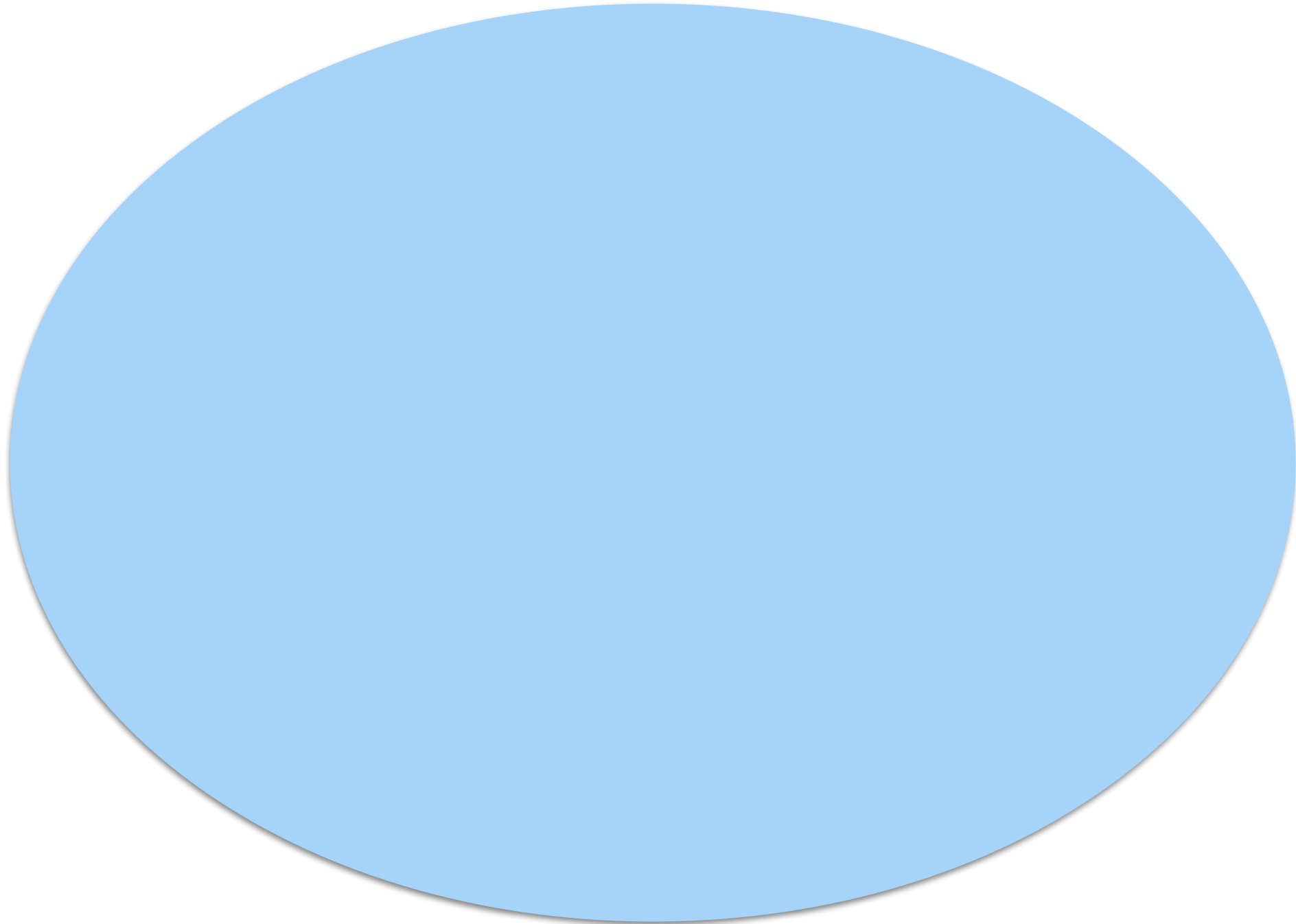
Central Finding

15%

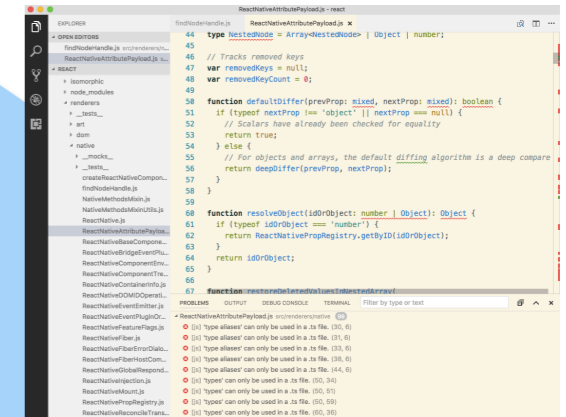
Central Finding



Bug Life Cycle



Bug Life Cycle



The screenshot shows a code editor with the following code:

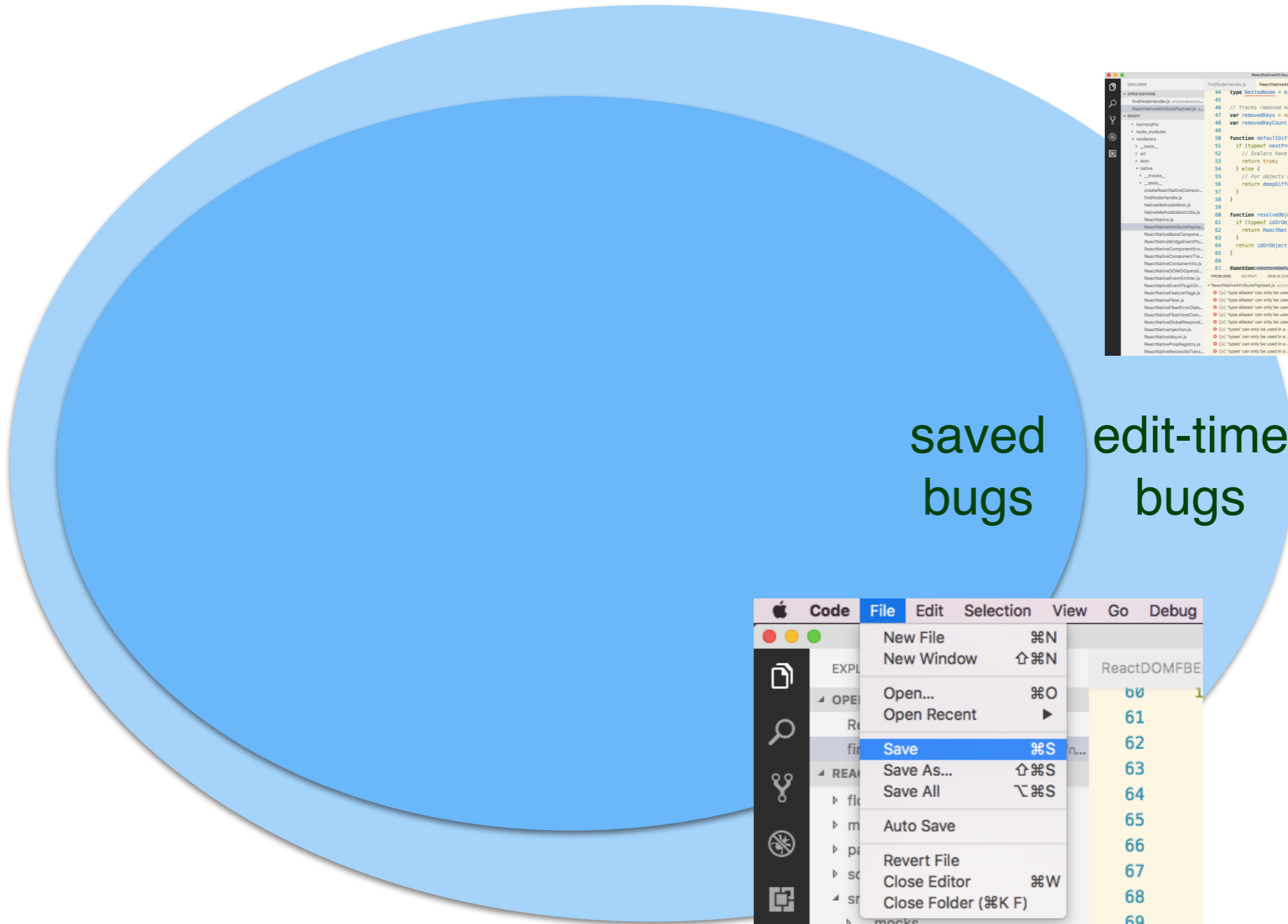
```
44 type NestedNode = Array<NestedNode> | Object | number;
45
46 // Tracks removed keys
47 var removedKeys = null;
48 var removedKeyCount = 0;
49
50 function defaultDiffer(prevProp: mixed, nextProp: mixed): boolean {
51   if (typeof nextProp !== 'object' || nextProp === null) {
52     // Scalars have already been checked for equality
53     return true;
54   } else {
55     // For objects and arrays, the default diffing algorithm is a deep compare
56     return deepDiffer(prevProp, nextProp);
57   }
58 }
59
60 function resolveObject(idObj: number | Object): Object {
61   if (typeof idObj === 'number') {
62     return ReactNativePropRegistry.getID(idObj);
63   }
64   return idObj;
65 }
66
67 function resolveObjectToNestedArray(...)
```

The bottom of the screenshot shows a PROBLEMS panel with several error messages:

- ReactNativePropRegistry.js: (1) 'type aliases' can only be used in a .ts file. (31, 6)
- ReactNativePropRegistry.js: (1) 'type aliases' can only be used in a .ts file. (33, 6)
- ReactNativePropRegistry.js: (1) 'type aliases' can only be used in a .ts file. (38, 6)
- ReactNativePropRegistry.js: (1) 'type aliases' can only be used in a .ts file. (44, 6)
- ReactNativePropRegistry.js: (1) 'type aliases' can only be used in a .ts file. (50, 34)
- ReactNativePropRegistry.js: (1) 'type aliases' can only be used in a .ts file. (50, 51)
- ReactNativePropRegistry.js: (1) 'type aliases' can only be used in a .ts file. (50, 59)
- ReactNativePropRegistry.js: (1) 'type aliases' can only be used in a .ts file. (50, 38)

edit-time bugs

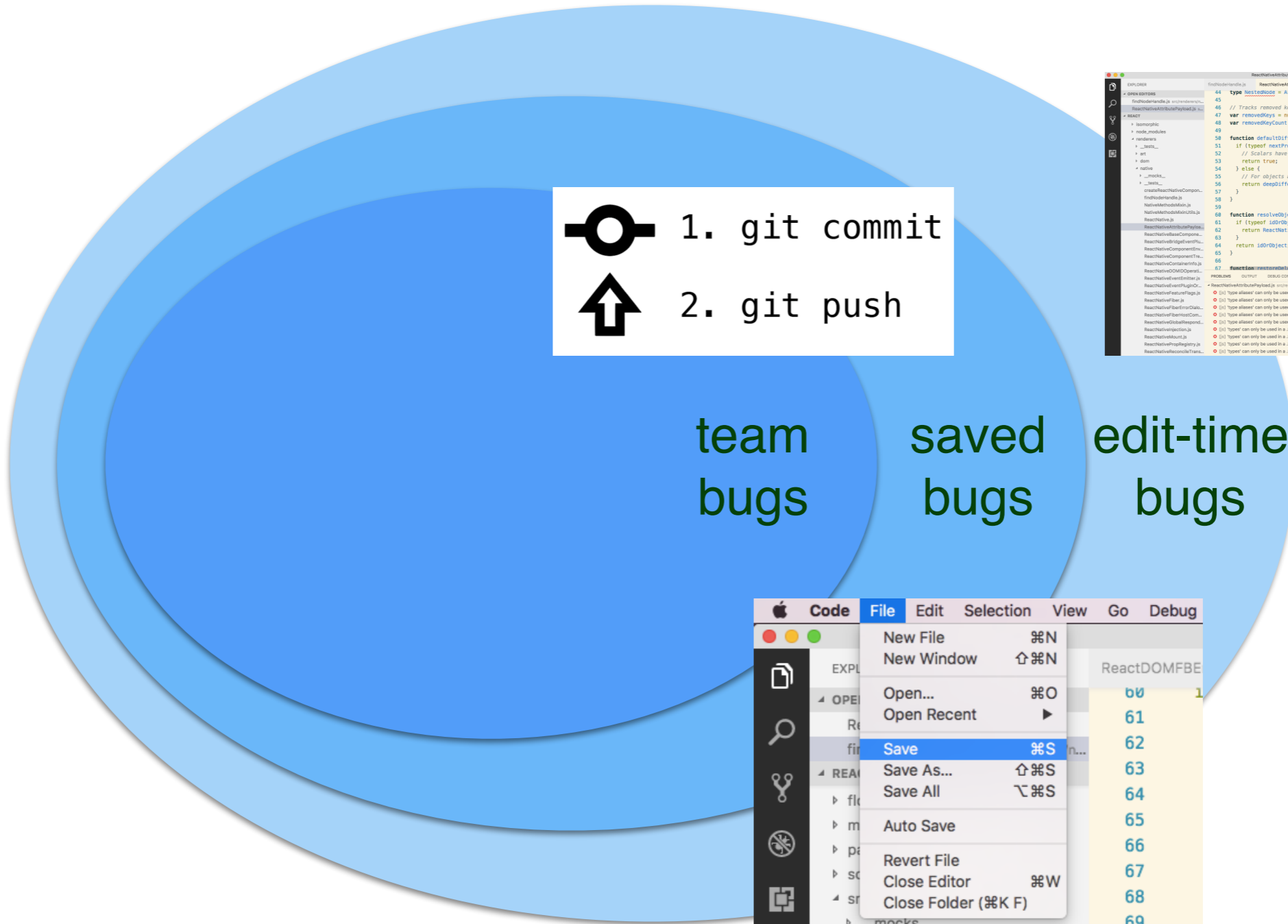
Bug Life Cycle



```
44 type NestedNode = Array<NestedNode> | Object | number;
45
46 // Tracks removed keys
47 var removedKeys = null;
48 var removedKeyCount = 0;
49
50 function defaultDiffer(prevProp: mixed, nextProp: mixed): boolean {
51   if (typeof nextProp !== 'object' || nextProp === null) {
52     // Scalars have already been checked for equality
53     return true;
54   } else {
55     // For objects and arrays, the default diffing algorithm is a deep compare
56     return deepDiffer(prevProp, nextProp);
57   }
58 }
59
60 function resolveObject(id: number | Object): Object {
61   if (typeof id !== 'number') {
62     return ReactNativePropRegistry.getID(id);
63   }
64   return id;
65 }
66
67 function resolveObjectID(id: number): Object {
68   return ReactNativePropRegistry.getID(id);
69 }
```

```
Code File Edit Selection View Go Debug
New File ⌘N
New Window ⇧⌘N
Open... ⌘O
Open Recent ▶
Save ⌘S
Save As... ⇧⌘S
Save All ⌘⌘S
Auto Save
Revert File
Close Editor ⌘W
Close Folder (⌘K F)
ReactDOMFB...
00
61
62
63
64
65
66
67
68
69
```

Bug Life Cycle



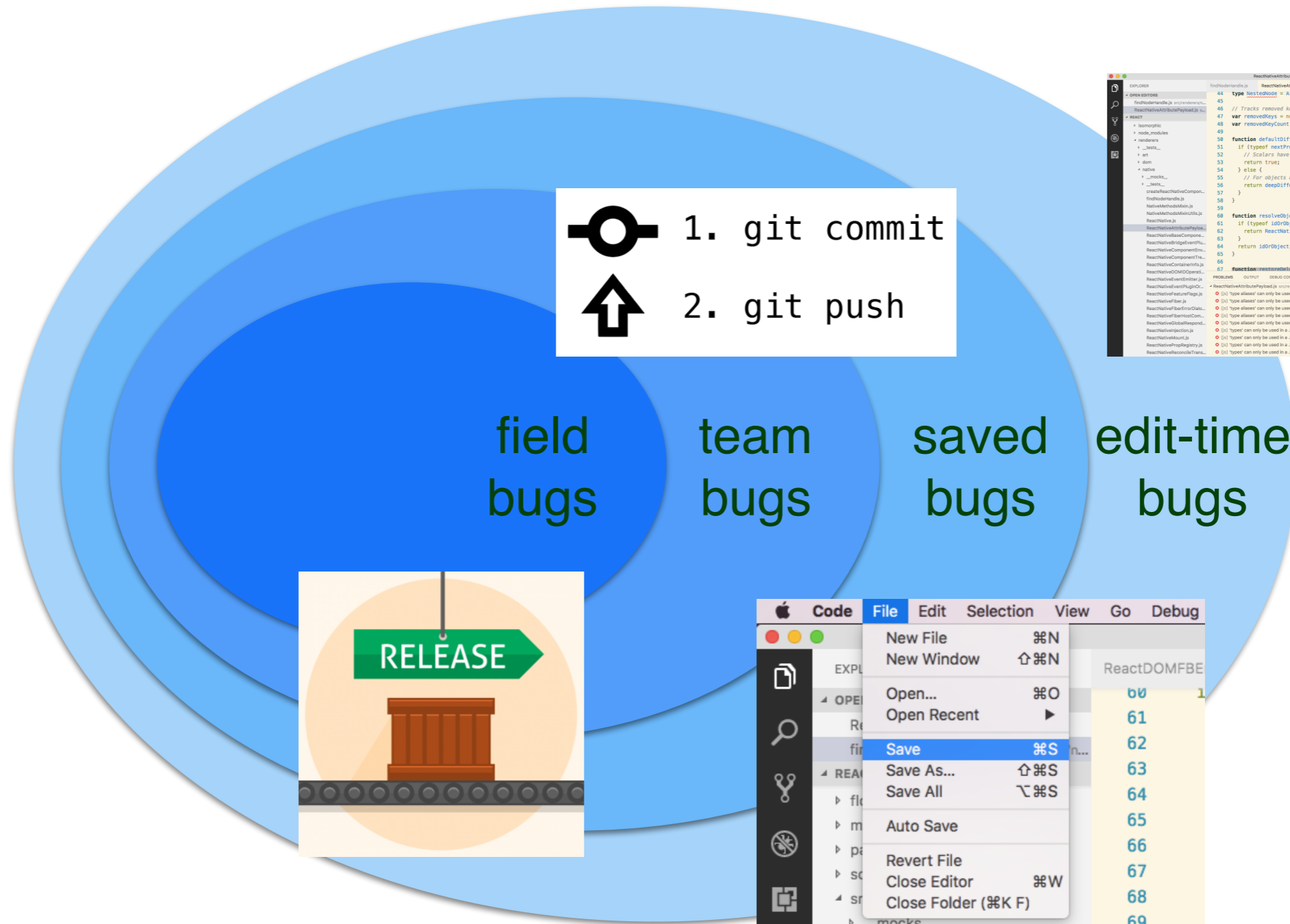
```
44 type NestedNode = Array<NestedNode> | Object | number;
45
46 // Tracks removed keys
47 var removedKeys = null;
48 var removedKeyCount = 0;
49
50 function defaultDiffer(prevProp: mixed, nextProp: mixed): boolean {
51   if (typeof nextProp !== 'object' || nextProp === null) {
52     // Scalars have already been checked for equality
53     return true;
54   } else {
55     // For objects and arrays, the default diffing algorithm is a deep compare
56     return deepDiffer(prevProp, nextProp);
57   }
58 }
59
60 function resolveObject(id: number | Object): Object {
61   if (typeof id === 'number') {
62     return ReactNativePropRegistry.getID(id);
63   }
64   return id;
65 }
66
67 function resolveNestedArray(id: number | Object): Array<NestedNode> {
68   if (typeof id === 'number') {
69     return ReactNativePropRegistry.getID(id);
70   }
71   return id;
72 }
73
74 function resolveNestedObject(id: number | Object): Object {
75   if (typeof id === 'number') {
76     return ReactNativePropRegistry.getID(id);
77   }
78   return id;
79 }
80
81 function resolveNestedArray(id: number | Object): Array<NestedNode> {
82   if (typeof id === 'number') {
83     return ReactNativePropRegistry.getID(id);
84   }
85   return id;
86 }
87
88 function resolveNestedObject(id: number | Object): Object {
89   if (typeof id === 'number') {
90     return ReactNativePropRegistry.getID(id);
91   }
92   return id;
93 }
94
95 function resolveNestedArray(id: number | Object): Array<NestedNode> {
96   if (typeof id === 'number') {
97     return ReactNativePropRegistry.getID(id);
98   }
99   return id;
100 }
```

Code File Edit Selection View Go Debug

- New File ⌘N
- New Window ⇧⌘N
- Open... ⌘O
- Open Recent ▶
- Save ⌘S**
- Save As... ⇧⌘S
- Save All ⇧⌘S
- Auto Save
- Revert File
- Close Editor ⌘W
- Close Folder (⌘K F)

```
ReactDOMFB...
00
61
62
63
64
65
66
67
68
69
```

Bug Life Cycle

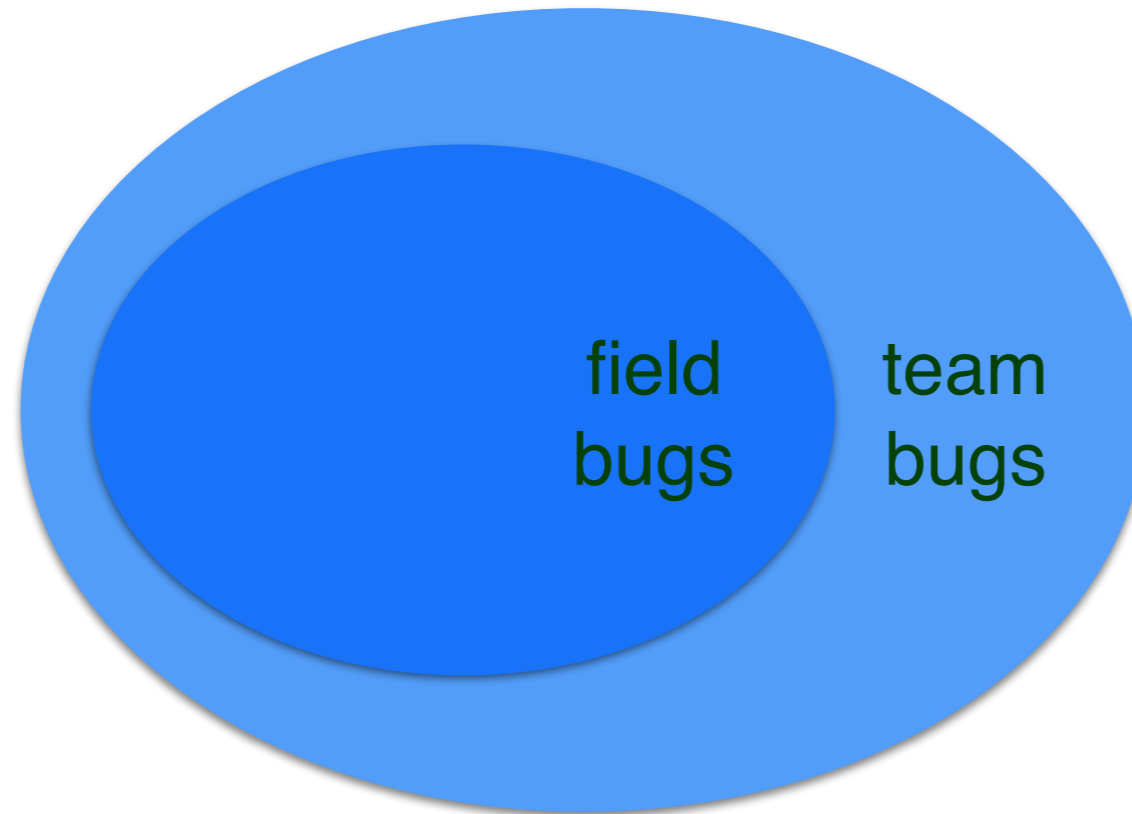


```
44 type NestedNode = Array<NestedNode> | Object | number;
45
46 // Tracks removed keys
47 var removedKeys = null;
48 var removedKeyCount = 0;
49
50 function defaultDiffer(prevProp: mixed, nextProp: mixed): boolean {
51   if (typeof nextProp !== 'object' || nextProp === null) {
52     // Scalars have already been checked for equality
53     return true;
54   } else {
55     // For objects and arrays, the default diffing algorithm is a deep compare
56     return deepDiffer(prevProp, nextProp);
57   }
58 }
59
60 function resolveObject(id: number | Object): Object {
61   if (typeof id === 'number') {
62     return ReactNativePropRegistry.getID(id);
63   }
64   return id;
65 }
66
67 function resolveNestedArray(nestedArray: Array<NestedNode>): Array<NestedNode> {
68   return nestedArray.map(resolveObject);
69 }
```

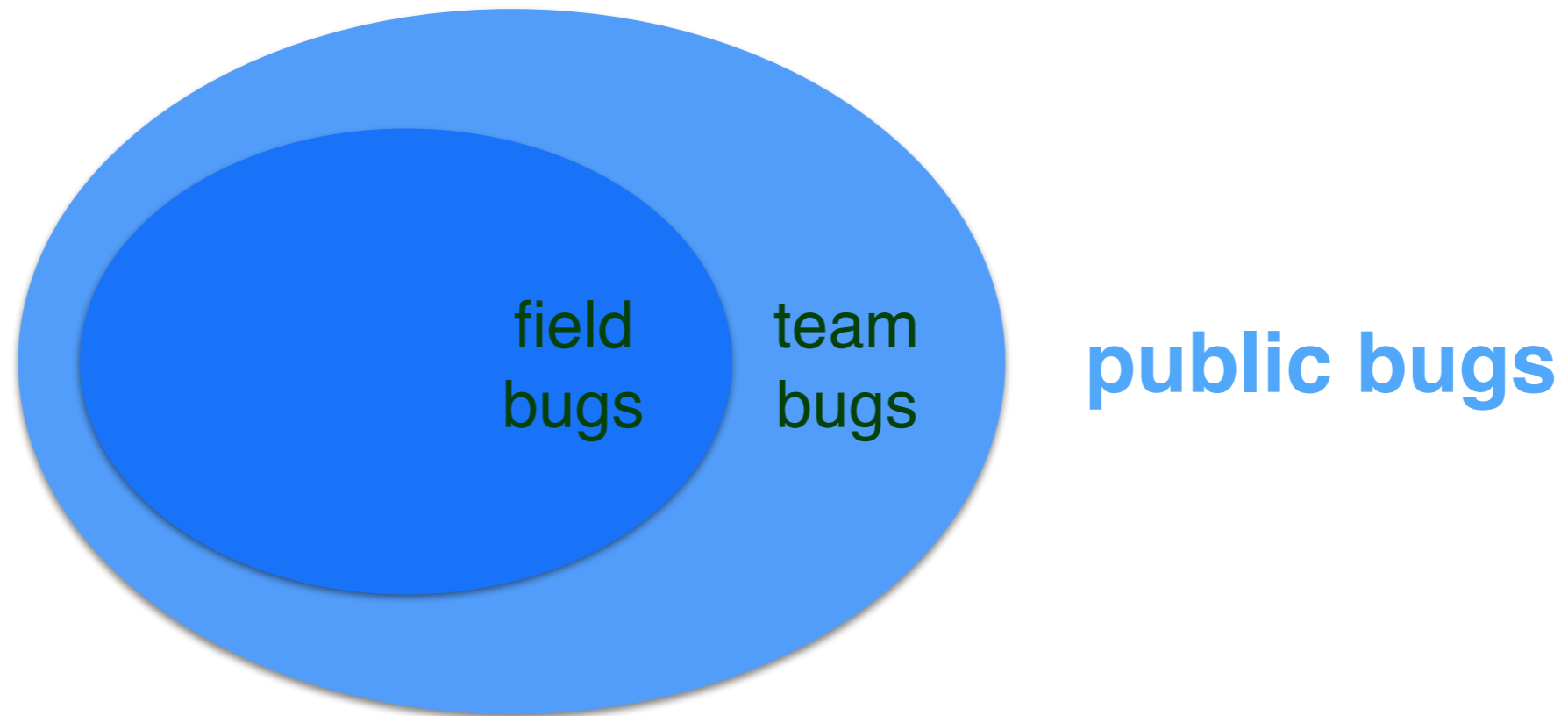
```
Code File Edit Selection View Go Debug
New File ⌘N
New Window ⇧⌘N
Open... ⌘O
Open Recent
Save ⌘S
Save As... ⇧⌘S
Save All ⇧⌘S
Auto Save
Revert File
Close Editor ⌘W
Close Folder (⌘K F)
```

60
61
62
63
64
65
66
67
68
69

Bug Life Cycle



Bug Life Cycle



Type System Detectable

Definition (*ts-detectable*): Given a static type system *ts*, a bug is *ts-detectable* when

1. adding or changing type annotations causes the program containing the bug to fail to type check on a line a fix changes.

Problem

When the type of `b` is nullable `number`, annotating

```
var a = b + 1;
```

to

```
var a : boolean = b + 1;
```

“trivially” triggers a type error.

Consistency

Definition (Consistency): The added or changed type annotations are consistent with a fixed version of the program containing the bug f , if they carried to f type check, and the type of every annotated term is a supertype of that term's type when an oracle precisely annotates it in f .

Type System Detectable

Definition (*ts-detectable*): Given a static type system *ts*, a bug is *ts-detectable* when

1. adding or changing type annotations causes the program containing the bug to fail to type check on a line a fix changes;
2. the new annotations are *consistent* with a fixed version of the program containing the bug.

Example of Detection

```
1 // addNumbers in JavaScript
2 function addNumbers(x, y) {
3     return x + y;
4 }
5 console.log(addNumbers(3, "0"));
```

Error-free in JavaScript,
and unexpectedly displays
an string, 30

```
1 // addNumbers in TypeScript
2 function addNumbers(x:number, y:number) {
3     return x + y;
4 }
5 console.log(addNumbers(3, "0"));
```

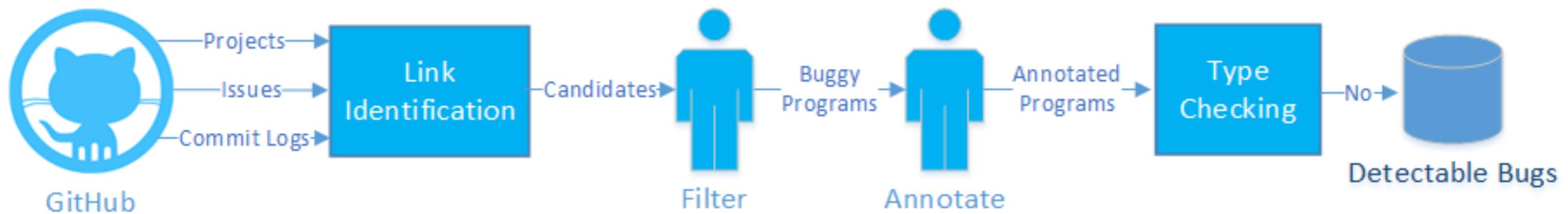
TypeScript throws the
following error:

```
t.ts(5,27): error TS2345: Argument of type 'string' is not  
assignable to parameter of type 'number'.
```

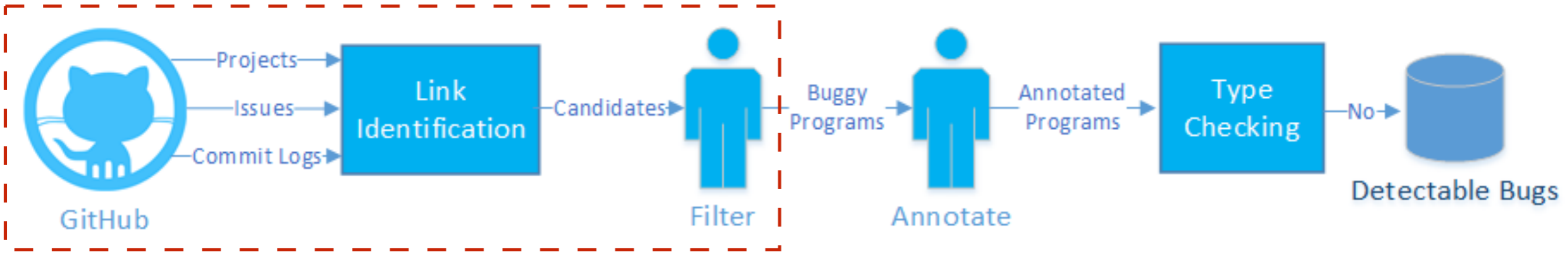
Research Question

What percentage of public bugs are detectable under Flow or TypeScript?

Experiment Overview

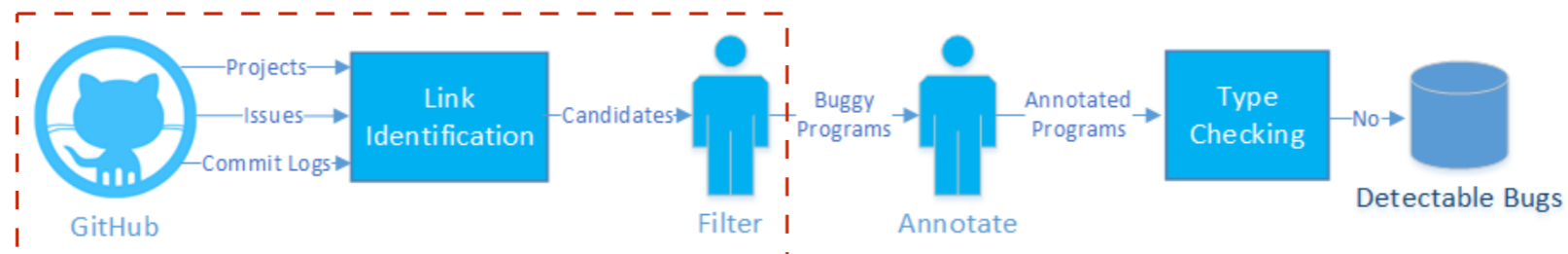


Corpus Collection



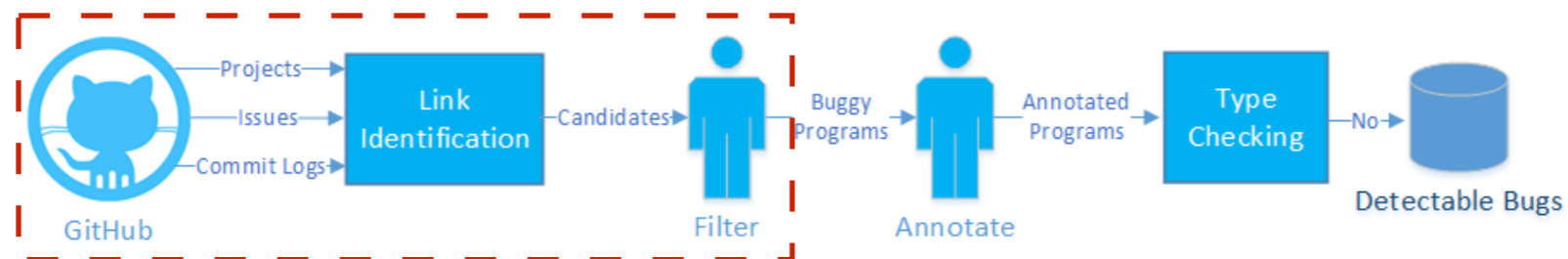
Corpus Collection

- ▶ What is the sample size?
- ▶ How to identify public bugs?

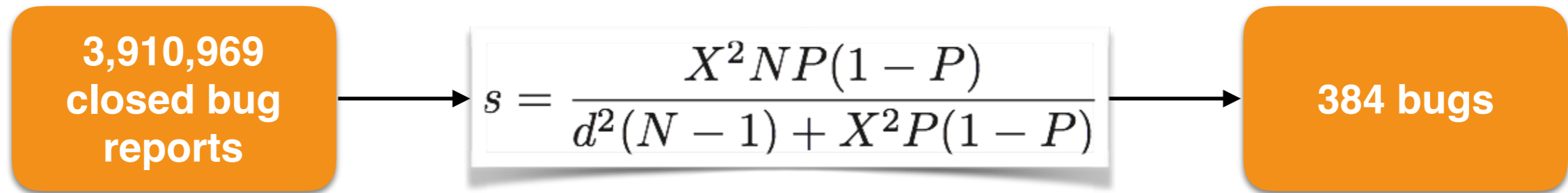


Corpus Collection

- ▶ What is the sample size?
- ▶ How to identify public bugs?



Sample Size Calculation



s: sample size

X^2 : a constant for the confidence level of 95%

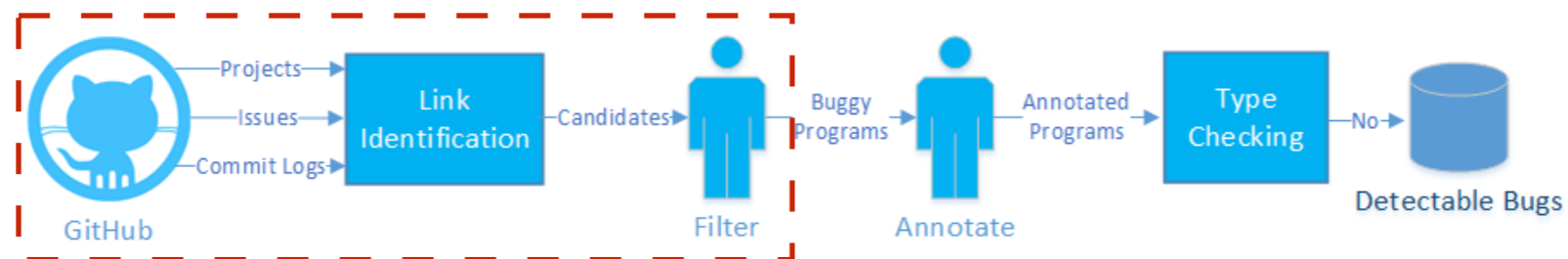
N: population size, 3910969

P: population proportion, 0.5

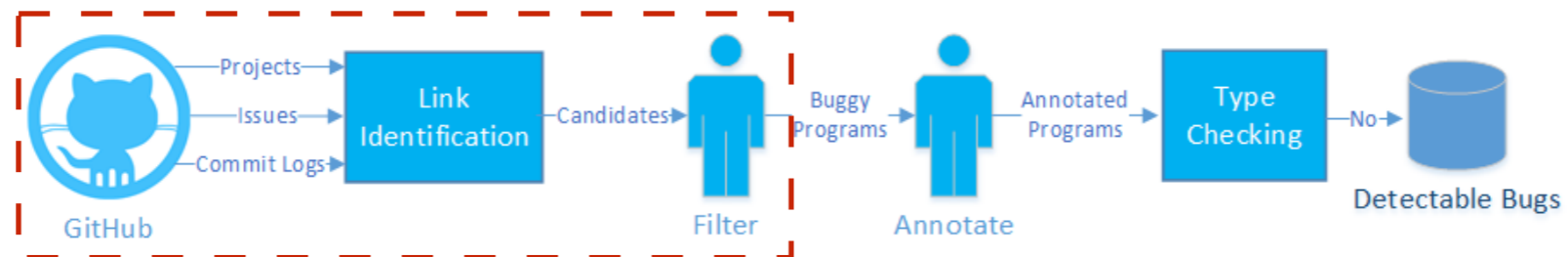
d: degree of accuracy, 0.05

Corpus Collection

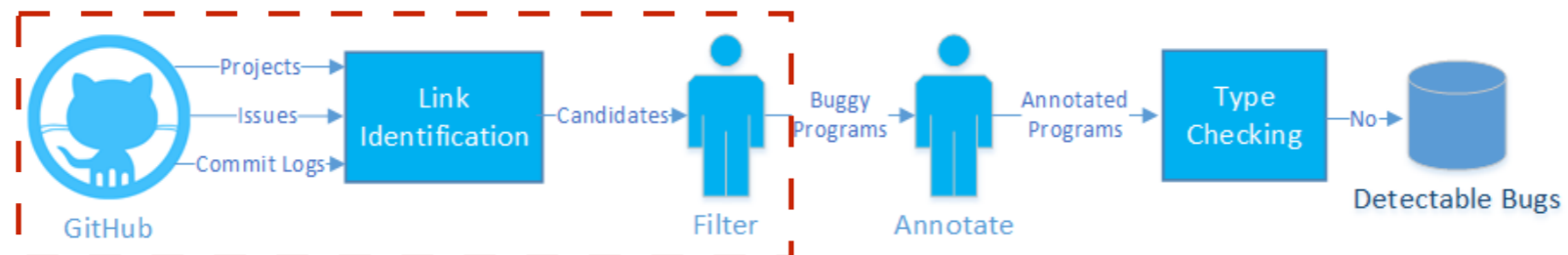
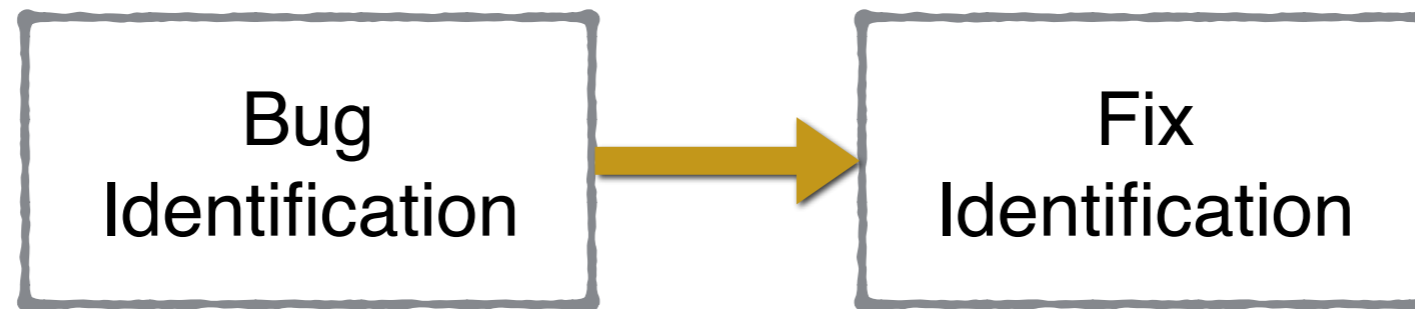
- ▶ What is the sample size?
- ▶ How to identify public bugs?



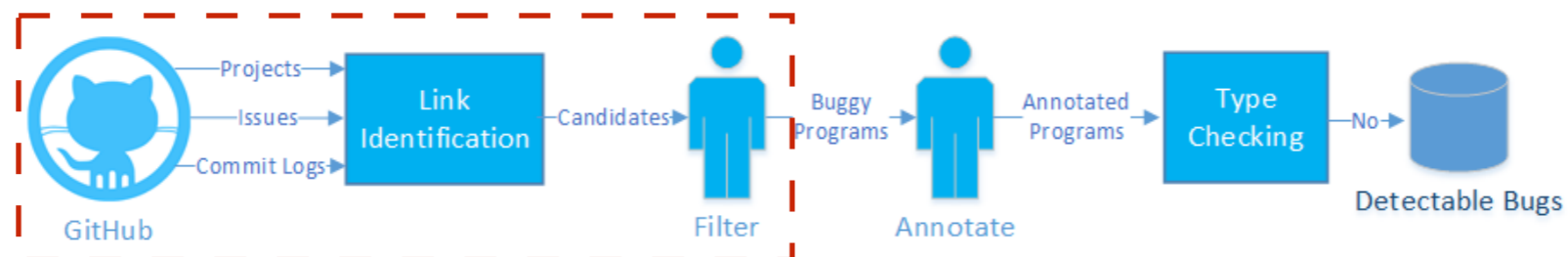
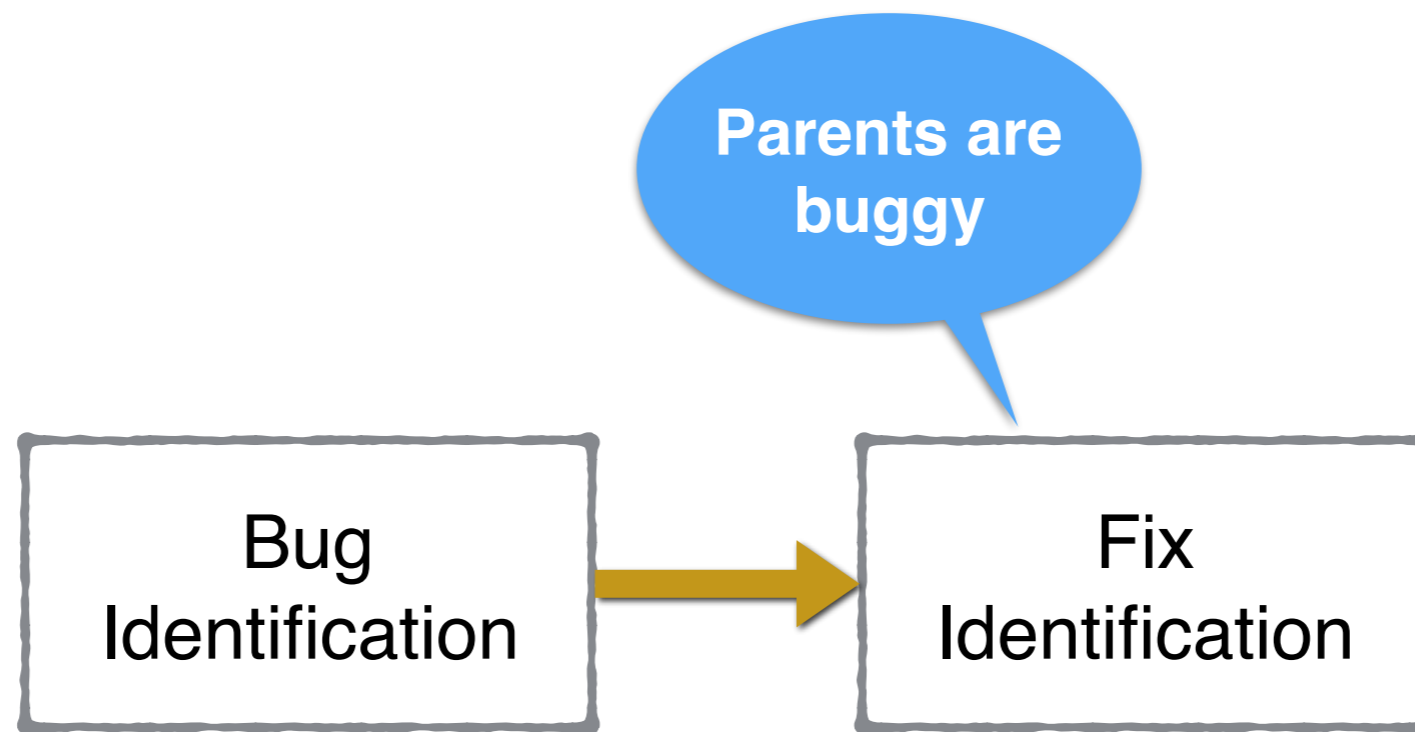
Bug Identification



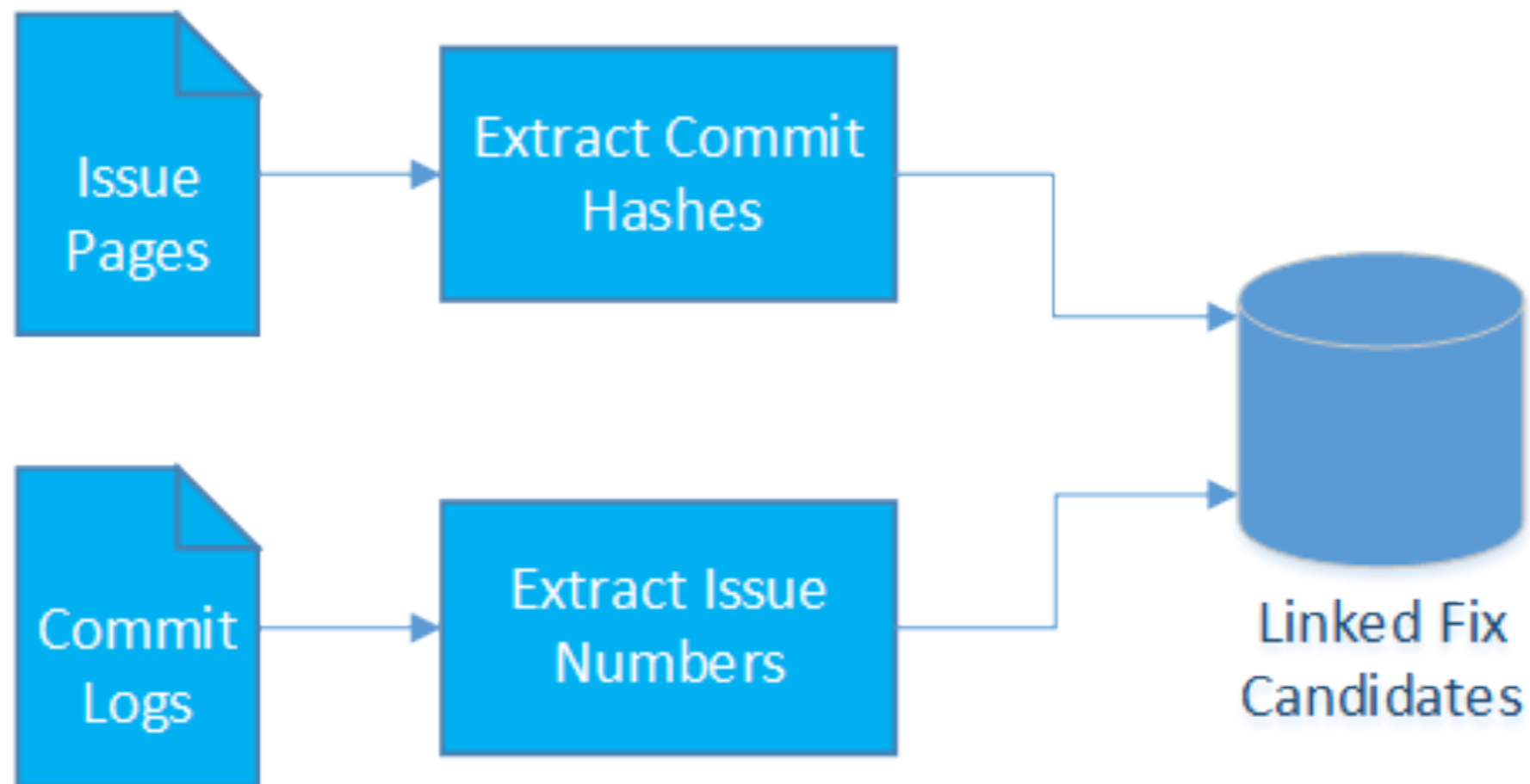
Bug Identification



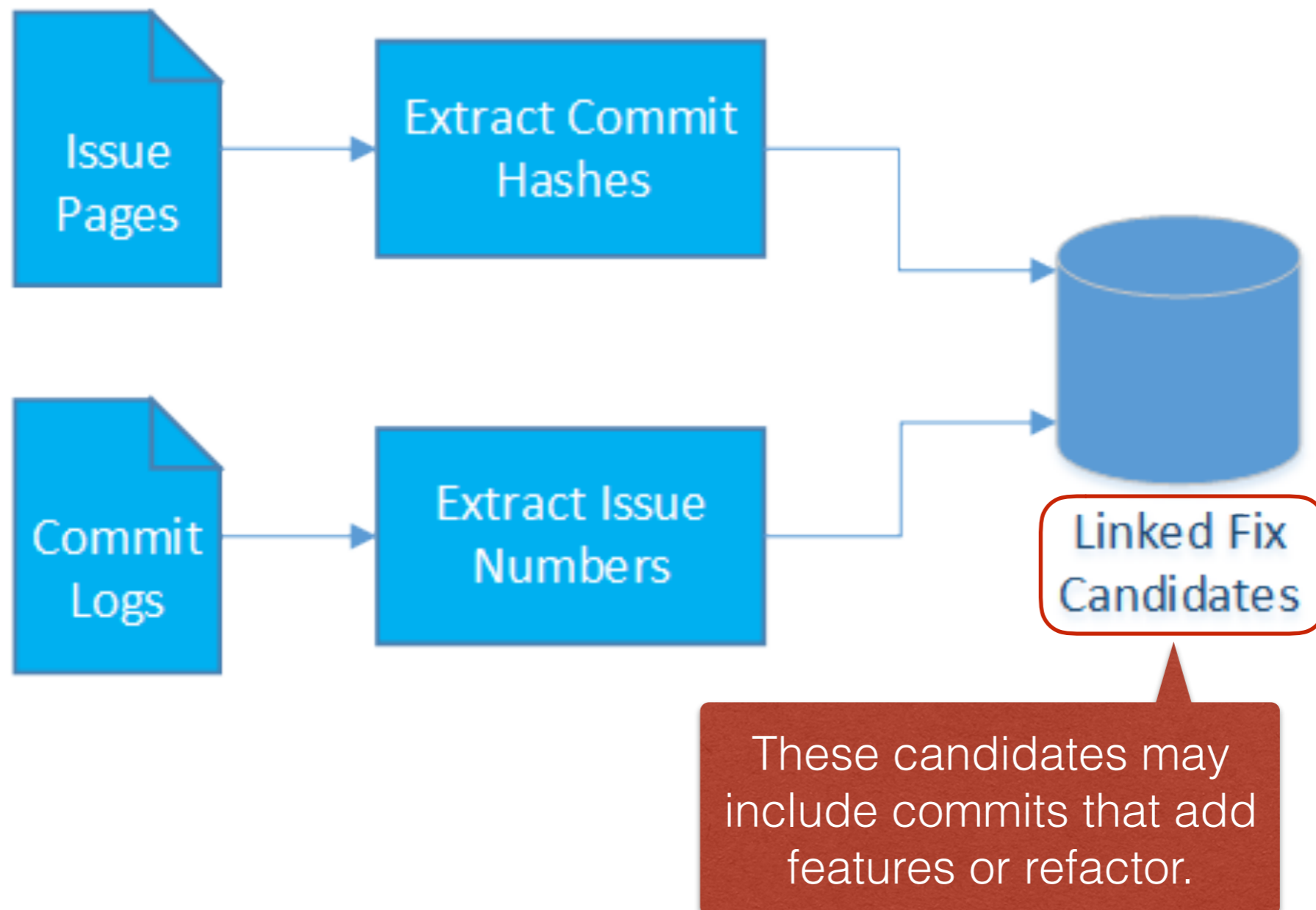
Bug Identification



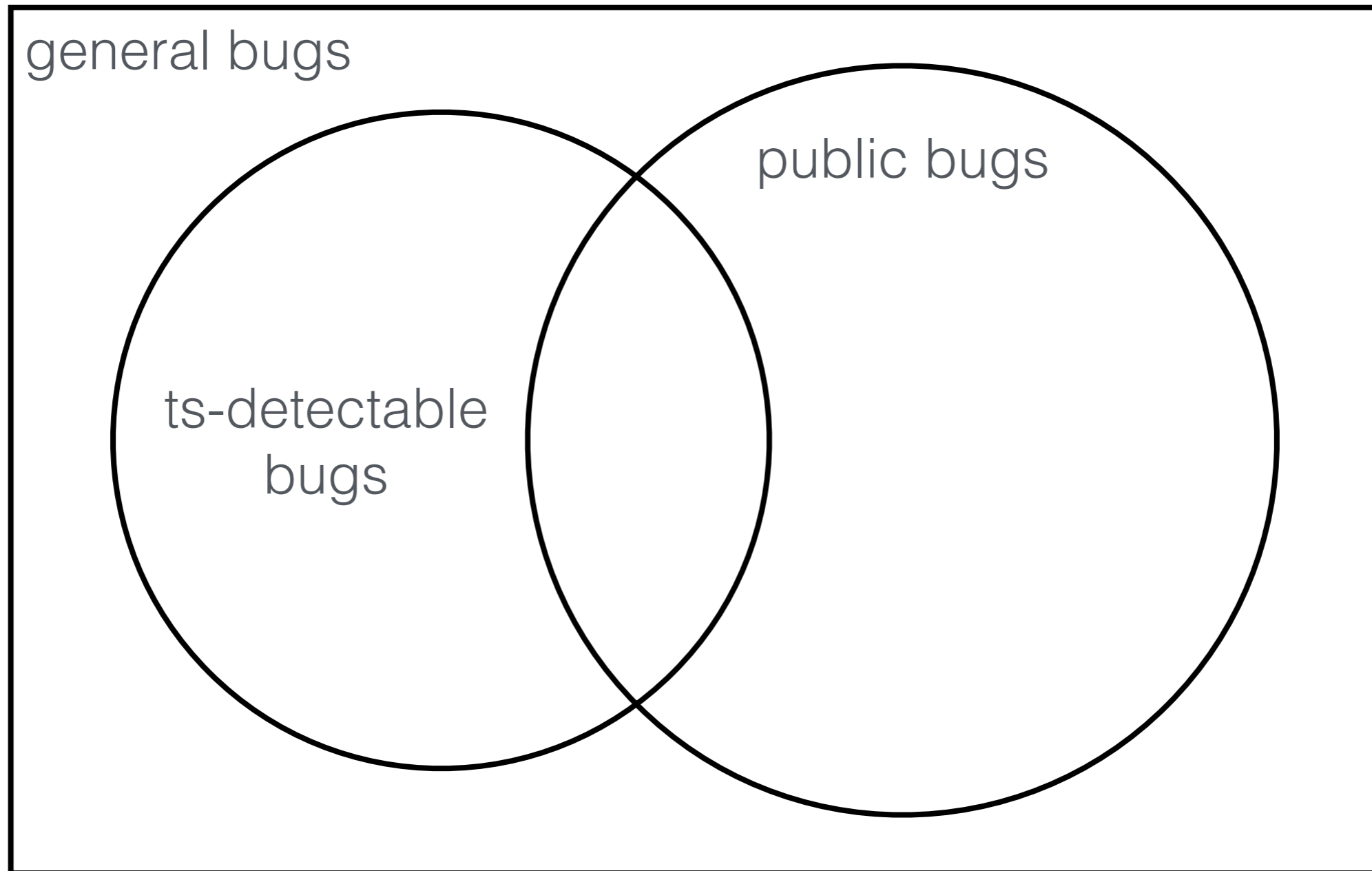
Fix Identification



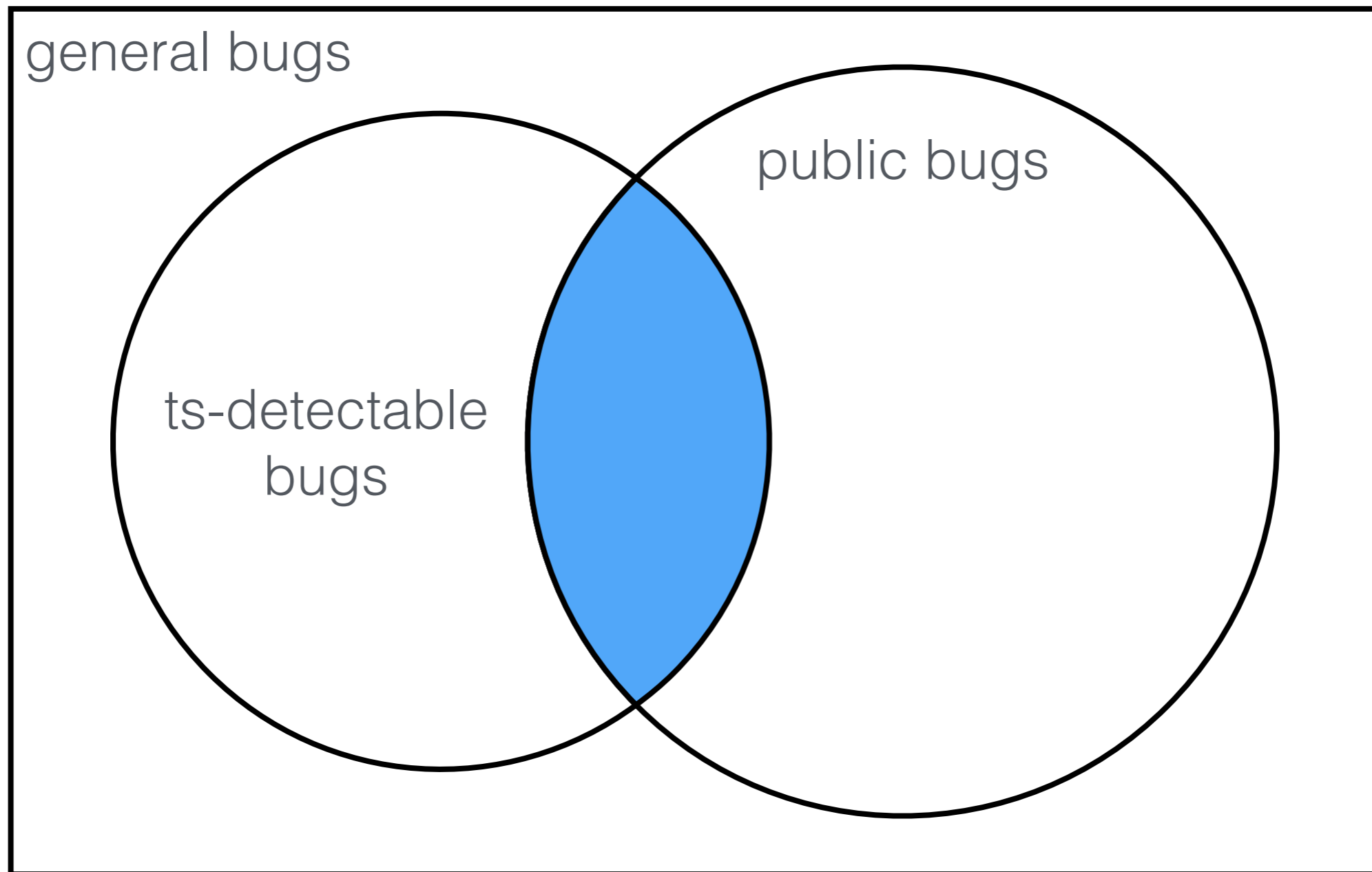
Fix Identification



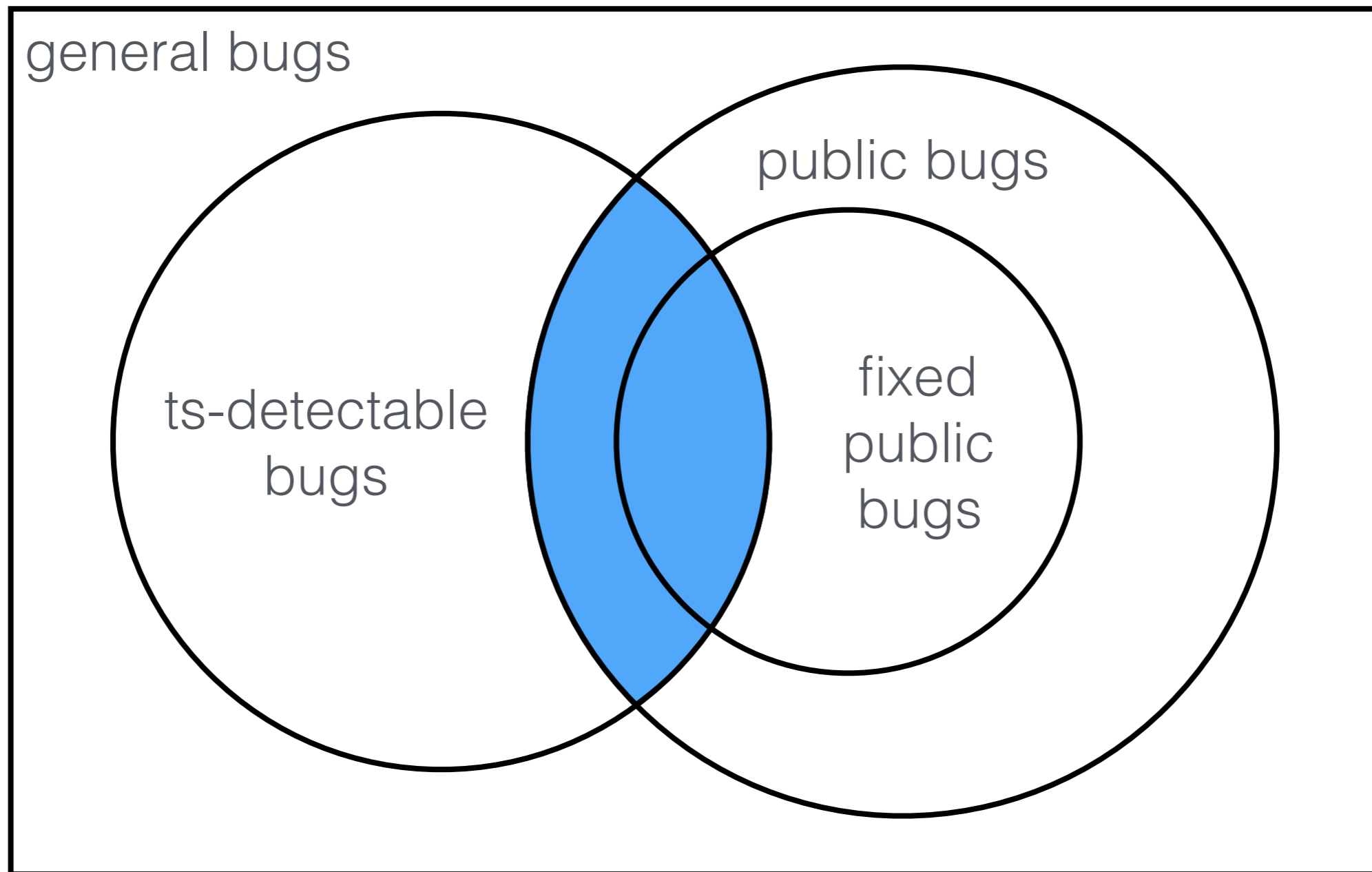
Subjects



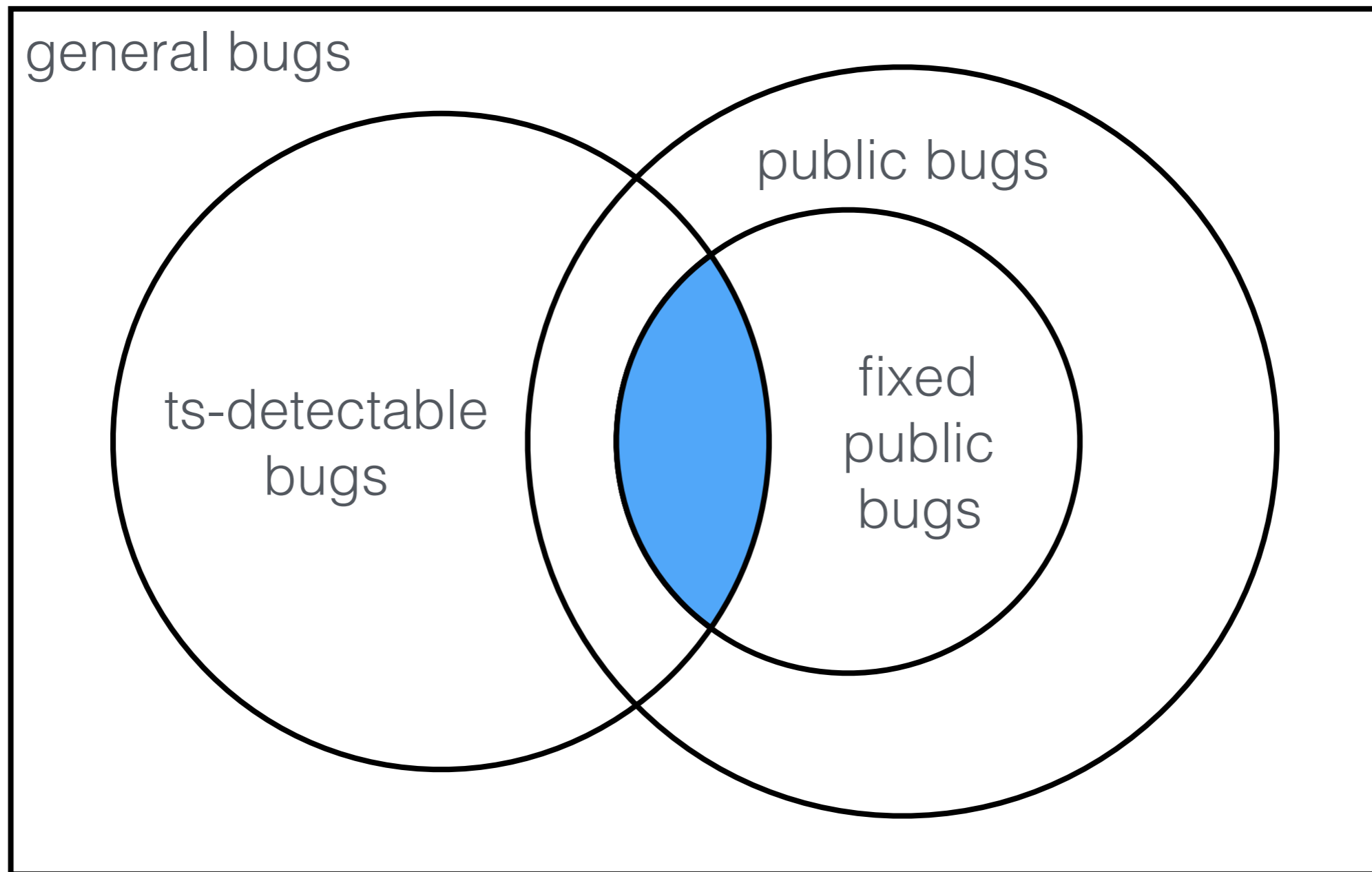
Subjects



Subjects



Subjects

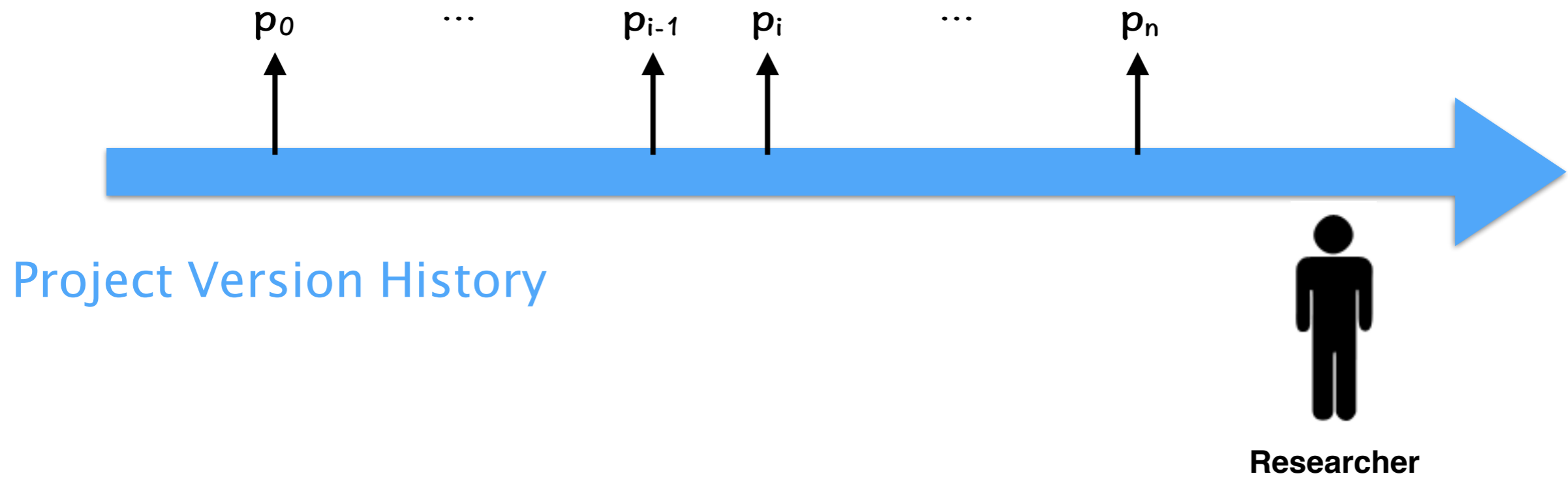


Size Statistics of the Corpus

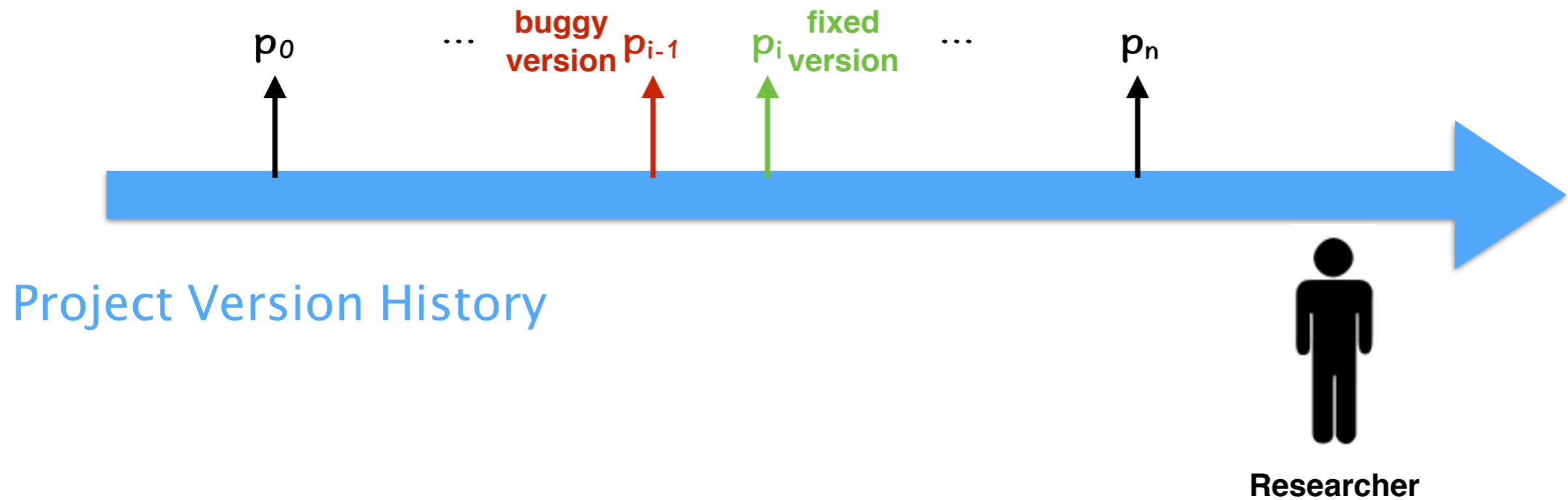
	Max	Min	Mean	Median
Project	1144440	32	18117.9	1736
Fix	270	1	16.2	6

The sizes are in lines of code.

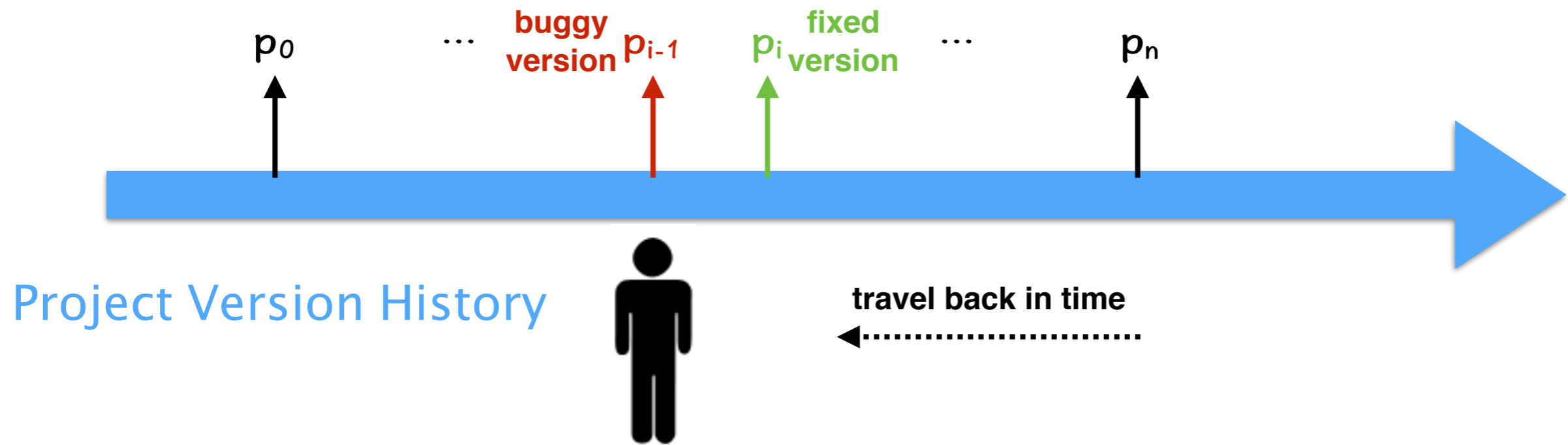
Methodology



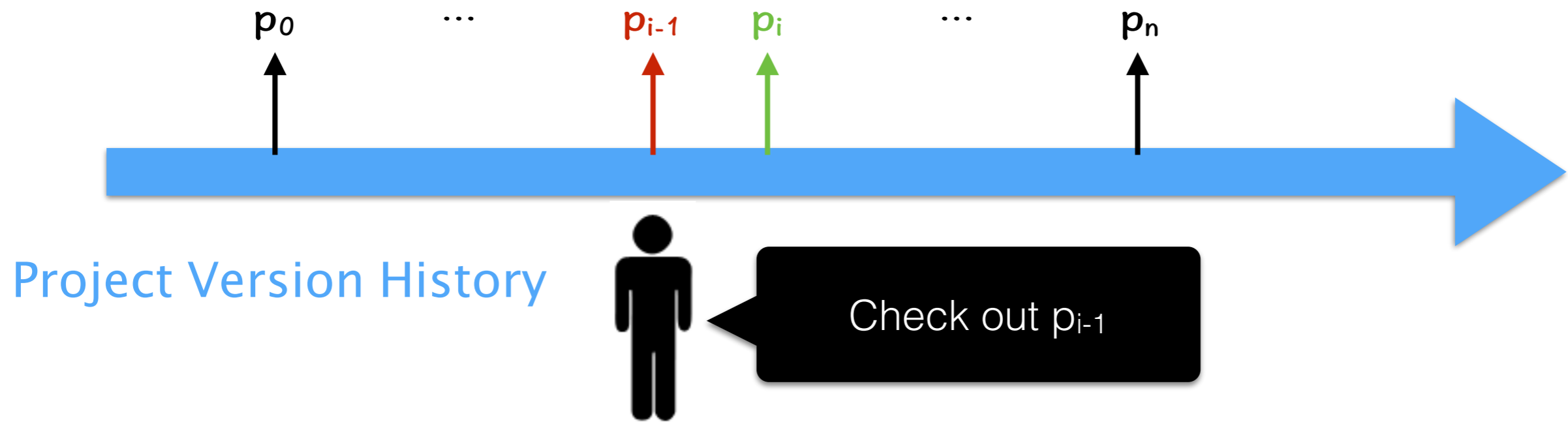
Methodology



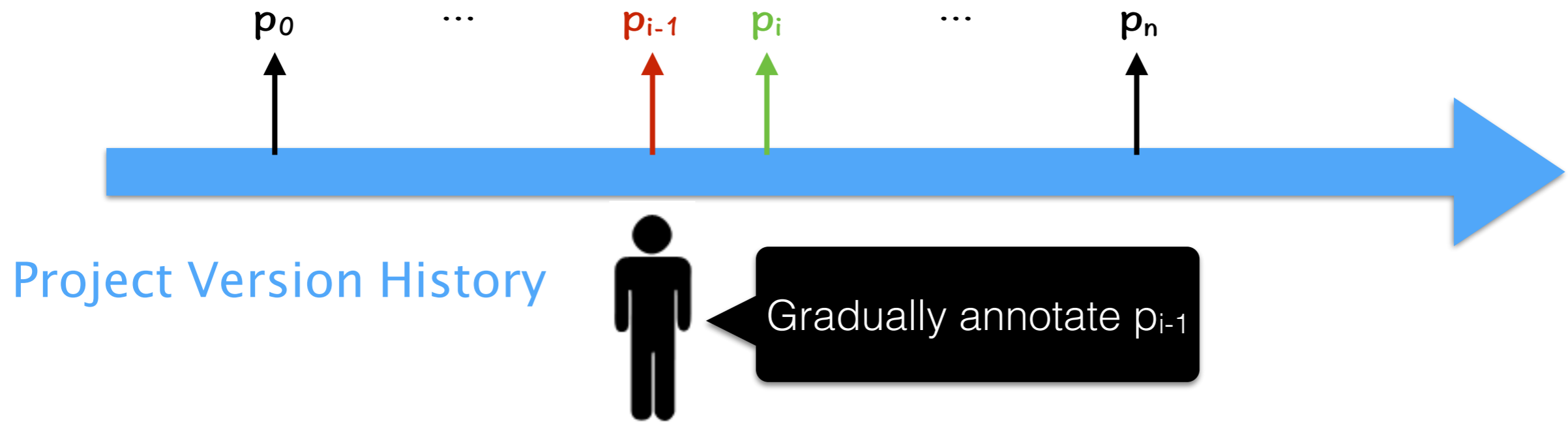
Methodology



Methodology

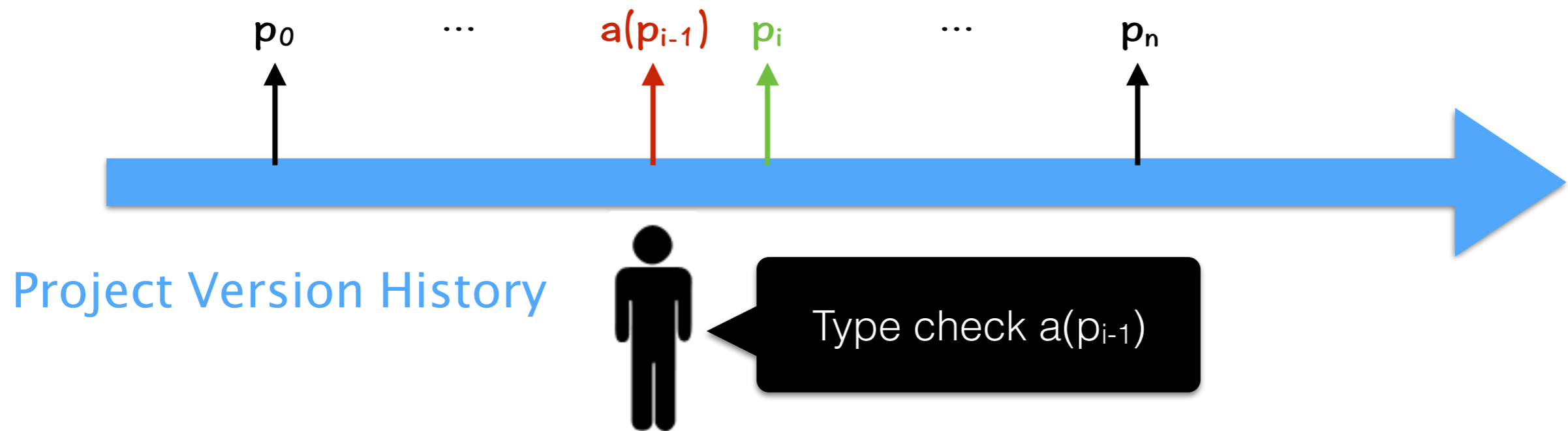


Methodology

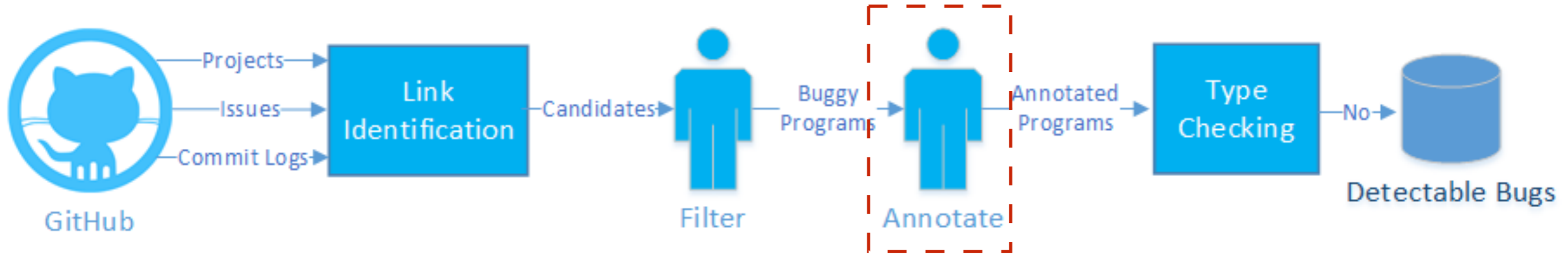


Methodology

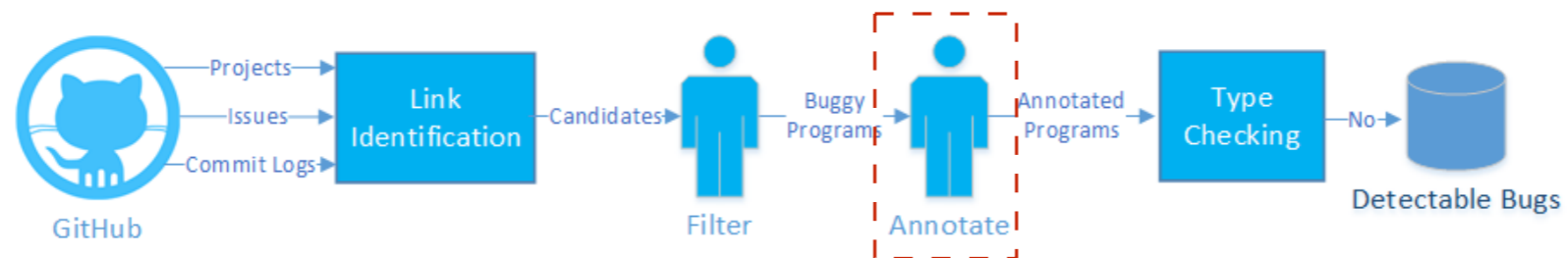
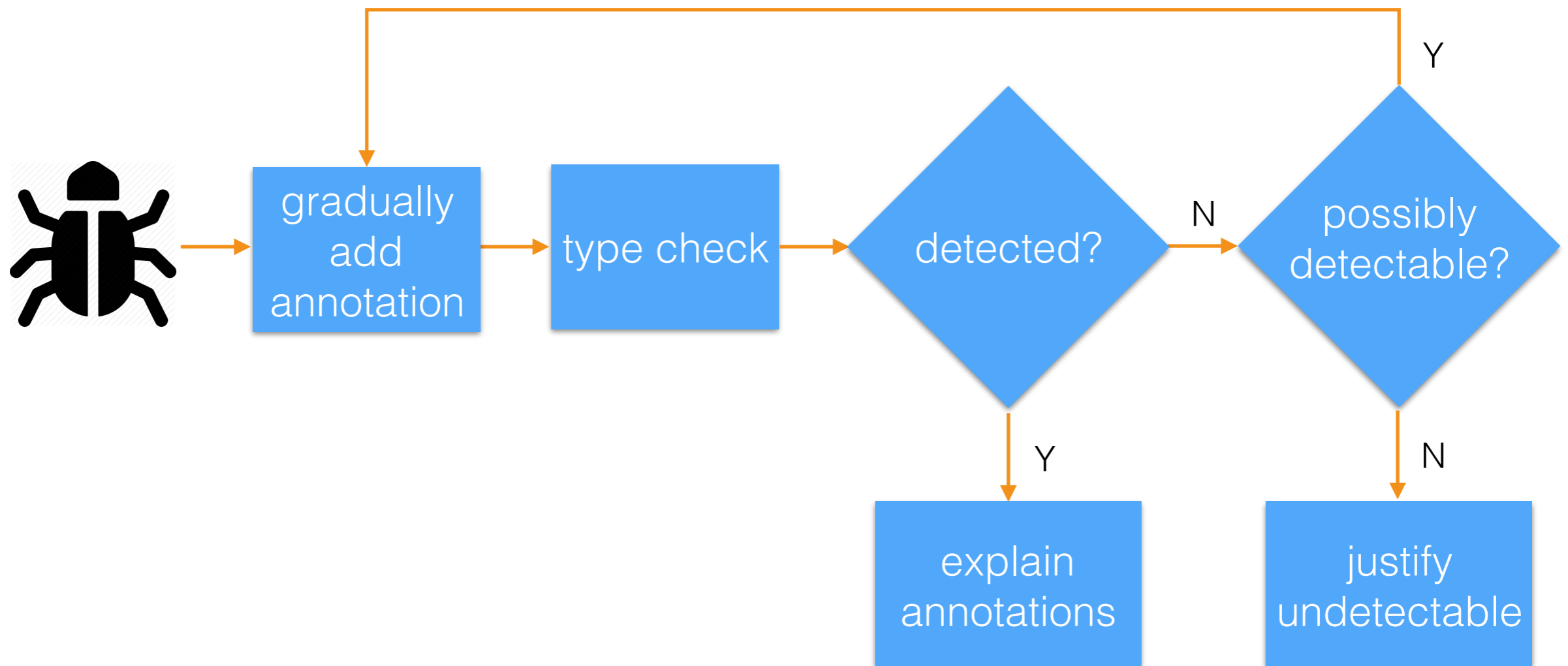
a is the annotation function



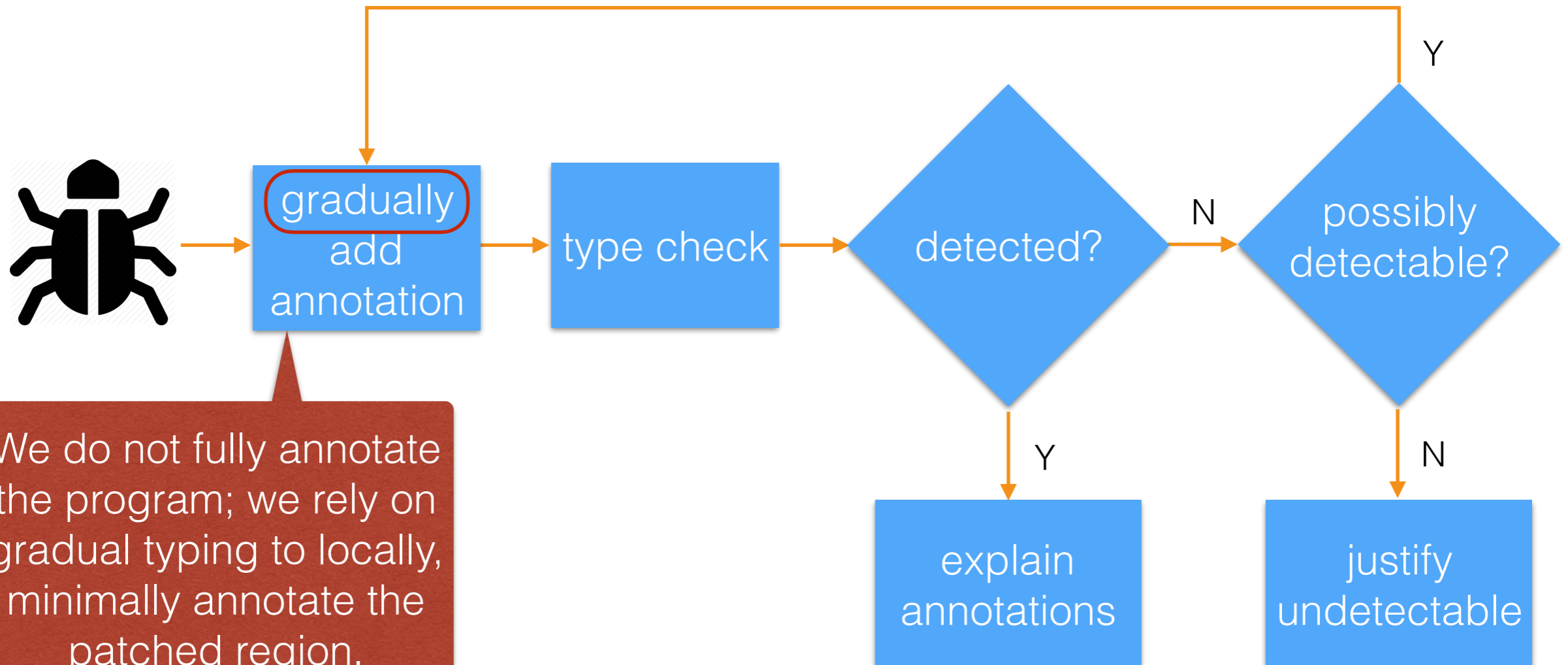
Annotation



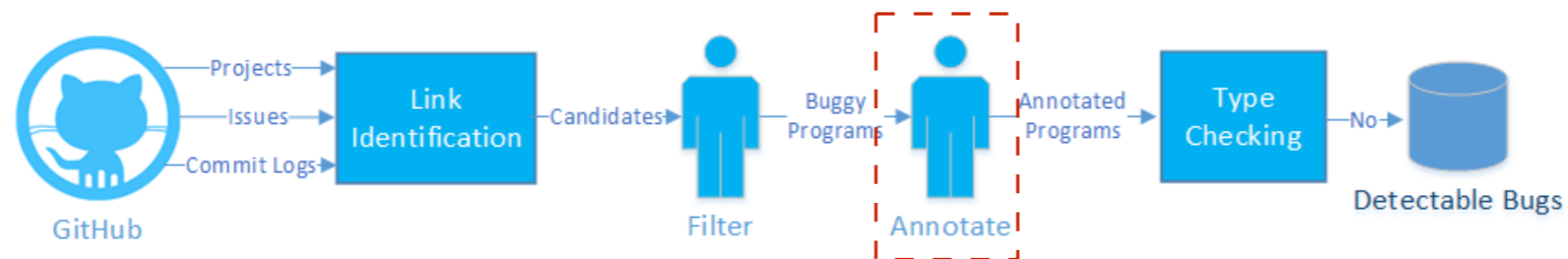
Annotation



Annotation



We do not fully annotate the program; we rely on gradual typing to locally, minimally annotate the patched region.



Annotation Sources



bug fixes



bug reports



project
documentation

Expert Source


TypeScript



Problem

```
var t = {x:0, z:1};  
t.x = t.y; // the error is y does not exist on t  
t.x = t.z;
```

Problem

What is the type of variable `t`?

```
var t = {x:0, z:1};  
t.x = t.y; // the error is y does not exist on t  
t.x = t.z;
```

Problem

What is the type of variable `t`?

Seems to be `{x: number, z: number}`?

```
var t = {x:0, z:1};  
t.x = t.y; // the error is y does not exist on t  
t.x = t.z;
```

Problem

What is the type of variable `t`?

Seems to be
`{x: number, z: number}`?

```
var t = {x:0, z:1};  
t.x = t.y; // the error is y does not exist on t  
t.x = t.z;
```

Not necessarily!

Problem

What is the type of variable `t`?

Seems to be `{x: number, z: number}`?

```
var t = {x:0, z:1};  
t.x = t.y; // the error is y does not exist on t  
t.x = t.z;  
  
...  
...  
...  
t.x = "a";
```

Not necessarily!

Problem

What is the type of variable `t`?

Seems to be
{`x`: number, `z`: number}?

```
var t = {x:0, z:1};  
t.x = t.y; // the error is y does not exist on t  
t.x = t.z;  
  
...  
...  
...  
t.x = "a";
```

Not necessarily!

Now becomes
{`x`: number | string, `z`: number}.

Type Shims

A set of type bindings for the free identifiers that

1). is *consistent* with but

2). may not exist in

a fixed version of the program containing the bug.

Shim Example

```
var t = {x:0, z:1};  
t.x = t.y; // y does not exist on t  
t.x = t.z;
```



annotate

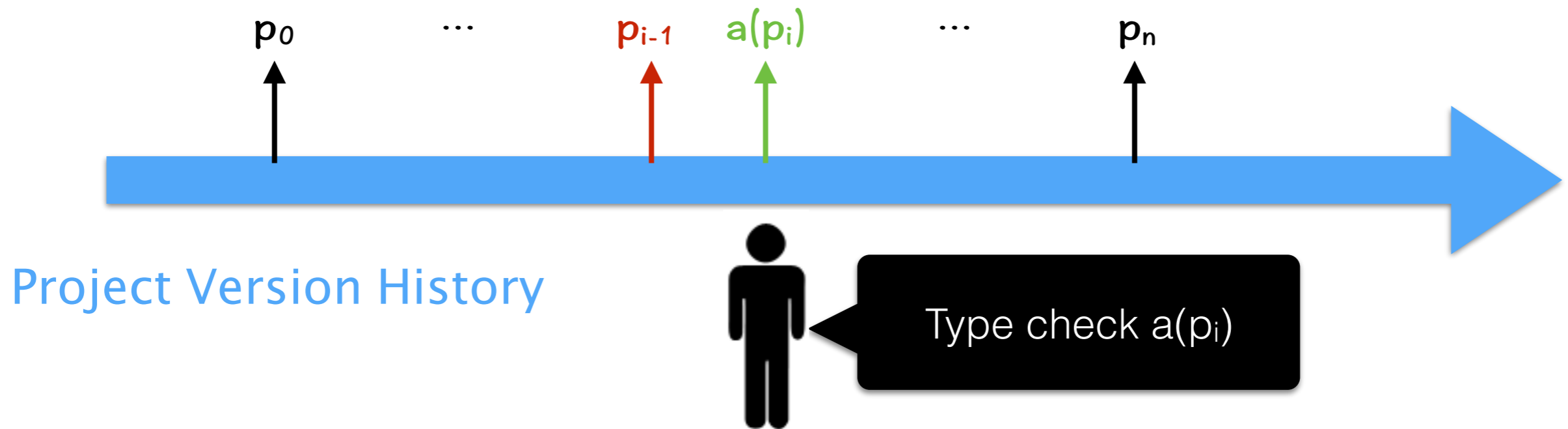
This shim is consistent, as T must be the supertype.

```
interface T {  
  x:any;  
  z:any;  
}
```

```
var t:T = {x:0, z:1};  
t.x = t.y;  
t.x = t.z;
```

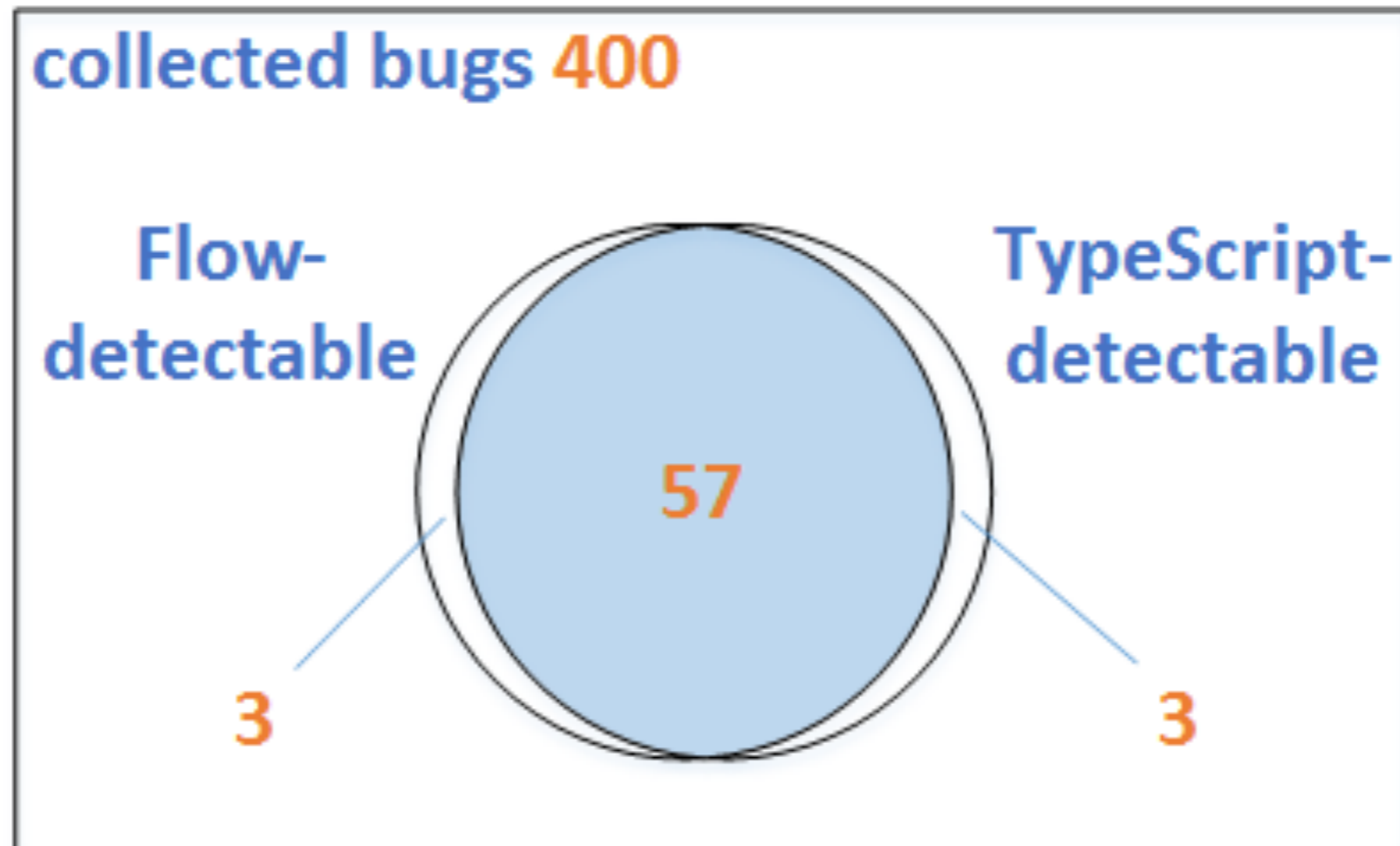

Annotation Quality

we add the same annotations to p_i



84% of the annotated fixed versions type check.

Results



Both Flow and TypeScript detect **15%** of the collected bugs; the confidence range is **[11.5%,18.5%]**, at a 95% confidence level.

Implications

“That’s shocking. If you could make a change to the way we do development that would reduce the number of bugs being checked in by 10% or more overnight, that’s a no-brainer. Unless it doubles development time or something, we’d do it.”

- An engineering manager at Microsoft

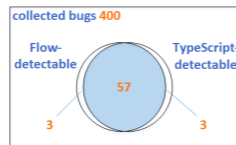
Experimental Artefacts

To Type or Not to Type: Quantifying Detectable Bugs in JavaScript

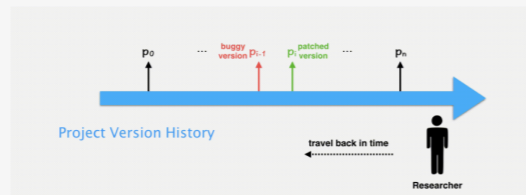
OVERVIEW

JavaScript is growing explosively and is now used in large mature projects even outside the web domain. JavaScript is also a dynamically typed language for which static type systems, notably Facebook's Flow and Microsoft's TypeScript, have been written. What benefits do these static type systems provide?

Leveraging JavaScript project histories, we select a fixed bug and check out the code just prior to the fix. We manually add type annotations to the buggy code and test whether Flow and TypeScript report an error on the buggy code, thereby possibly prompting a developer to fix the bug before its public release. We then report the proportion of bugs on which these type systems reported an error. Evaluating static type systems against public bugs, which have survived testing and review, is conservative: it understates their effectiveness at detecting bugs during private development, not to mention their other benefits such as facilitating code search/completion and serving as documentation. Despite this uneven playing field, our central finding is that both static type systems find an important percentage of public bugs: both Flow 0.30 and TypeScript 2.0 successfully detect 15%!

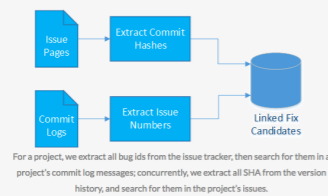


METHODOLOGY



The fact that long-running JavaScript projects have extensive version histories, coupled with the existence of static type systems that support gradual typing and can be applied to JavaScript programs with few modifications, enables us to conduct a "time-travel" experiment.

Corpus Collection



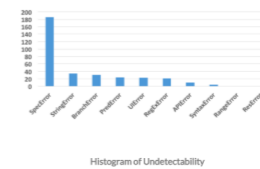
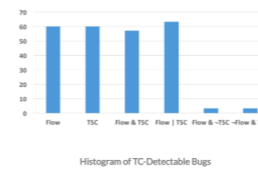
We seek to construct a corpus of bugs that is representative and sufficiently large to support statistical inference. As always, achieving representativeness is the main difficulty, which we address by uniform sampling. We cannot sample bugs directly, but rather commits that we must classify into fixes and non-fixes. Why fixes? Because a fix is often labelled as such, its parent is almost certainly buggy and it identifies the region in the parent that a developer deemed relevant to the bug. To identify bug-fixing commits, we consider only projects that use issue trackers, then we look for bug report references in commit messages and commit ids (SHA) in bug reports. This heuristic is not only noisy; it must also contend with bias in project selection and bias introduced by missing links.

We used the standard sample size computation to determine the sample size. On 19/08/2015, there were 3,910,969 closed issues for JavaScript projects on GitHub, which we used to approximate the population. We set the confidence level and confidence interval to 95% and 5%, respectively. The calculation showed that a sample of 384 bugs was sufficient for the experiment, which we rounded to 400.

Please click [here](#) for the list of the 400 studied bugs.

RESULTS

What we have discovered



	Token Tax	Time Tax (s)		
	Mean	Median	Mean	Median
Flow	1.7	2	231.4	133
TypeScript	2.4	2	306.8	262

What developers say

"That's shocking. If you could make a change to the way we do development that would reduce the number of bugs being checked in by 10% or more overnight, that's a no-brainer. Unless it doubles development time or something, we'd do it."

An Engineering Manager, Microsoft

"A scientifically true effect can be regularly reproduced by anyone who carries out the appropriate experiment in the way prescribed."



Assessment

The complete assessment on the 400 bugs.



Annotation Facilitator

The tool used to facilitate the annotation procedure.

CASE STUDY

Some bugs are worth closer inspection. Based on three criteria we select bugs for further manual assessment: ones whose TypeScript- or Flow-detectability differs among the three "raters", ones whose TypeScript- and Flow-detectability differ, and ones that are TypeScript-detectable under version 2.0 but not under 1.8. Please click [here](#) for a detailed discussion.

Disagreement

Of the 80 uniformly-sampled bugs that we used to calculate inter-rater agreement, each rater needed to make 160 decisions in total, 80 for TypeScript-preventability and 80 for Flow-preventability. 138 of these 160 decisions were unanimously labelled. We define a strong disagreement as a disagreement in which one rater deems the bug preventable while another deems it unpreventable. Of the 22 disagreements, 12 are strong.

Studied bugs

Flow vs. TypeScript

Though sharing a similar annotation syntax, Flow and TypeScript differ in some dimensions. We compared Flow and TypeScript in terms of their ability to potentially prevent public bugs had they been used when those bugs were introduced. Flow and TypeScript both catch a nontrivial portion of public bugs. In our dataset, the bugs they can prevent largely overlap, with 6 exceptions: 3 bugs are only Flow-preventable and 3 only TypeScript-preventable.

Studied bugs

TypeScript 1.8 vs. 2.0

TypeScript 2.0 was released during this study, giving us the opportunity to measure the effectiveness of its handling of `null` and `undefined`. Prior to 2.0, all types were nullable in TypeScript. TypeScript 2.0 added the compiler option `strictNullChecks`, which makes most types nonnullable. We reviewed our corpus and found that 22 bugs, an increase of 58%, are preventable under TypeScript 2.0 but not under TypeScript 1.8.

Studied bugs

http://ttendency.cs.ucl.ac.uk/projects/type_study/index.html