

# MUSE Site Visit: Planned Agenda

## **Welcome DARPA and AFRL Visitors!**

### **MUSE Meeting in DLC 170**

1:00pm-1:10pm Welcome

1:10pm-1:45pm Overview of the **Fixr** Project, Evan Chang

1:45pm-3:00pm Demo and Discussion: Analysis and Synthesis of App Commits, Shawn Meier/Vaibhav Singh

3:00pm-3:20pm Break

3:20pm-4:00pm Demo and Discussion: Harvesting and Storing App Commits, Mazin Hakeem/Sanghee Kim

### **Additional Research Meetings in ECCS 1B11**

4:00pm-5:15pm Graduate Students



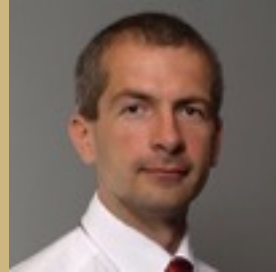
# **Fixr: Mining and Understanding Bug Fixes for App-Framework Protocol Defects**



Bor-Yuh Evan Chang



Ken Anderson



Pavol Cerny



Sriram Sankaranarayanan



Tom Yeh

**University of Colorado Boulder**



MUSE Site Visit  
February 25, 2015

# A bug that manifests spectacularly ...



# A bug that manifests spectacularly ...





# A bug that manifests spectacularly ...



A bug that manifests spectacularly ...



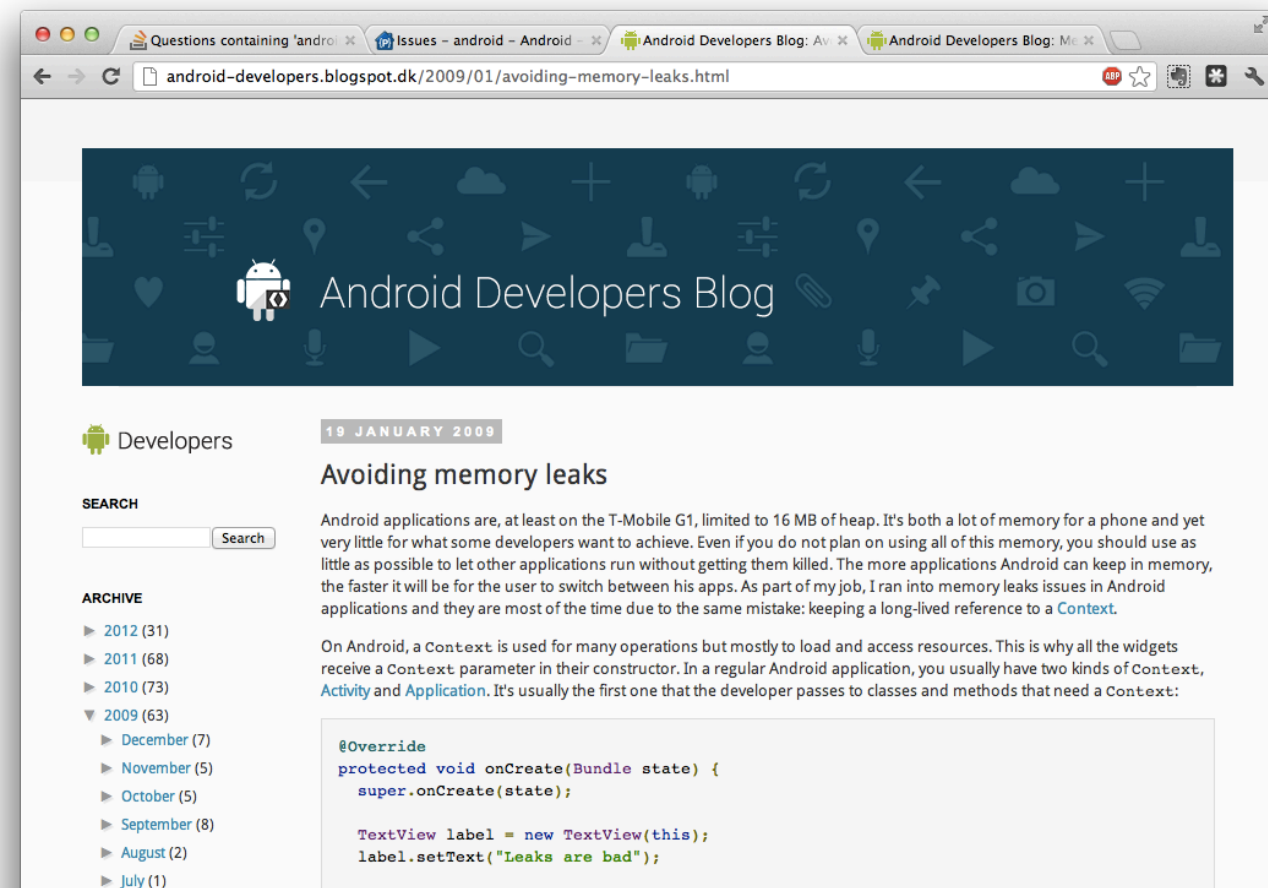
A bug that manifests spectacularly ...



**caused by an app-created memory leak**

Ask framework devs ...

# Ask framework devs ...



The screenshot shows a web browser window displaying the Android Developers Blog. The browser's address bar shows the URL `android-developers.blogspot.dk/2009/01/avoiding-memory-leaks.html`. The page features a dark blue header with the Android logo and the text "Android Developers Blog". Below the header, the article is dated "19 JANUARY 2009" and titled "Avoiding memory leaks". The article text explains that Android applications are limited to 16 MB of heap memory and discusses the importance of avoiding memory leaks by using `Context` correctly. A code snippet is provided, showing an `onCreate` method that initializes a `TextView` and sets its text to "Leaks are bad".

Developers

19 JANUARY 2009

## Avoiding memory leaks

Android applications are, at least on the T-Mobile G1, limited to 16 MB of heap. It's both a lot of memory for a phone and yet very little for what some developers want to achieve. Even if you do not plan on using all of this memory, you should use as little as possible to let other applications run without getting them killed. The more applications Android can keep in memory, the faster it will be for the user to switch between his apps. As part of my job, I ran into memory leaks issues in Android applications and they are most of the time due to the same mistake: keeping a long-lived reference to a `Context`.

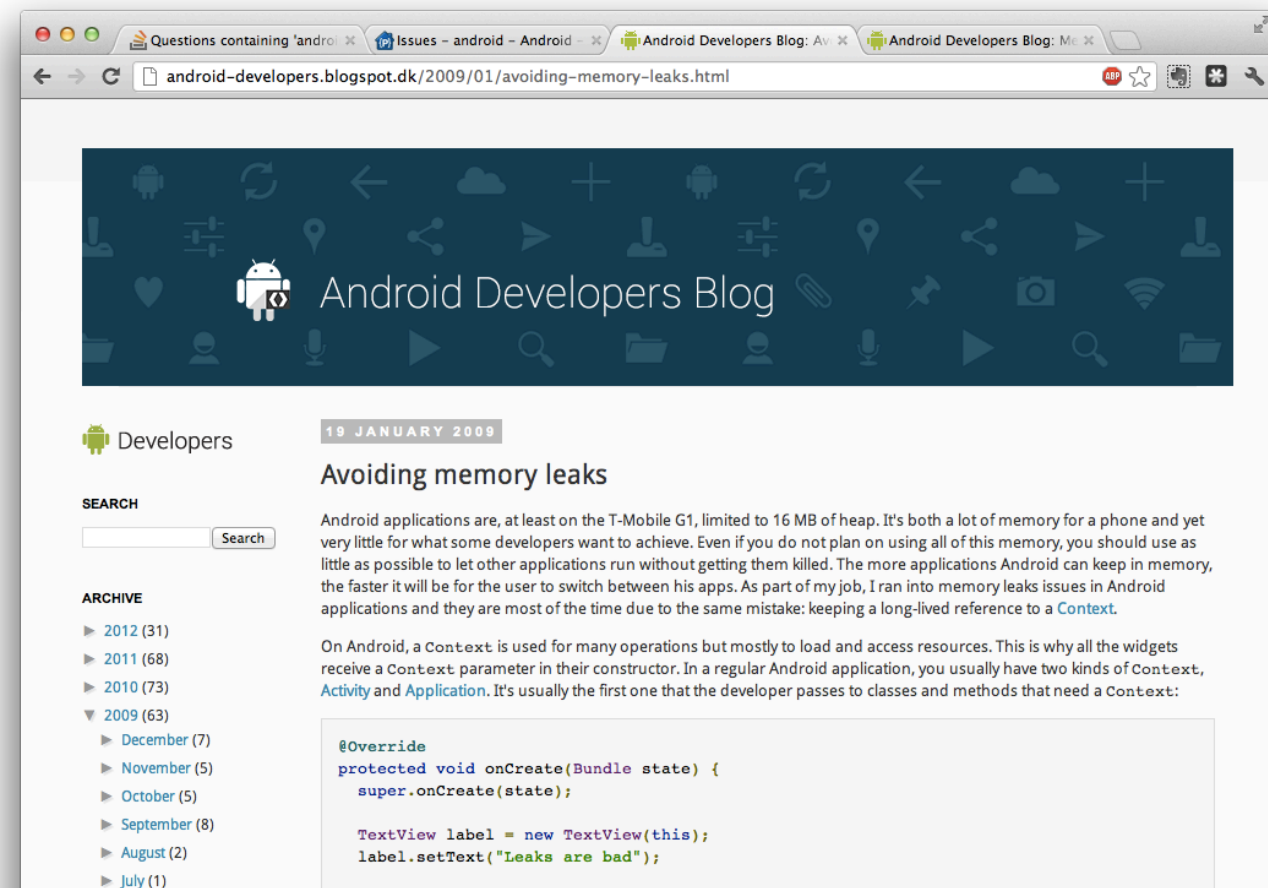
On Android, a `Context` is used for many operations but mostly to load and access resources. This is why all the widgets receive a `Context` parameter in their constructor. In a regular Android application, you usually have two kinds of `Context`, `Activity` and `Application`. It's usually the first one that the developer passes to classes and methods that need a `Context`:

```
@Override
protected void onCreate(Bundle state) {
    super.onCreate(state);

    TextView label = new TextView(this);
    label.setText("Leaks are bad");
}
```

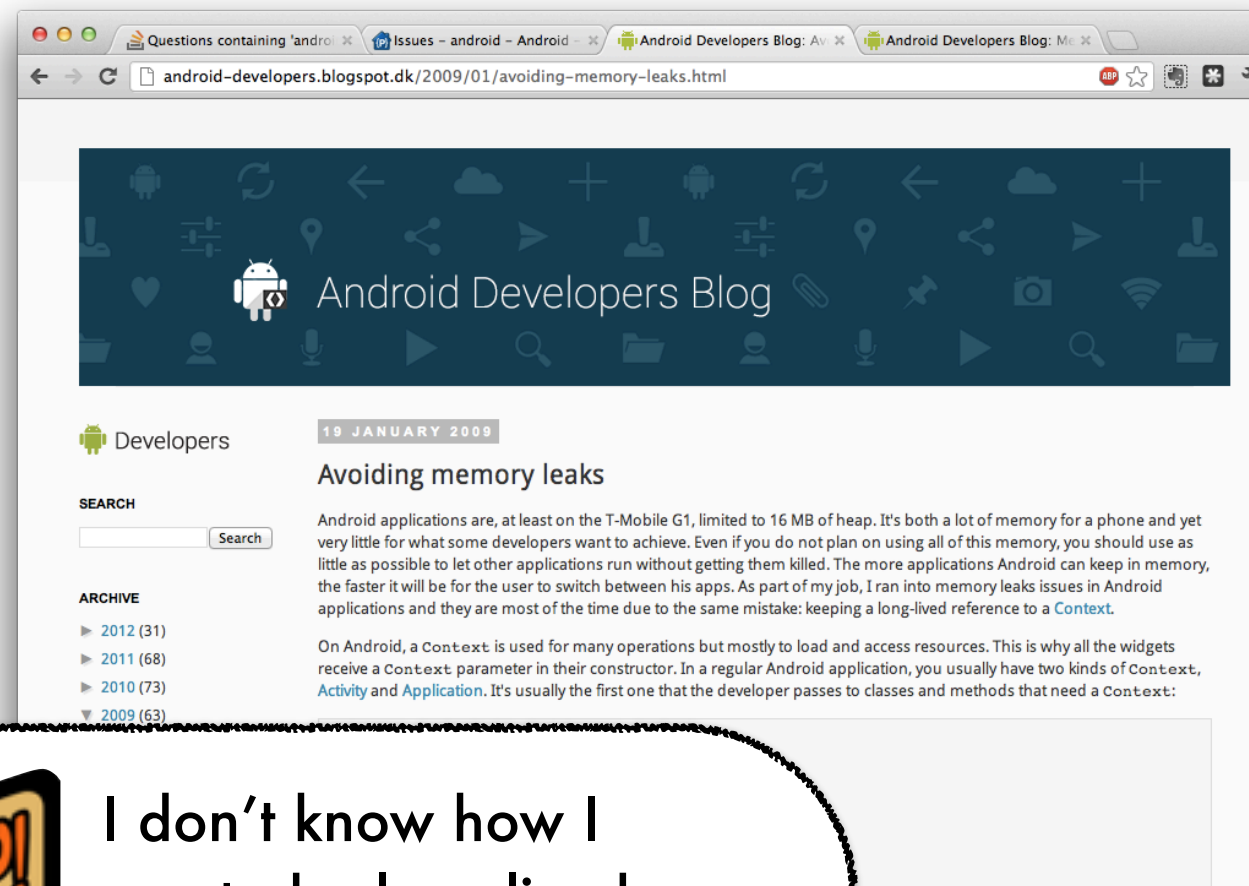
# Ask framework devs ...

“Do not keep long-lived references to a context-activity”



# Ask framework devs ...

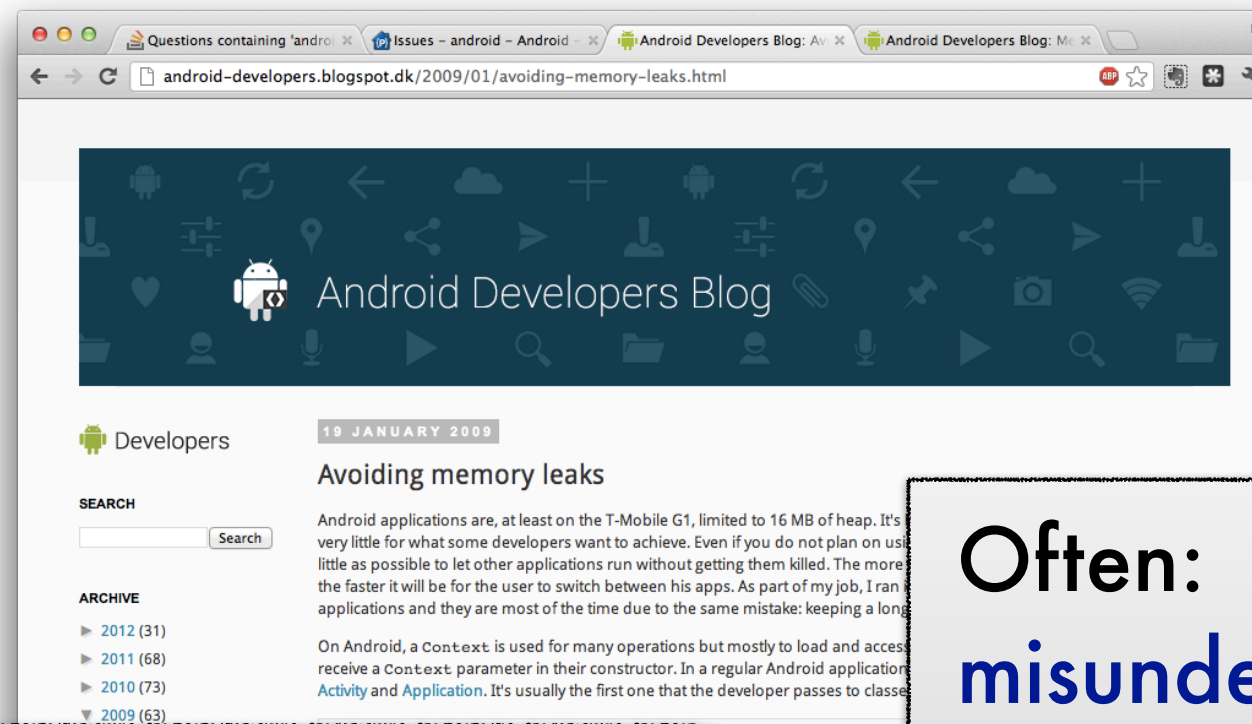
“Do not keep long-lived references to a context-activity”



I don't know how I created a long-lived reference to an Activity!

# Ask framework devs ...

“Do not keep long-lived references to a context-activity”



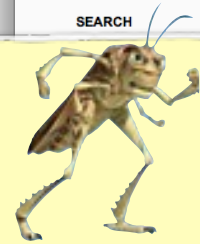
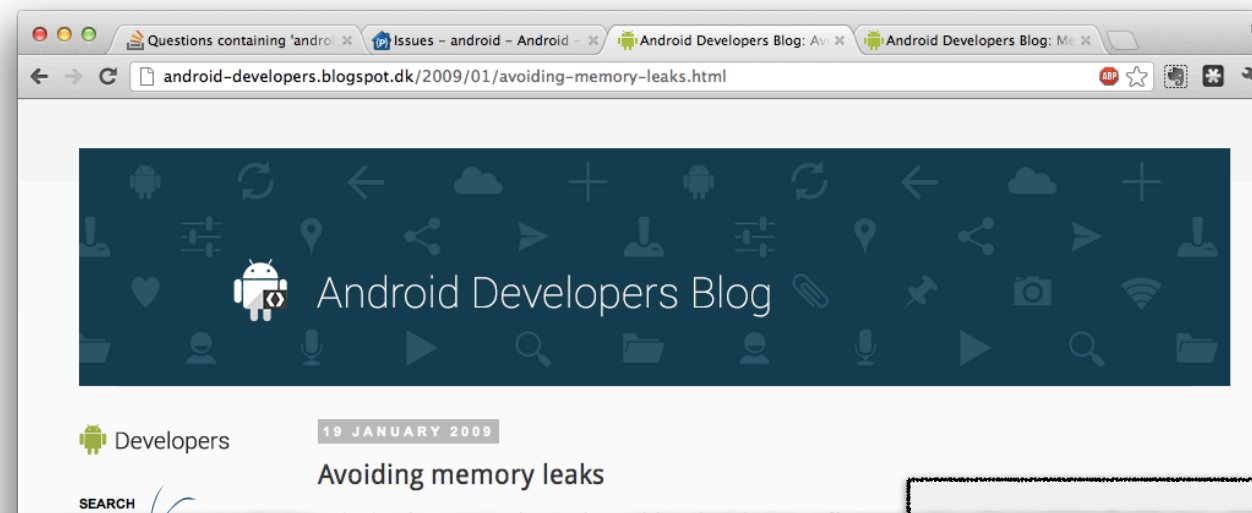
I don't know how I created a long-lived reference to an Activity!

Often: A misunderstanding of a library causes the library to keep the Activity



# Ask framework devs ...

“Do not keep long-lived references to a context-activity”



**Bug** from violating  
(implicit) framework protocol rules

# Imagining a post- MUSE scenario ...

for



I don't know how I  
created a long-lived  
reference to an Activity!

# Elsewhere, following the state of practice for debugging leaks ...



The screenshot shows a web browser window with the URL `android-developers.blogspot.dk/2011/03/memory-analysis-for-android.html`. The page features a dark blue header with the Android Developers Blog logo and a grid of icons. Below the header, the article title "Memory Analysis for Android Applications" is displayed, dated "24 MARCH 2011". The author is identified as Patrick Dubroy. The article text discusses memory management in the Dalvik runtime and mentions tools like the Allocation Tracker in DDMS. A sidebar on the left contains a search bar and an archive menu for the years 2012 and 2011.

Android Developers Blog

24 MARCH 2011

## Memory Analysis for Android Applications

*[This post is by Patrick Dubroy, an Android engineer who writes about programming, usability, and interaction on his [personal blog](#). — Tim Bray]*

The Dalvik runtime may be garbage-collected, but that doesn't mean you can ignore memory management. You should be especially mindful of memory usage on mobile devices, where memory is more constrained. In this article, we're going to take a look at some of the memory profiling tools in the Android SDK that can help you trim your application's memory usage.

Some memory usage problems are obvious. For example, if your app leaks memory every time the user touches the screen, it will probably trigger an `OutOfMemoryError` eventually and crash your app. Other problems are more subtle, and may just degrade the performance of both your app (as garbage collections are more frequent and take longer) and the entire system.

### Tools of the trade

The Android SDK provides two main ways of profiling the memory usage of an app: the *Allocation Tracker* tab in DDMS, and heap dumps. The Allocation Tracker is useful when you want to get a sense of what kinds of allocation are happening over a

# Elsewhere, following the state of practice for debugging leaks ...



## 1. Run the app

# Elsewhere, following the state of practice for debugging leaks ...



The screenshot shows the Dalvik Debug Monitor (DDMS) interface, specifically the VM Heap tab. The interface displays a table of heap memory usage statistics. The table has columns for ID, Heap Size, Allocated, Free, % Used, and # Objects. The first row shows a heap size of 8.570 MB, with 8.452 MB allocated and 127,320 KB free. The % Used is 98.62% and the number of objects is 59,281. A red circle highlights the Allocated and Free columns. Below the table, there is a "Cause GC" button and a "Display: Stats" dropdown menu.

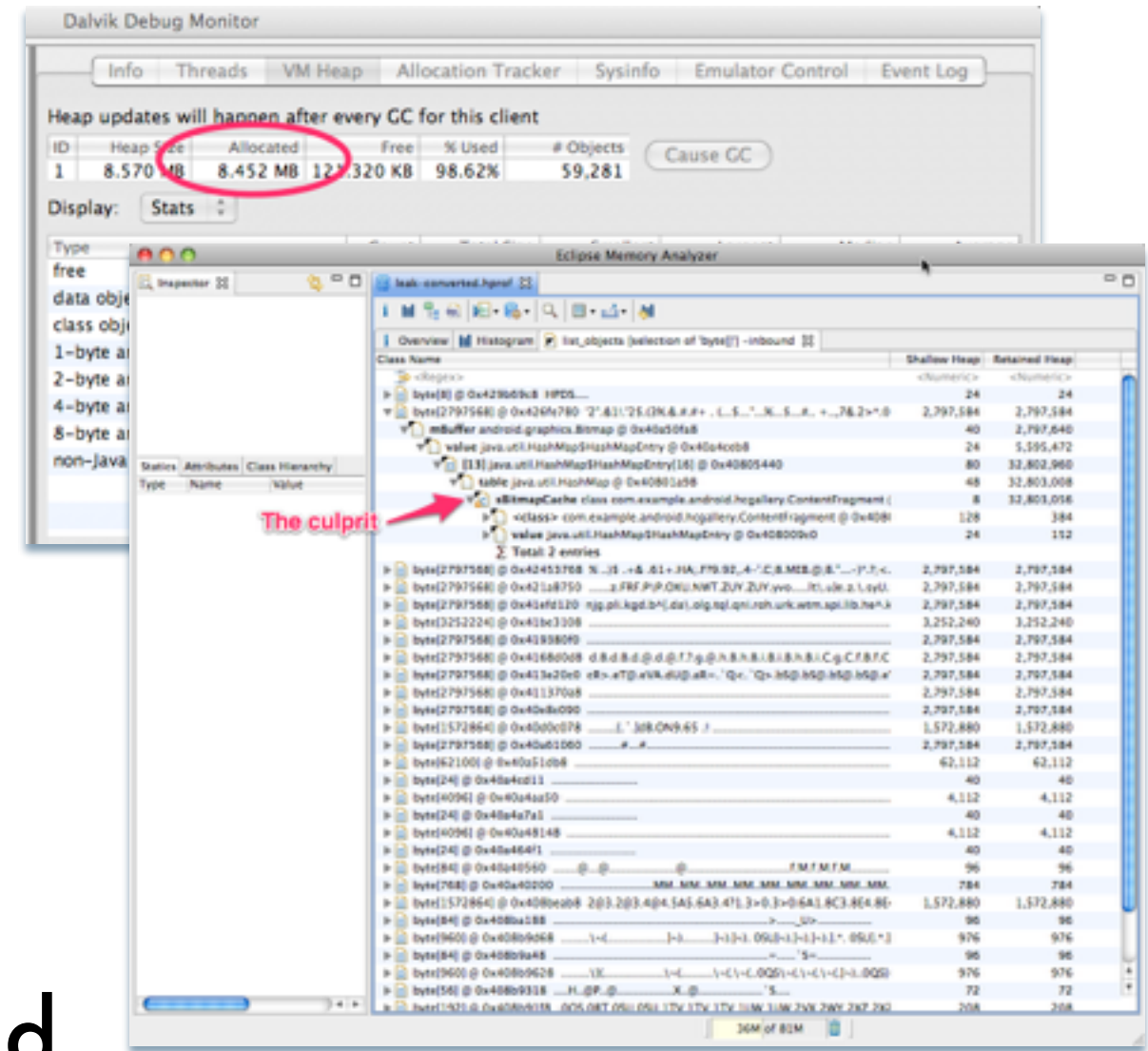
ID	Heap Size	Allocated	Free	% Used	# Objects
1	8.570 MB	8.452 MB	127,320 KB	98.62%	59,281

Type	Count	Total Size	Smallest	Largest	Median	Average
free	1,772	107.312 KB	16 B	48.297 KB	24 B	62 B
data object	40,528	1.229 MB	16 B	1.047 KB	32 B	31 B
class object	2,187	637.234 KB	168 B	34.125 KB	168 B	298 B
1-byte array (byte[], boolean[])	2,247	5.654 MB	24 B	1.500 MB	48 B	2.576 KB
2-byte array (short[], char[])	10,373	677.352 KB	24 B	28.023 KB	48 B	66 B
4-byte array (object[], int[], float[])	3,663	276.812 KB	24 B	16.023 KB	40 B	77 B
8-byte array (long[], double[])	283	14.875 KB	24 B	4.000 KB	32 B	53 B
non-java object	92	14.219 KB	16 B	8.023 KB	32 B	158 B

1. Run the app
2. Watch the heap usage

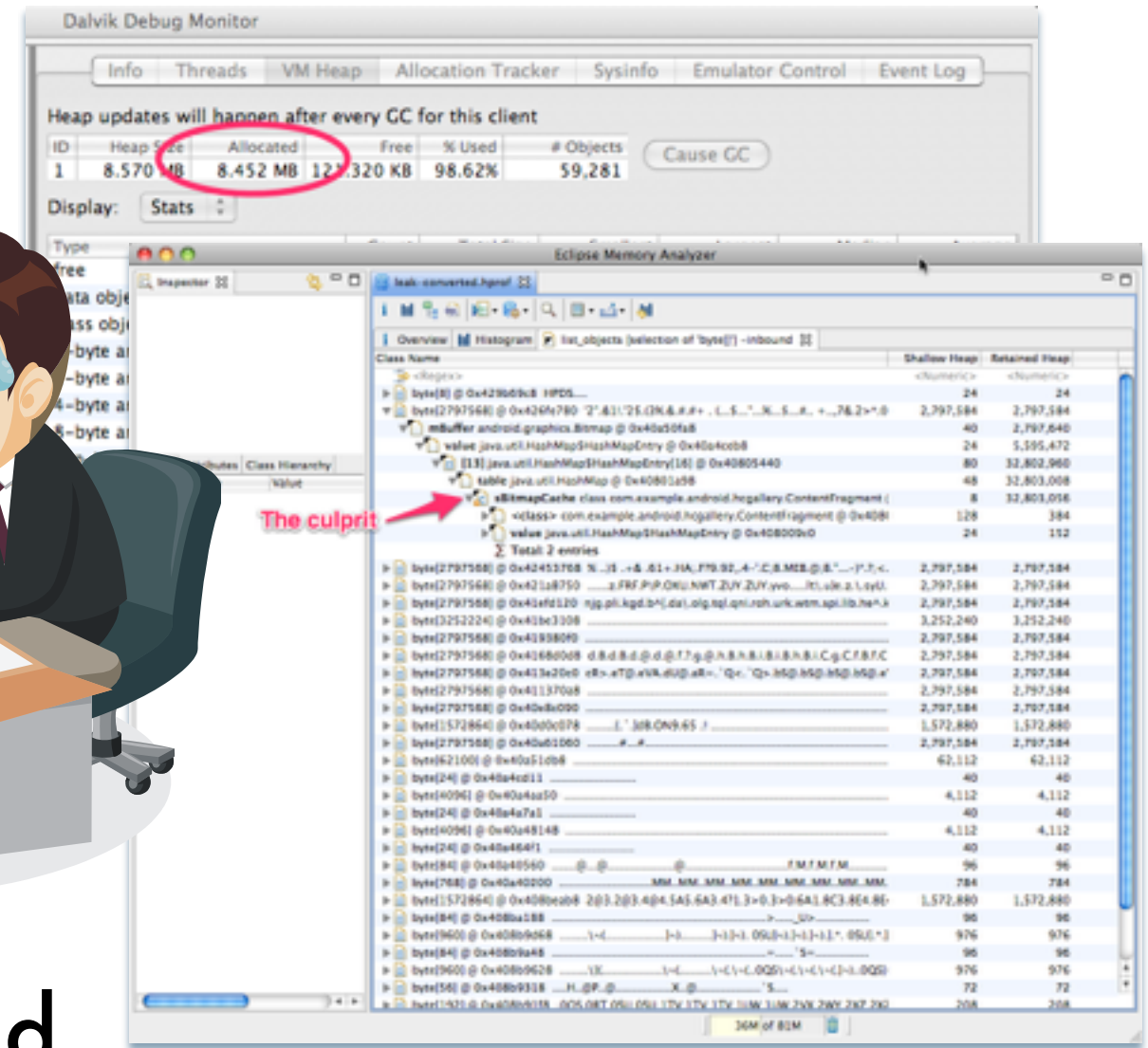
# Elsewhere, following the state of practice for debugging leaks ...



1. Run the app
2. Watch the heap usage
3. Dump the heap. Dig around and **finally** find the culprit!

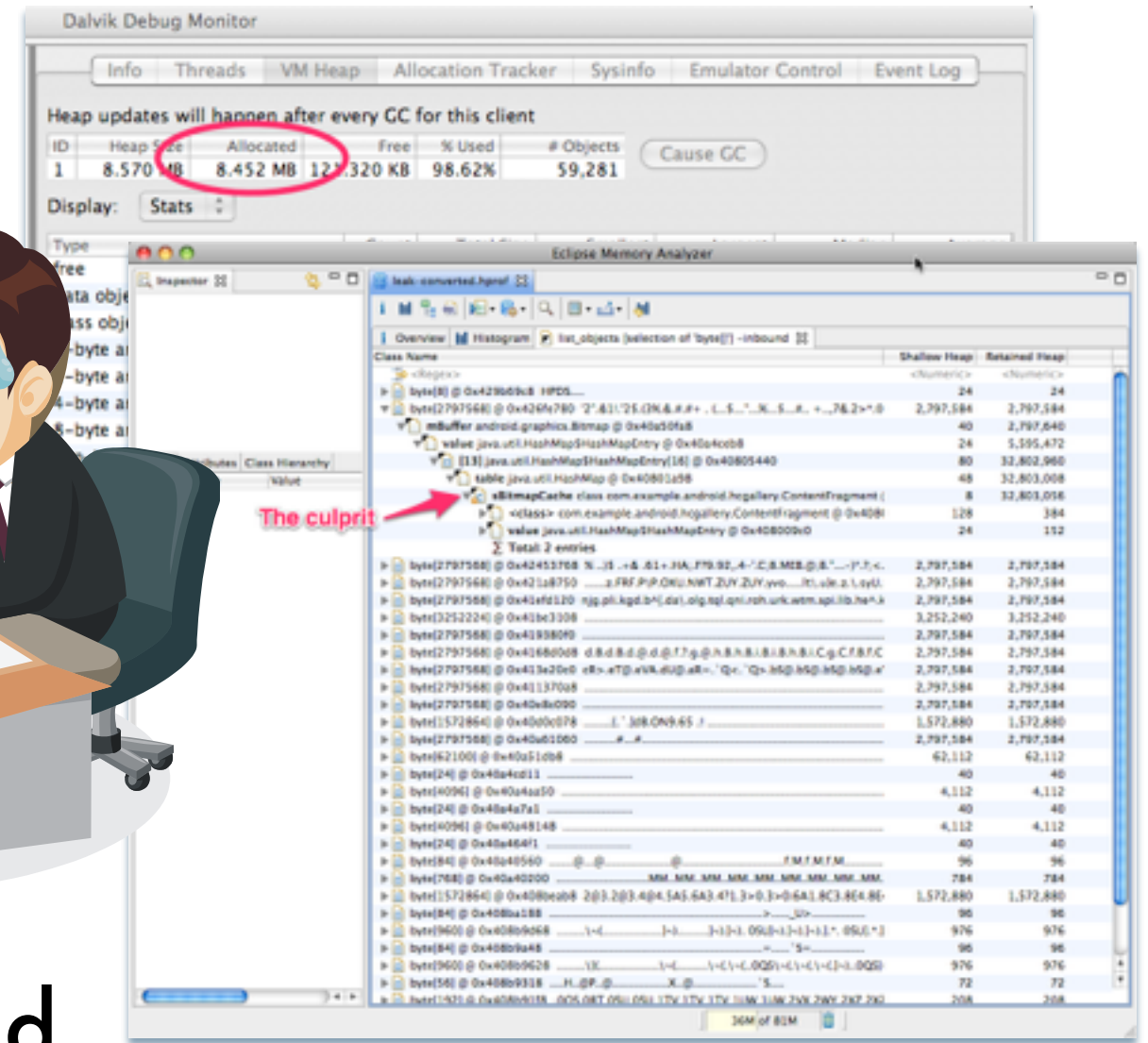


# Elsewhere, following the state of practice for debugging leaks ...



1. Run the app
2. Watch the heap usage
3. Dump the heap. Dig around and **finally** find the culprit!

# Elsewhere, following the state of practice for debugging leaks ...

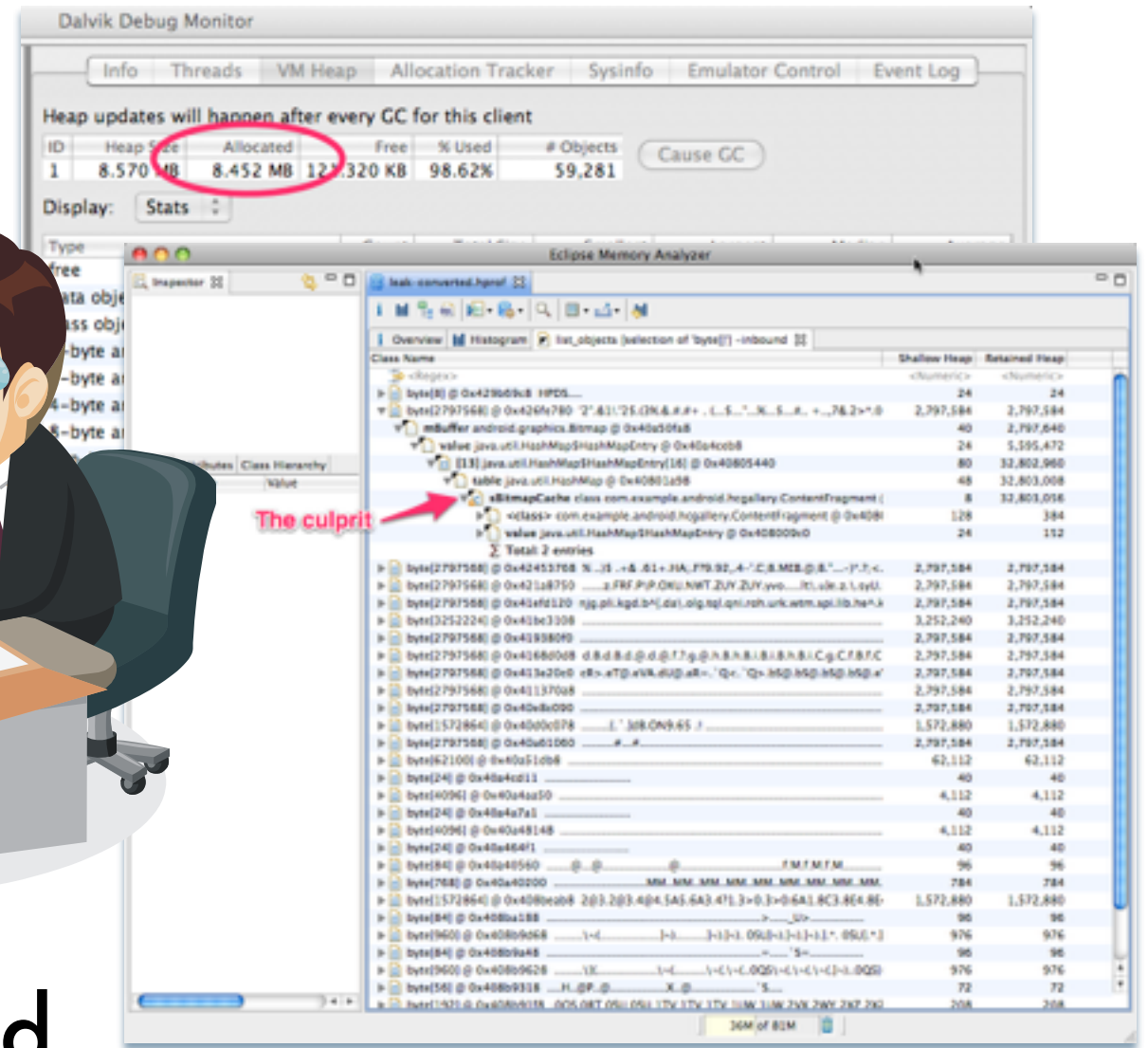


1. Run the app
2. Watch the heap usage
3. Dump the heap. Dig around and **finally** find the culprit!
4. **Commit** a **bugfix**





# Elsewhere, following the state of practice for debugging leaks ...



1. Run the app
2. Watch the heap usage
3. Dump the heap. Dig around and **finally** find the culprit!
4. **Commit** a **bugfix**
5. **Bugfix** is picked up by **Fixr**

GitHub

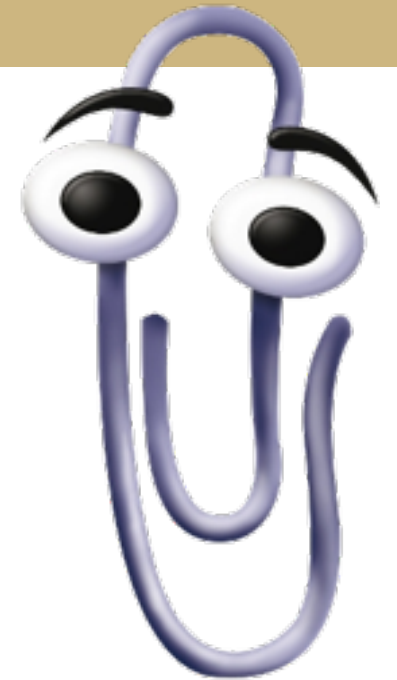


# A **Fixr**-enabled IDE responds ...





I don't know how I  
created a long-lived  
reference to an Activity!

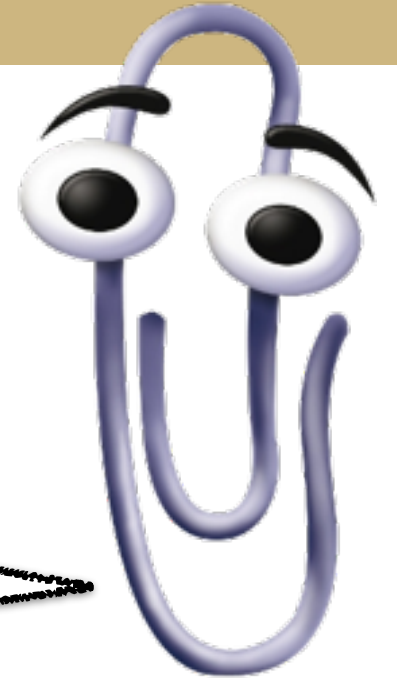
# A **Fixr**-enabled IDE responds ...



I don't know how I  
created a long-lived  
reference to an Activity!

# A Fixr-enabled IDE responds ...



It looks like you've created a memory leak like  and 100,000 others. Would you like to apply  ?



I don't know how I created a long-lived reference to an Activity!



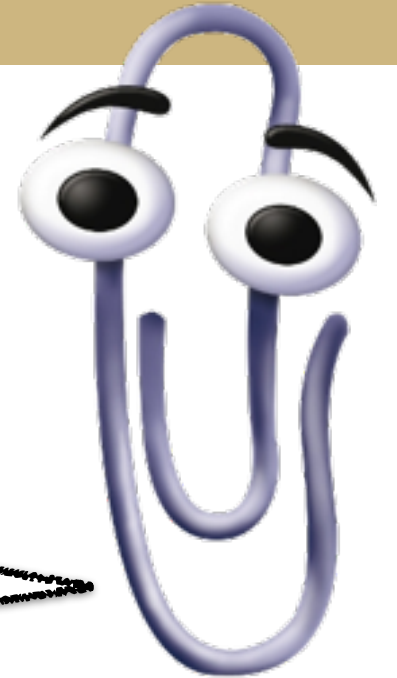
# A Fixr-enabled IDE responds ...

It looks like you've created a memory leak like  and 100,000 others. Would you like to apply  ?

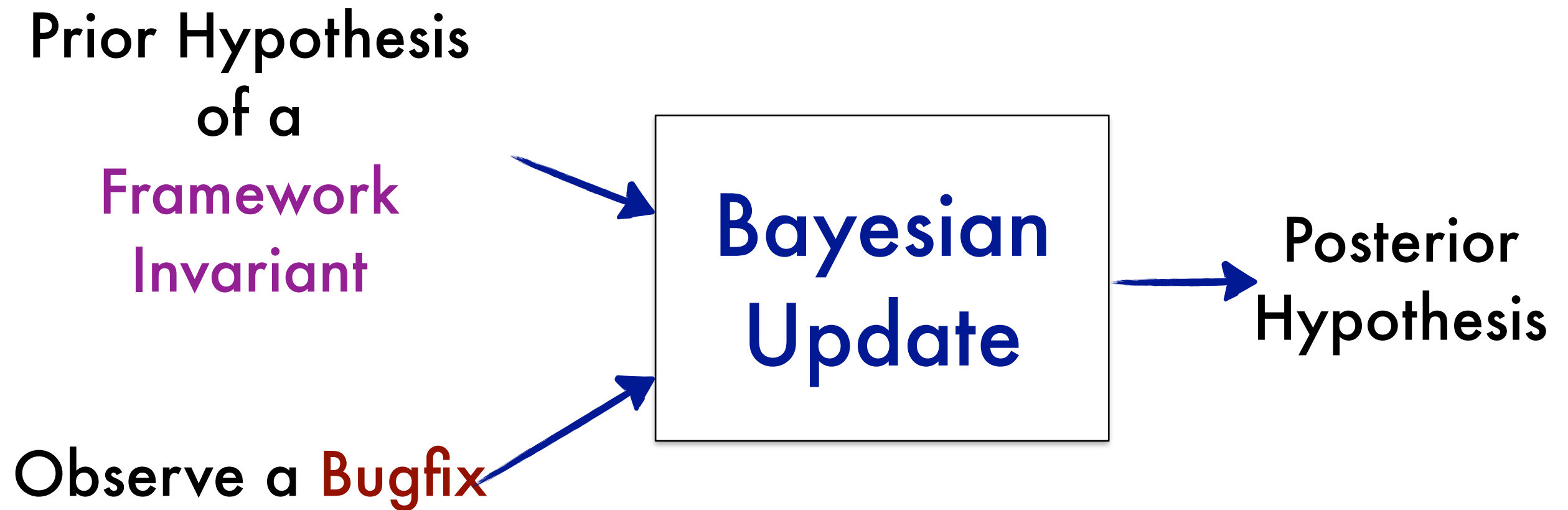
the **bugfix** is "transferred"



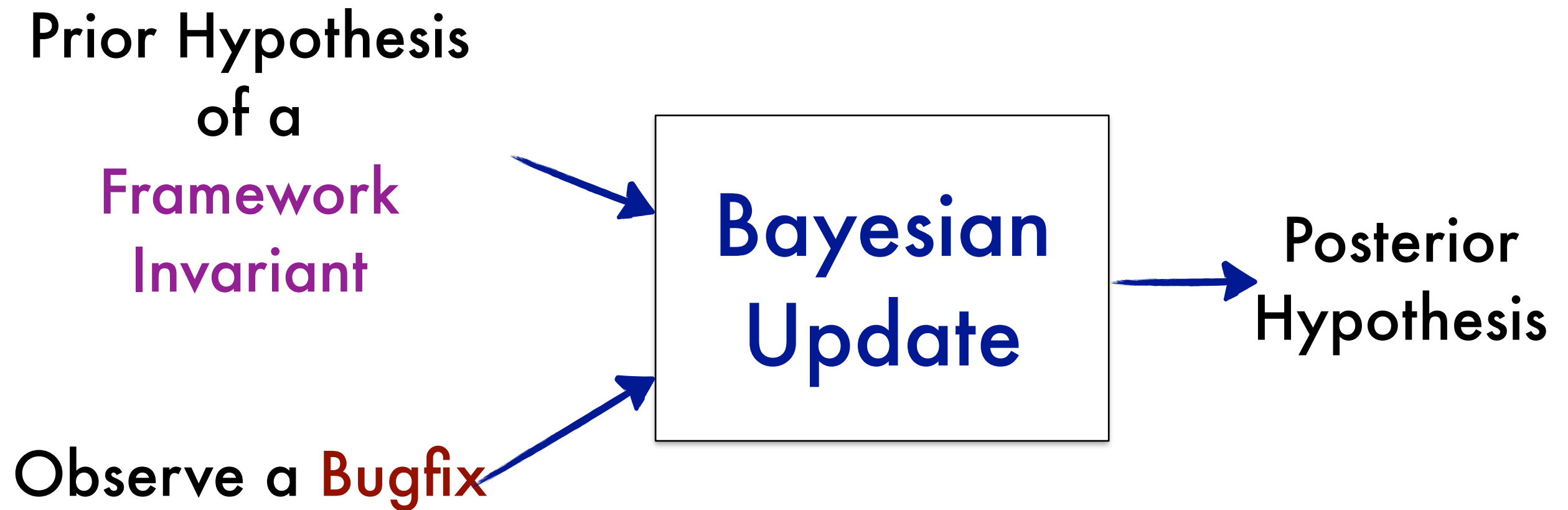
I don't know how I created a long-lived reference to an Activity!



# Summary: Mine framework specifications with bugfixes



Summary: **Mine** framework specifications with **bugfixes**



**The Fixr Loop:**  
Create as many observations as possible



**Simple motivating example:  
A well-understood Android  
bug**



# Simple motivating example: A well-understood Android bug



a common misuse of the framework

**Bug**  
(on Android <4)

**`aView.setTag(..., anObject)`**

**Bug**  
(on Android <4)

**aView.setTag(..., anObject)**



if anObject can reach aView

**Bug**  
(on Android <4)

# aView.setTag(..., anObject)



if anObject can reach aView

**Framework  
Invariant**

```
class View {  
    static WeakHashMap<View, SparseArray<Object>> sTags;  
    Object mTag;  
}
```

**Bug**  
(on Android <4)

# aView.setTag(..., anObject)

if anObject can reach aView


**Framework  
Invariant**

```
class View {  
    static WeakHashMap<View, SparseArray<Object>> sTags;  
    Object mTag;  
}
```

because of an **unspecified** class invariant: **sTags'**  
values (:**Object**) must not reach their keys (:**View**)

**Bug**  
(on Android <4)

# aView.setTag(..., anObject)



if anObject can reach aView

**Framework  
Invariant**

```
class View {  
    static WeakHashMap<View, SparseArray<Object>> sTags;  
    Object mTag;  
}
```

because of an **unspecified** class invariant: **sTags'**  
values (:**Object**) must not reach their keys (:**View**)

**A Fix**

Bug  
(on Android <4)

`aView.setTag(..., anObject)`

if anObject can reach aView

Framework  
Invariant

```
class View {  
    static WeakHashMap<View, SparseArray<Object>> sTags;  
    Object mTag;  
}
```

because of an **unspecified** class invariant: **sTags'**  
values (**:Object**) must not reach their keys (**:View**)

A Fix

~~`aView.setTag(..., anObject)`~~  
`aView.setTag(... anObject...)`

uses mTag instead

Bug  
(on Android <4)

`aView.setTag(..., anObject)`

*bug pre*

if anObject can reach aView

Framework  
Invariant

```
class View {  
    static WeakHashMap<View, SparseArray<Object>> sTags;  
    Object mTag;  
}
```

*invariant*

because of an **unspecified** class invariant: **sTags'**  
values (**:Object**) must not reach their keys (**:View**)

Goal: Produce this **repair specification**: *bug pre*,  
*framework invariant*, *fix suggestion*



# Challenges

Given a bugfix commit, how do we **summarize** and **generalize** the fix (to be able “transfer”)?

How do we find **bugfix commits**?

# Challenges

Given a bugfix commit, how do we **summarize** and **generalize** the fix (to be able “transfer”)?

a specification of the  
`View.setTag` repair

How do we find **bugfix commits**?

# Challenges

Given a bugfix commit, how do we **summarize** and **generalize** the fix (to be able “transfer”)?

a specification of the  
`View.setTag` repair

How do we find **bugfix commits**?

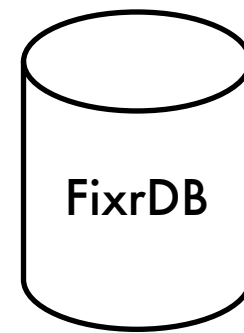
an instance of a  
`View.setTag` fix

**Fixr**

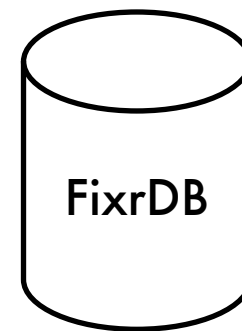
**Components and  
Workflows**

# Workflow 1a: Harvesting bugfix commits

# Workflow 1a: Harvesting bugfix commits

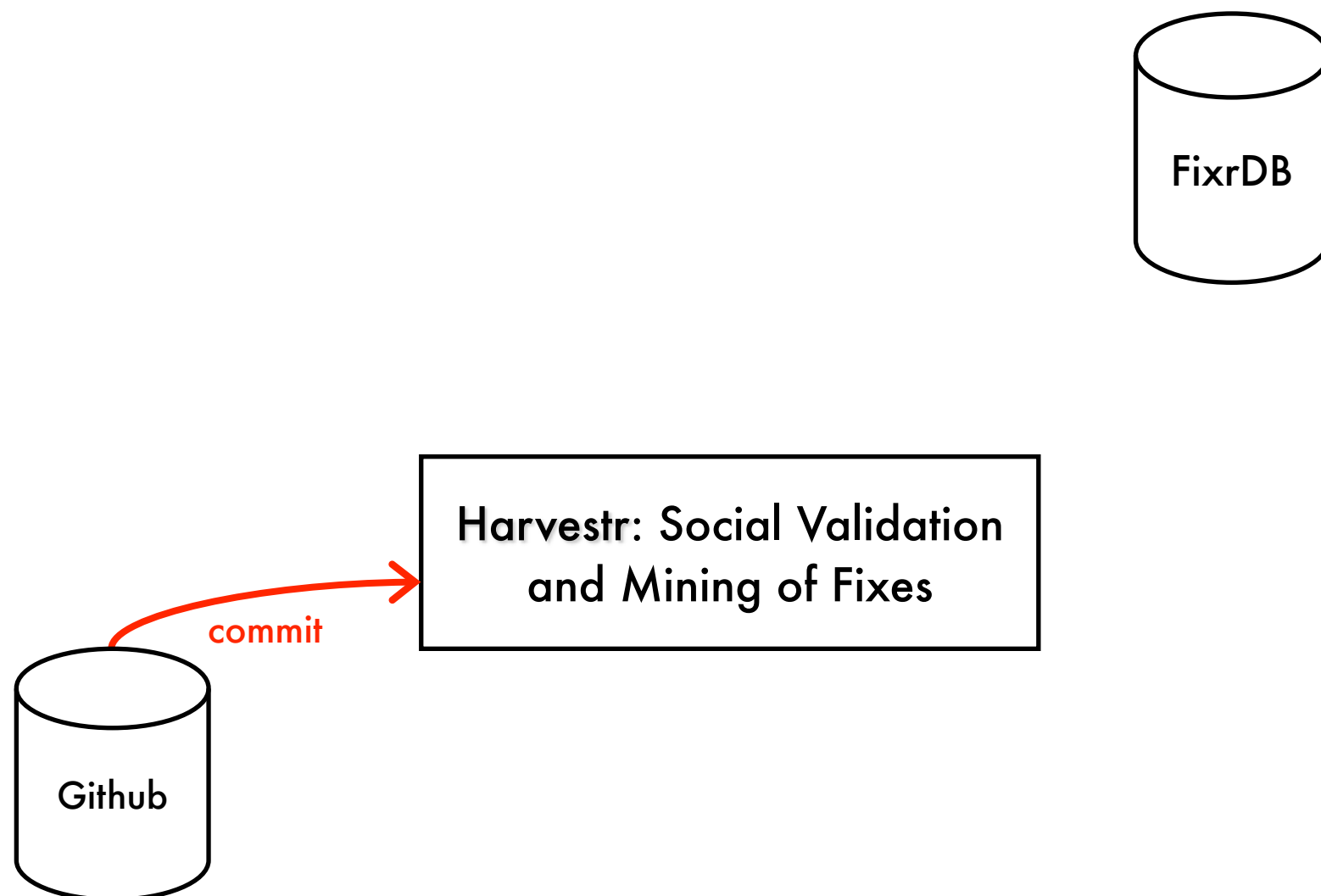


# Workflow 1a: Harvesting bugfix commits



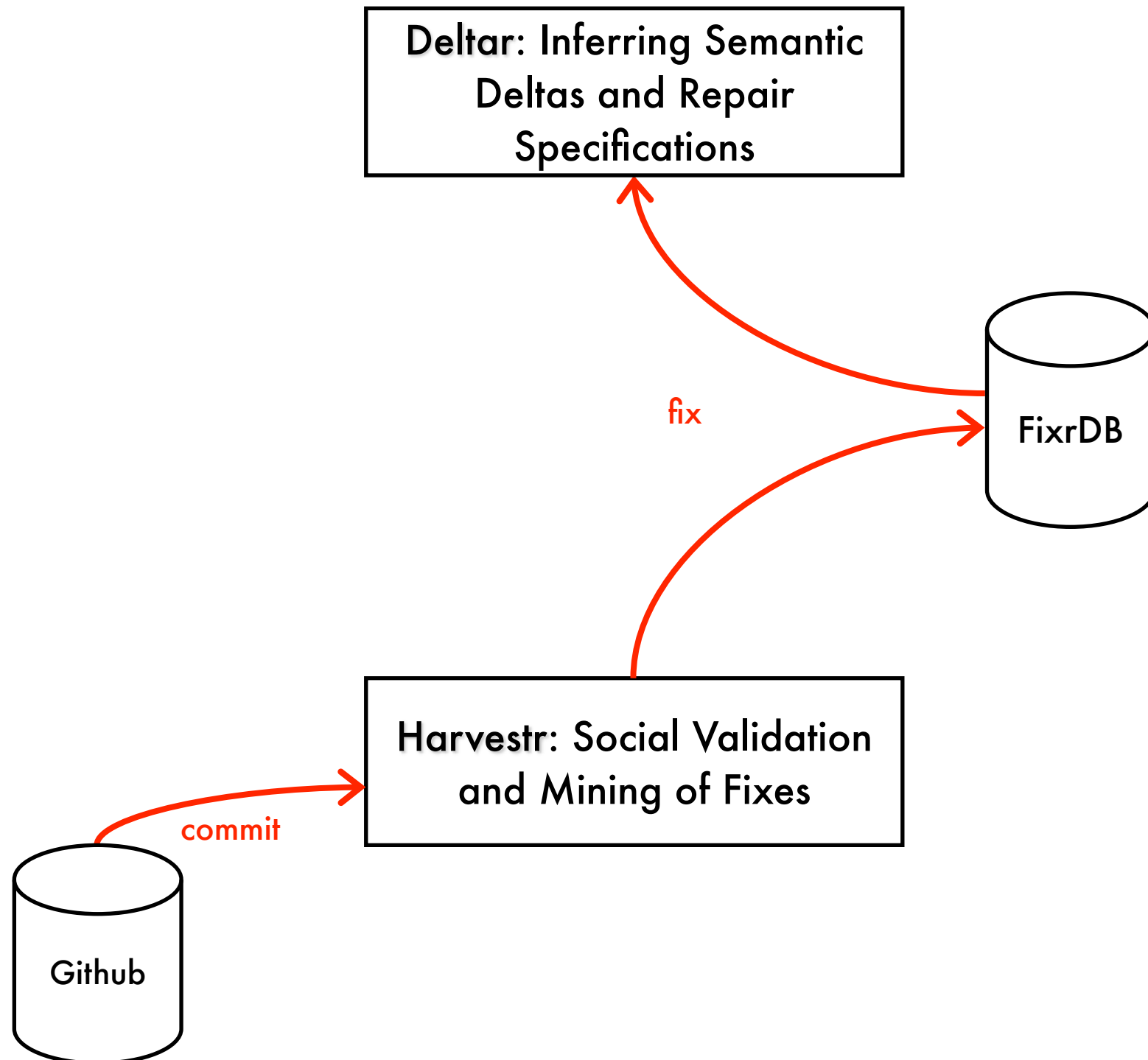
**Harvestr: Social Validation  
and Mining of Fixes**

# Workflow 1a: Harvesting bugfix commits

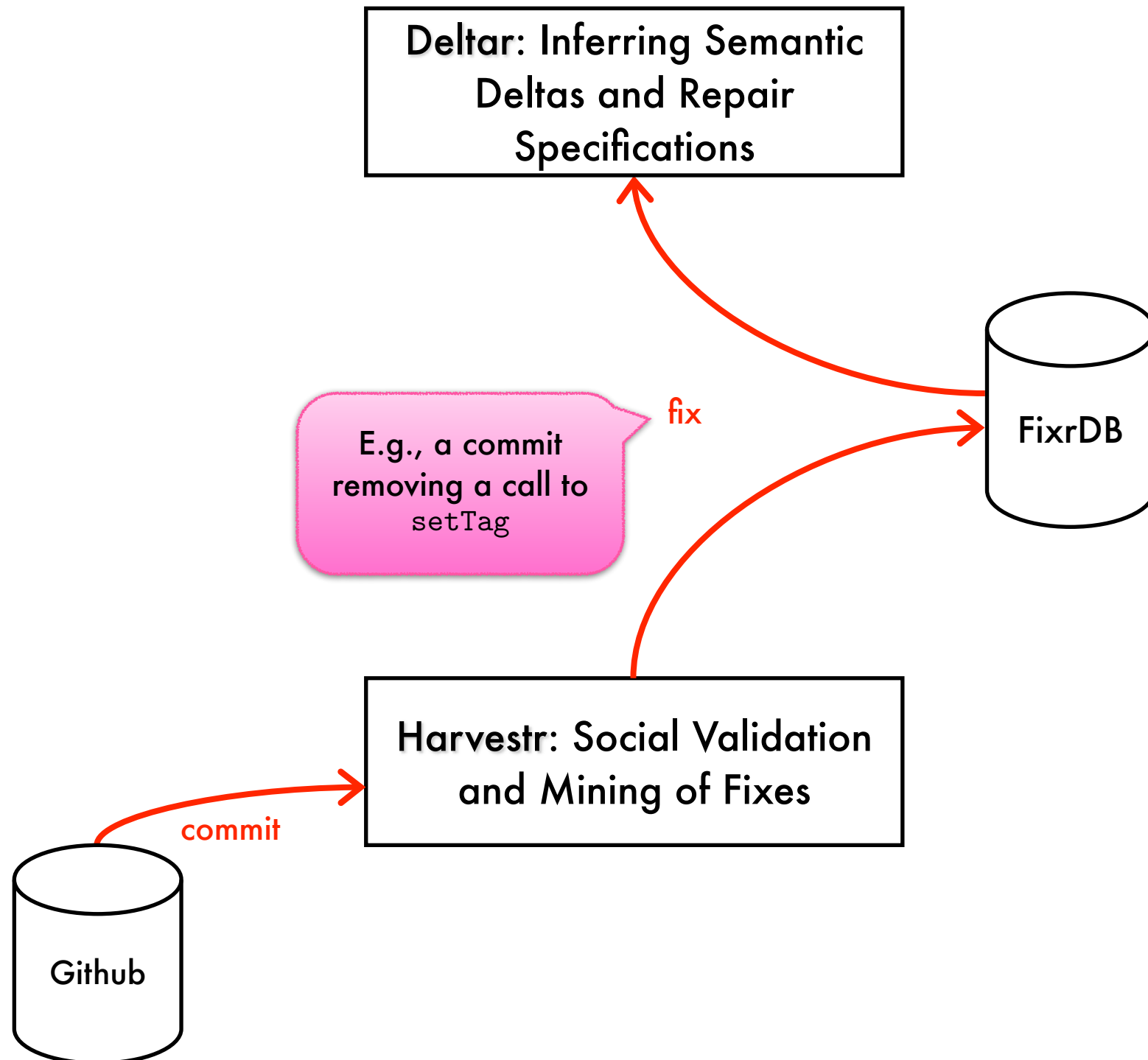




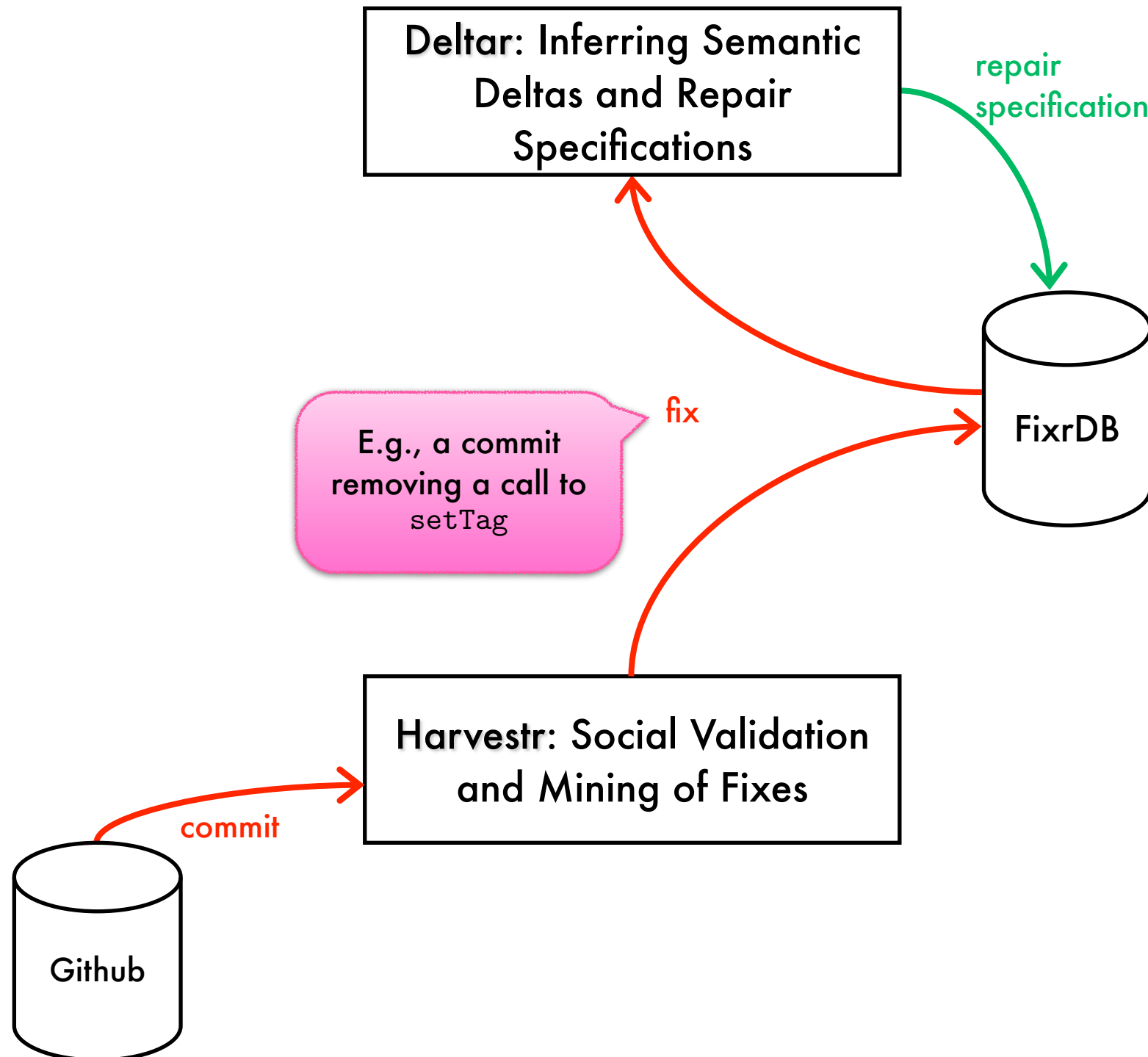
# Workflow 1a: Harvesting bugfix commits



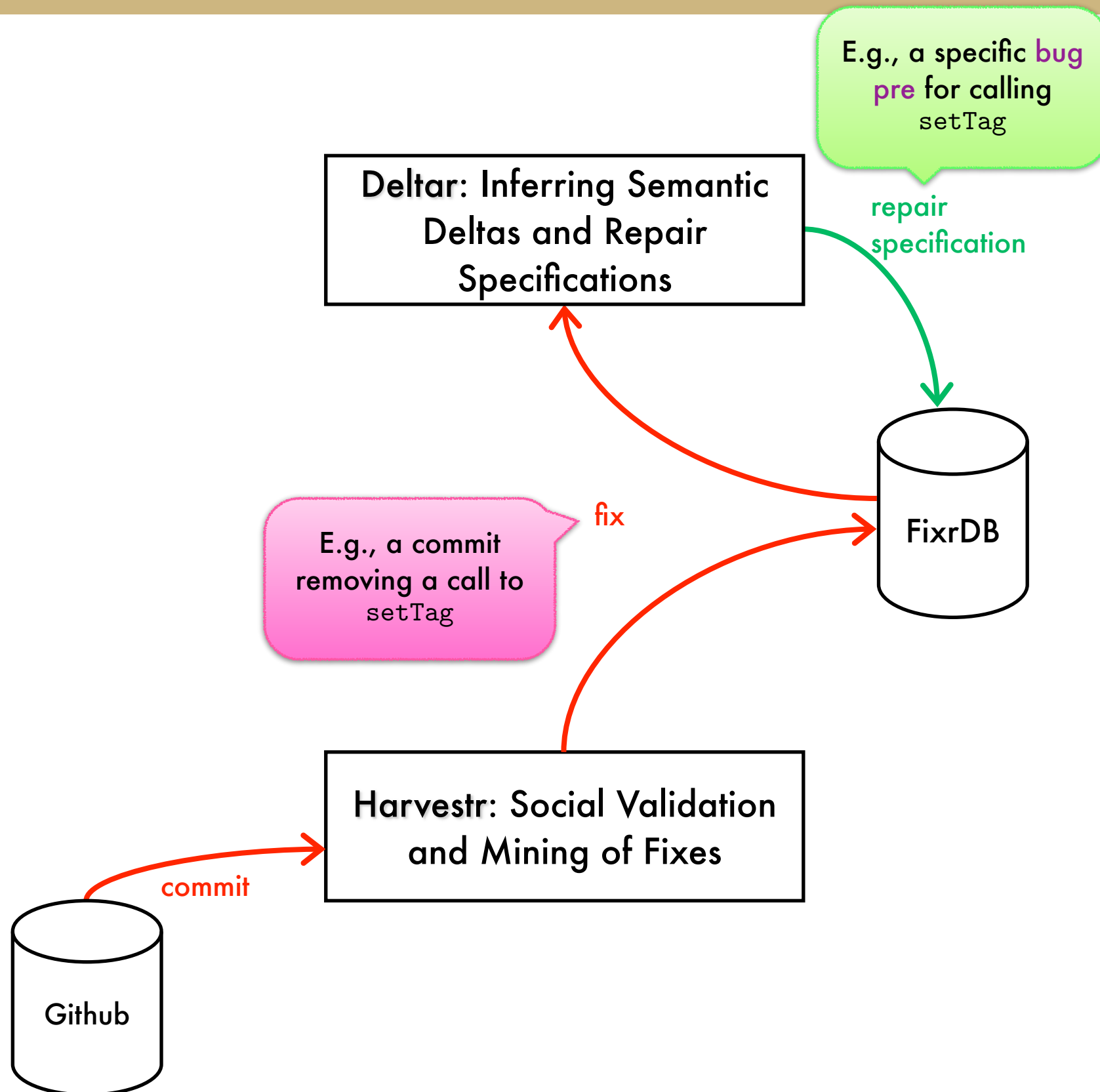
# Workflow 1a: Harvesting bugfix commits



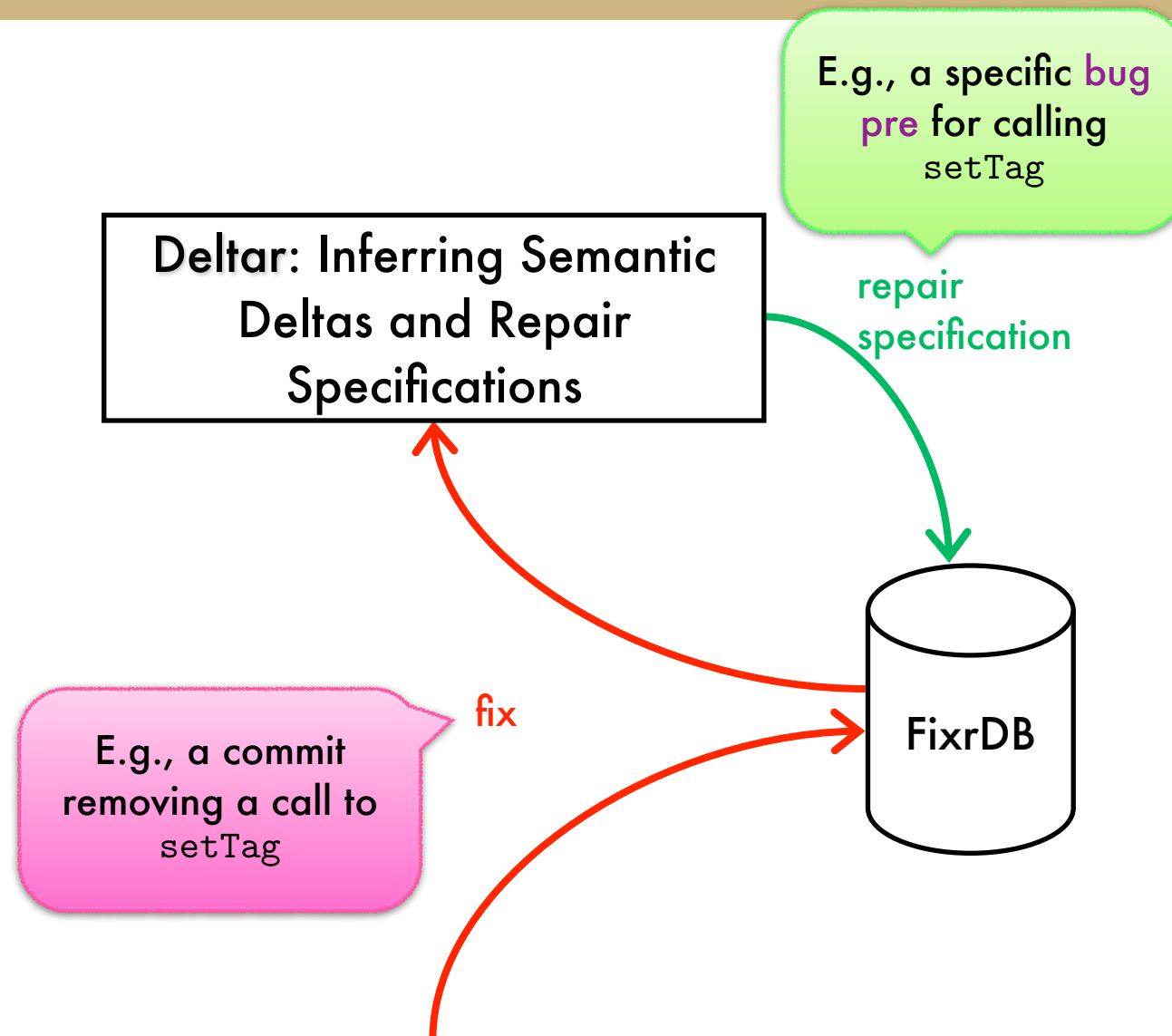
# Workflow 1a: Harvesting bugfix commits



# Workflow 1a: Harvesting bugfix commits



# Workflow 1a: Harvesting bugfix commits



E.g., a specific bug pre for calling setTag

Deltar: Inferring Semantic Deltas and Repair Specifications

repair specification

FixrDB

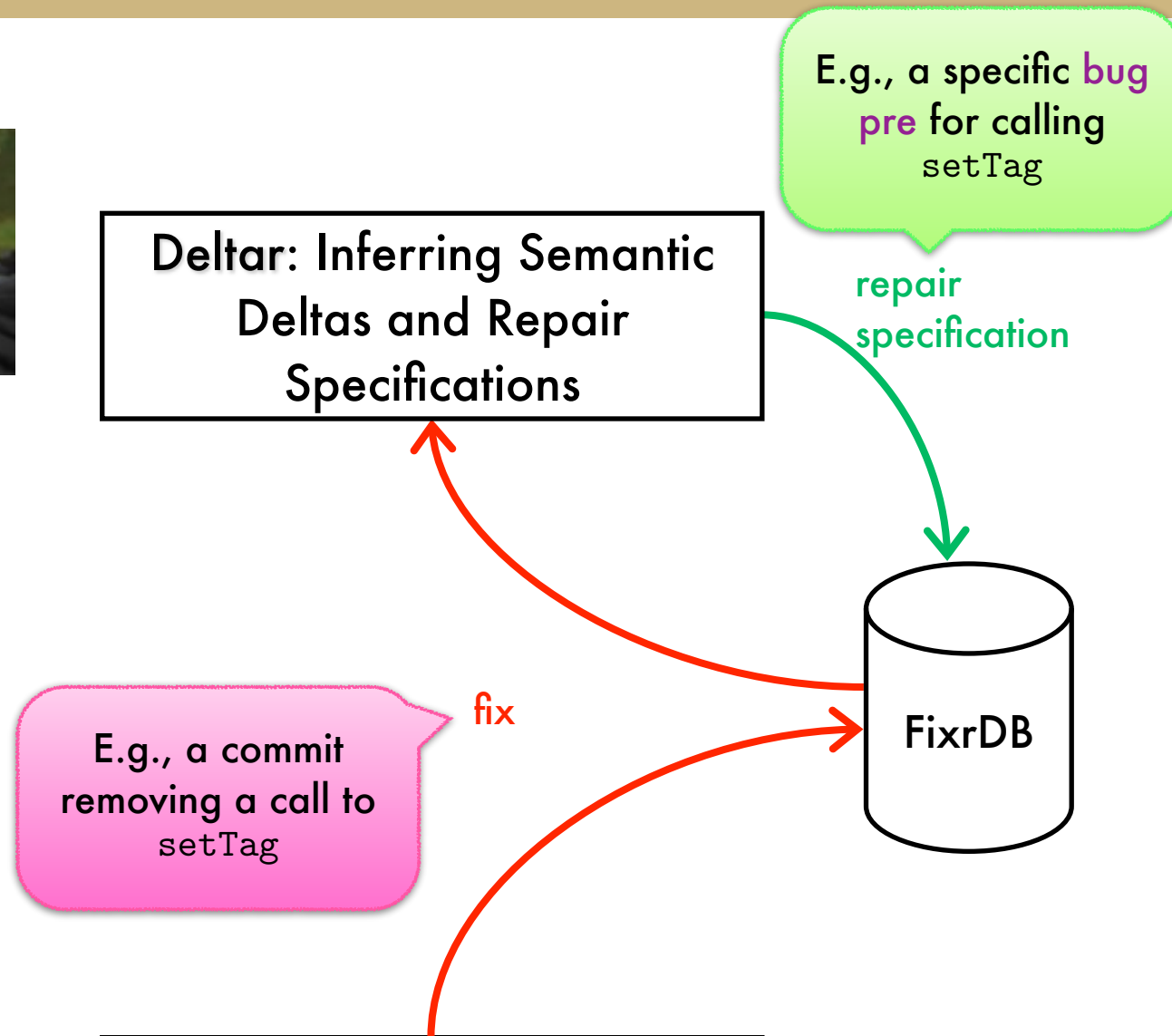
E.g., a commit removing a call to setTag

fix

Component: **Deltar** maps bugfixes to candidate repair specifications (including bug pre)

# Workflow 1a: Harvesting bugfix commits

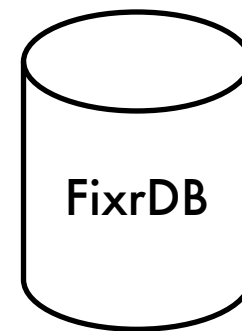
Shawn Meier



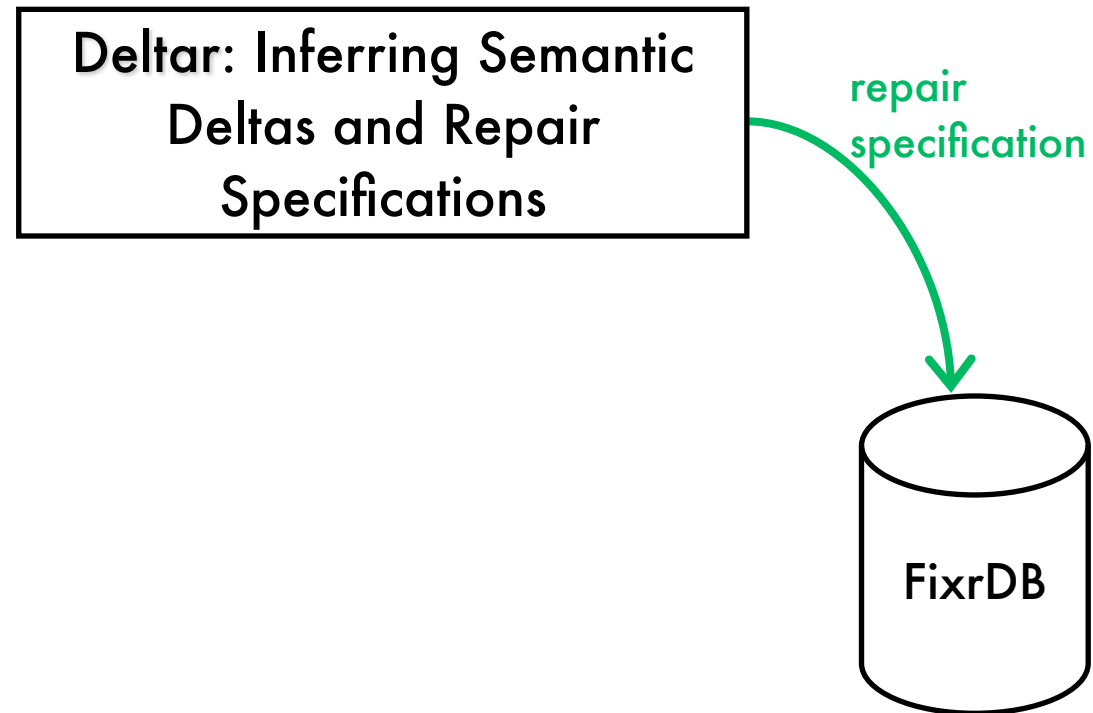
Component: **Deltar** maps bugfixes to candidate repair specifications (including bug pre)

# Workflow 1b: Aggregating repairs

**Deltar: Inferring Semantic  
Deltas and Repair  
Specifications**

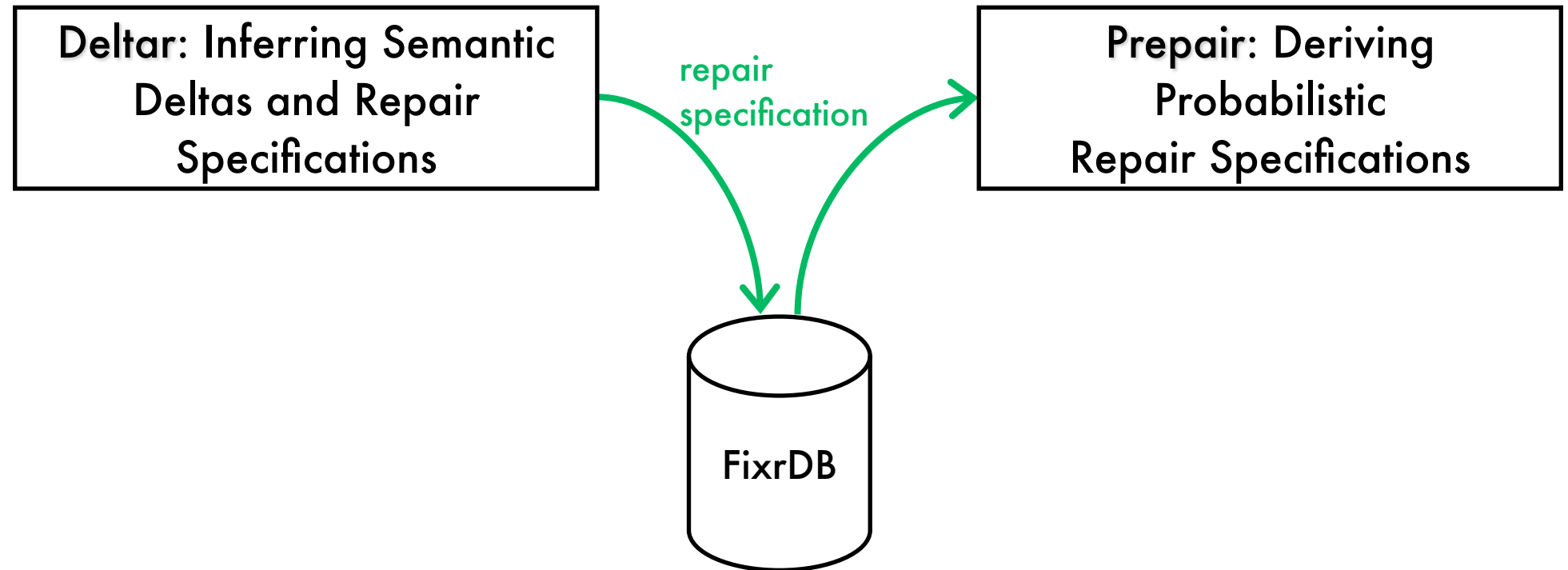


# Workflow 1b: Aggregating repairs

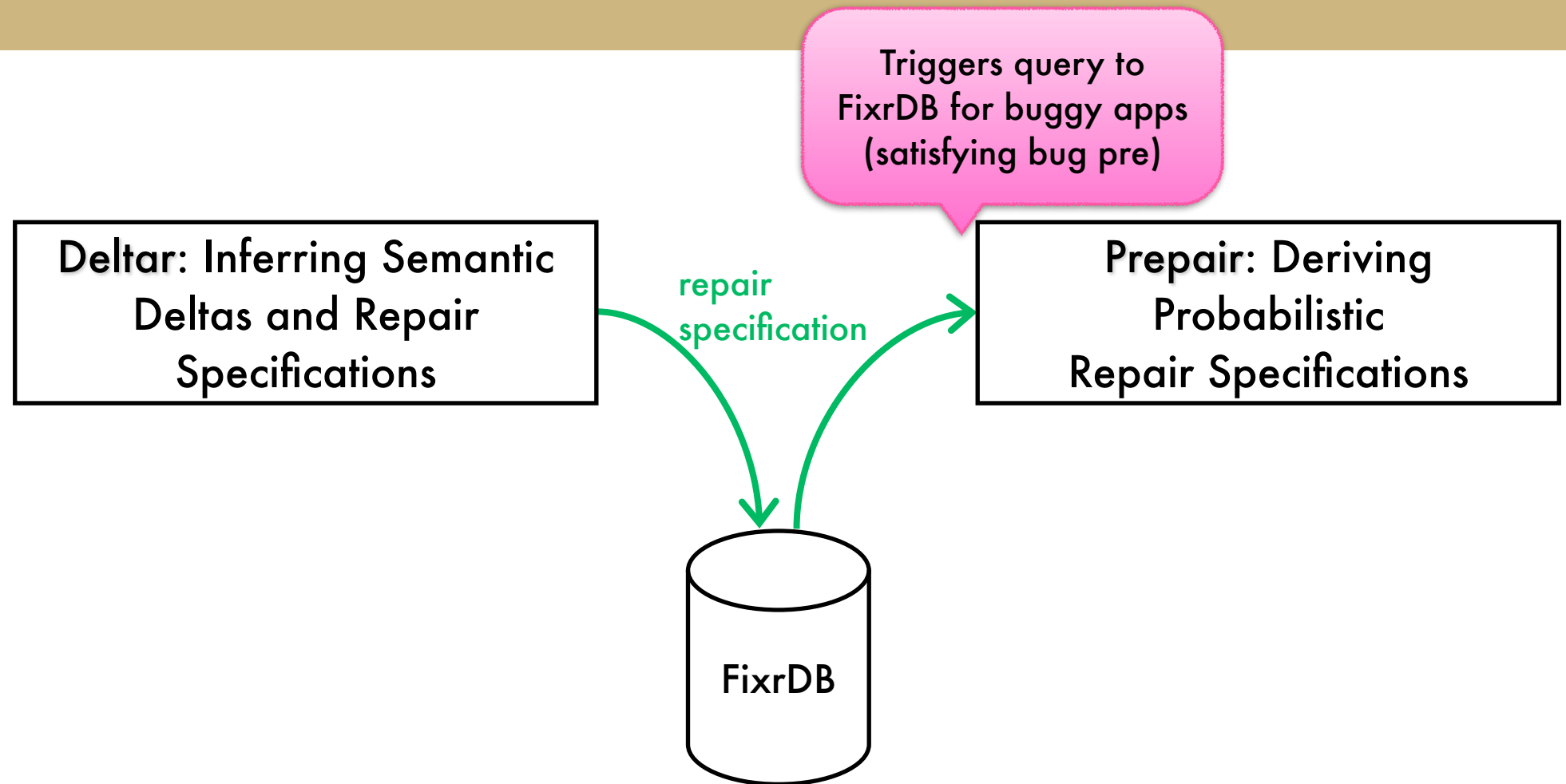




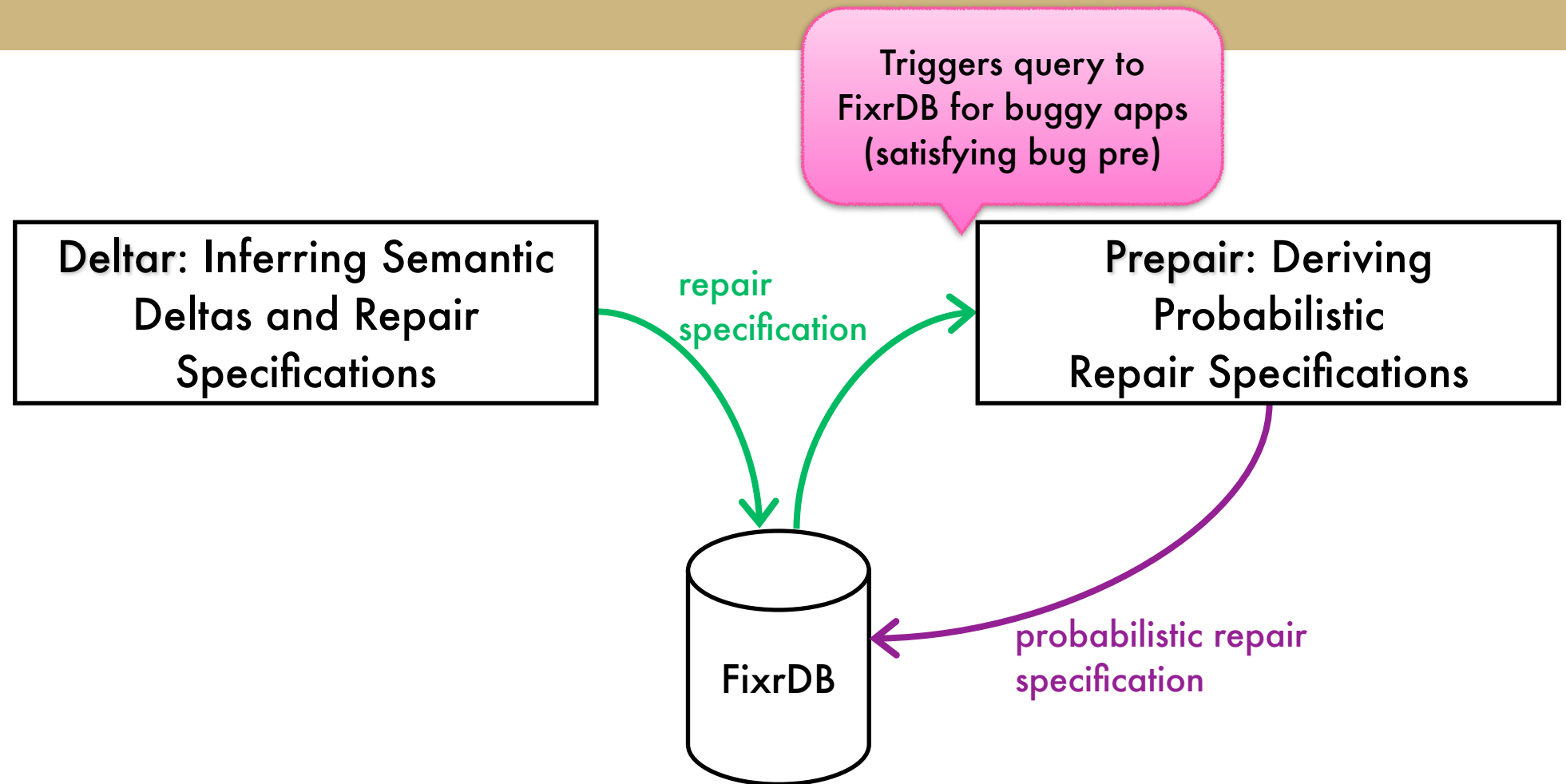
# Workflow 1b: Aggregating repairs



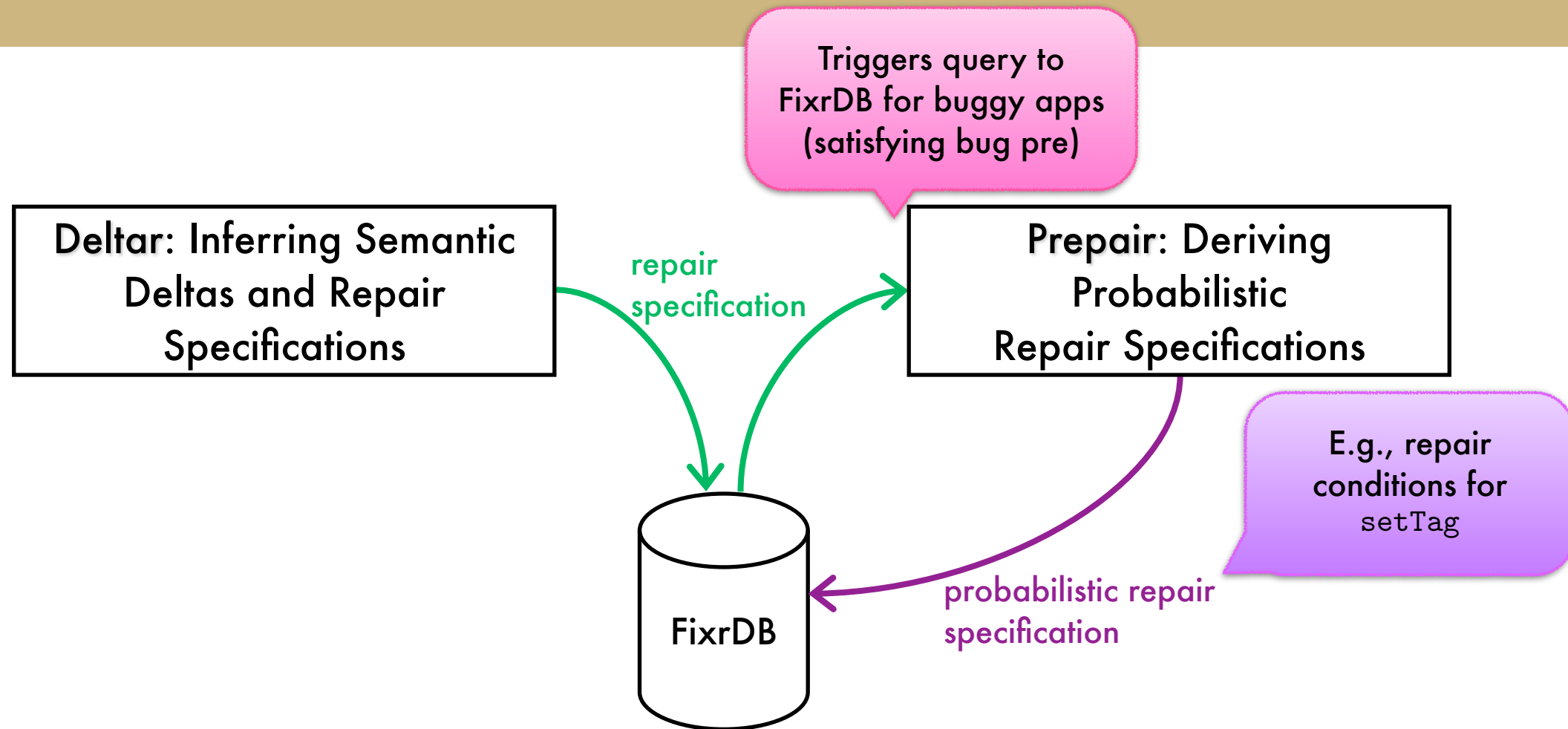
# Workflow 1b: Aggregating repairs



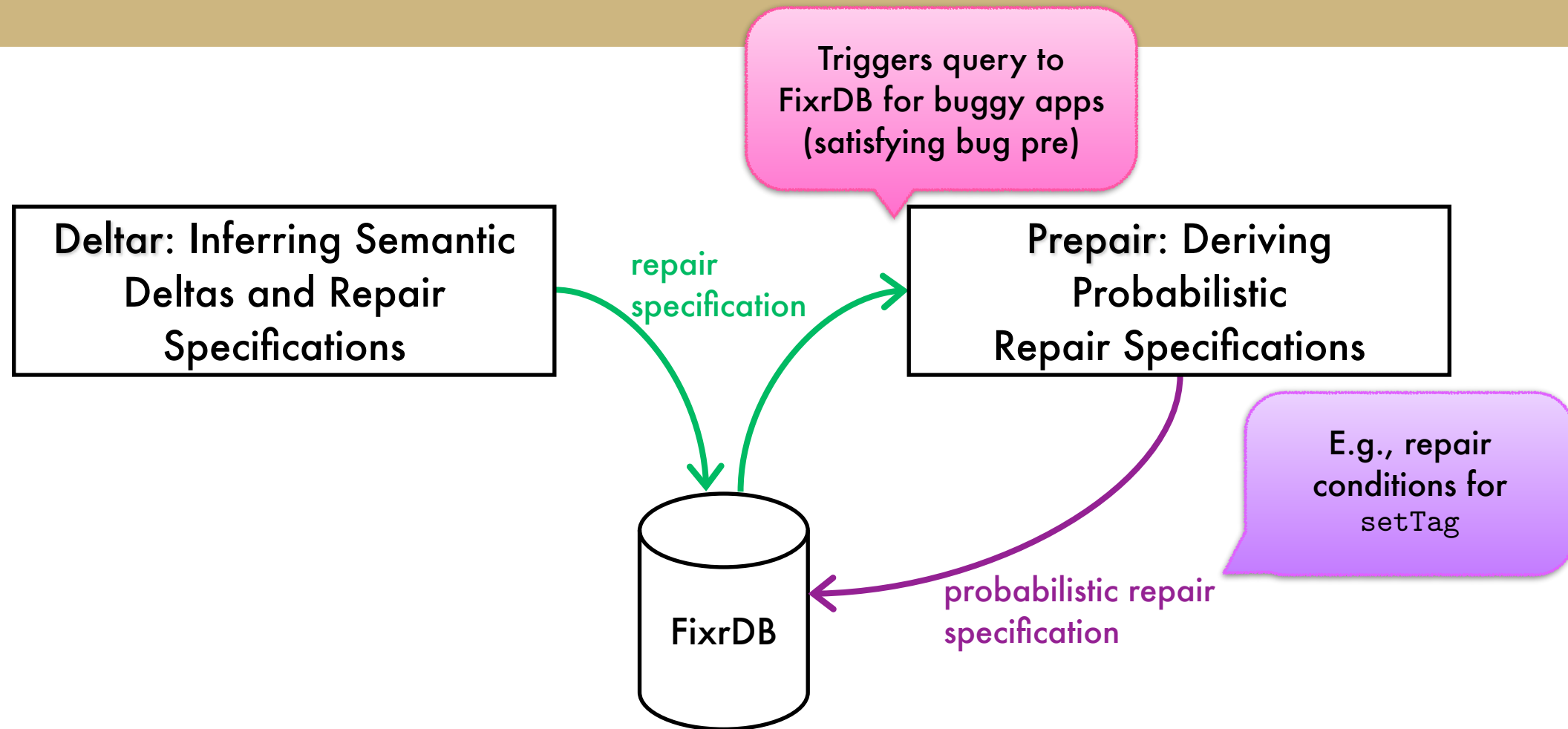
# Workflow 1b: Aggregating repairs



# Workflow 1b: Aggregating repairs



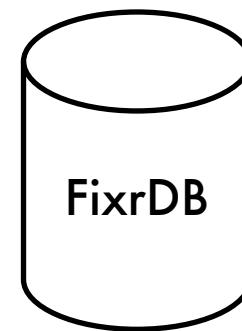
# Workflow 1b: Aggregating repairs



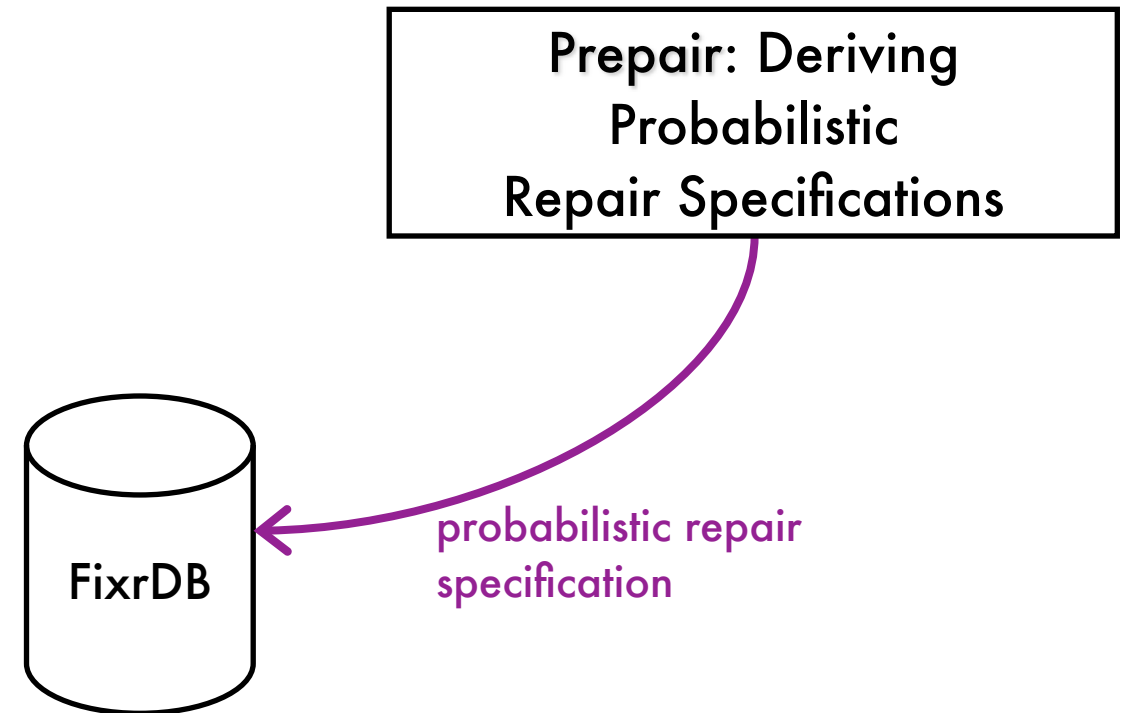
Component: **Prepair** reduces candidate repair specifications to generalized **probabilistic** repair specifications

# Workflow 1c: Synthesizing patches

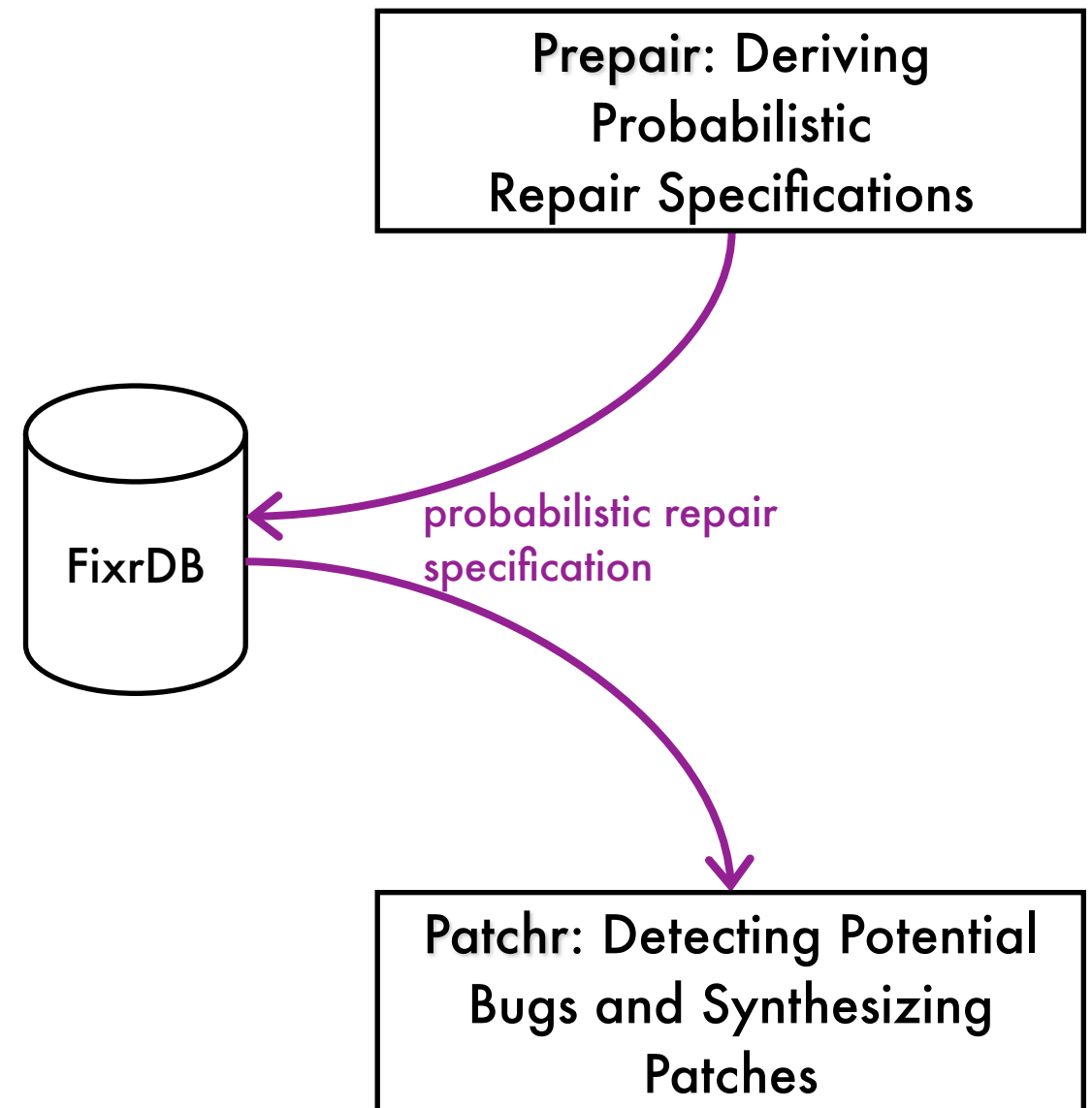
**Prepair: Deriving  
Probabilistic  
Repair Specifications**



# Workflow 1c: Synthesizing patches

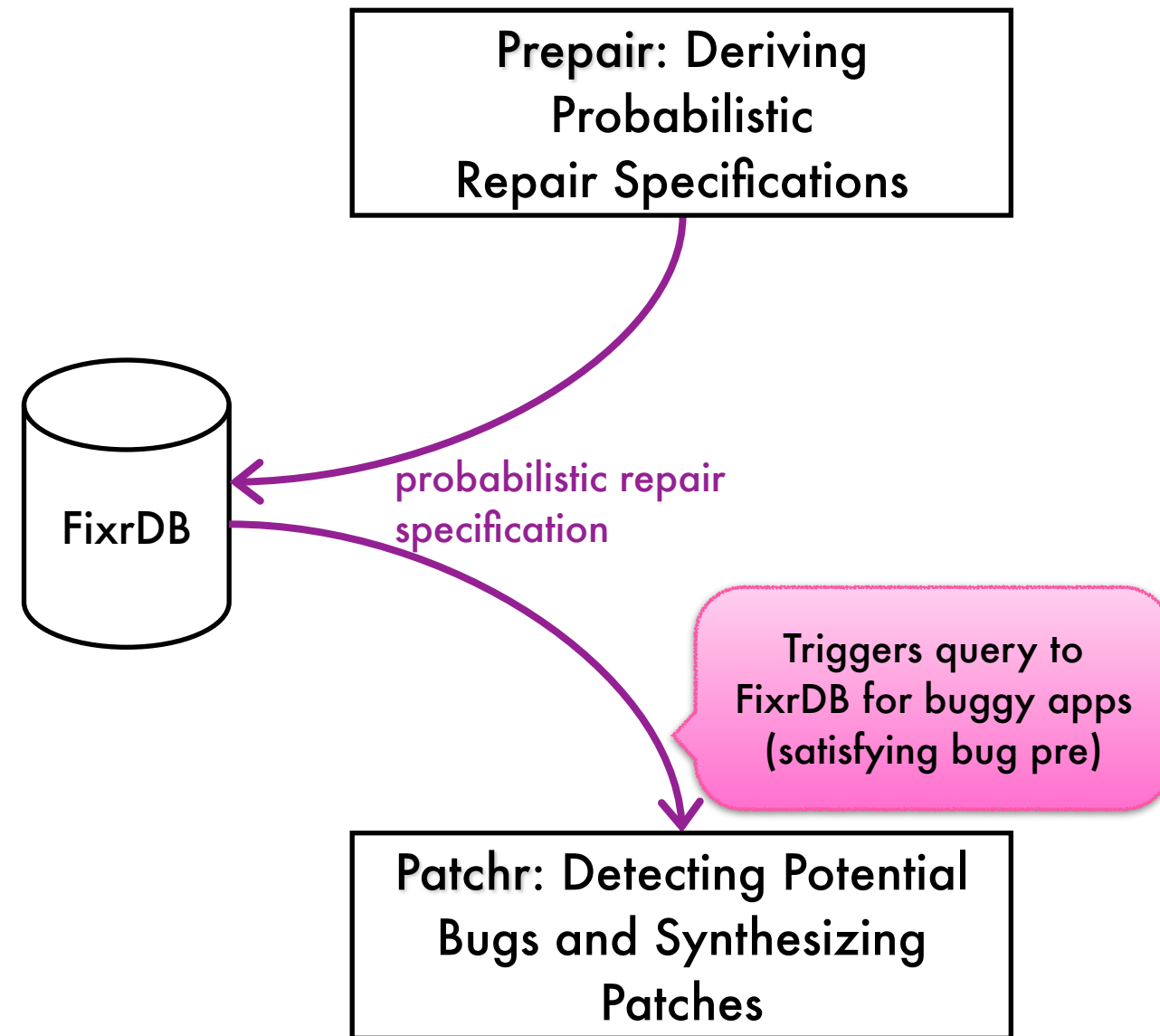


# Workflow 1c: Synthesizing patches

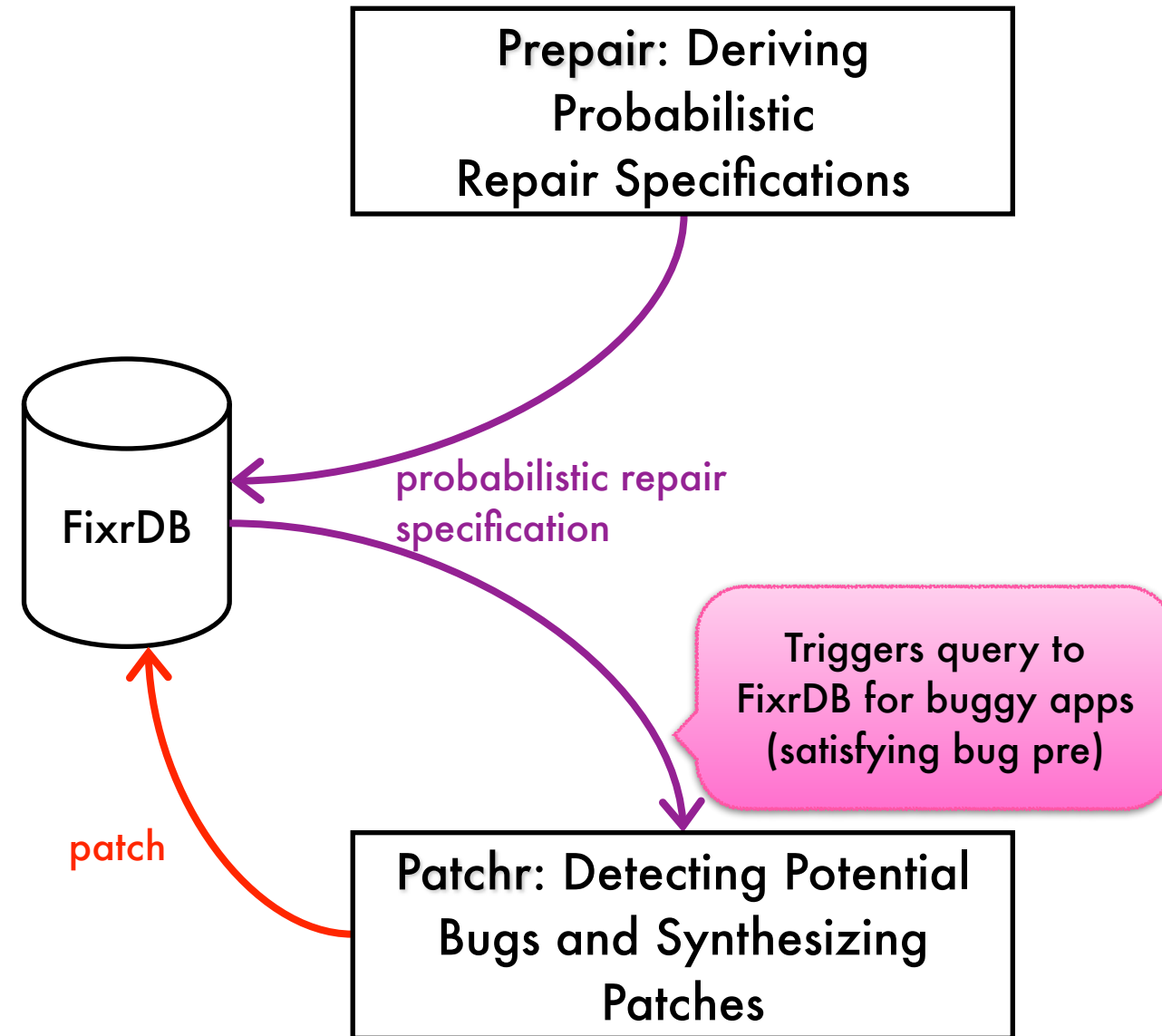




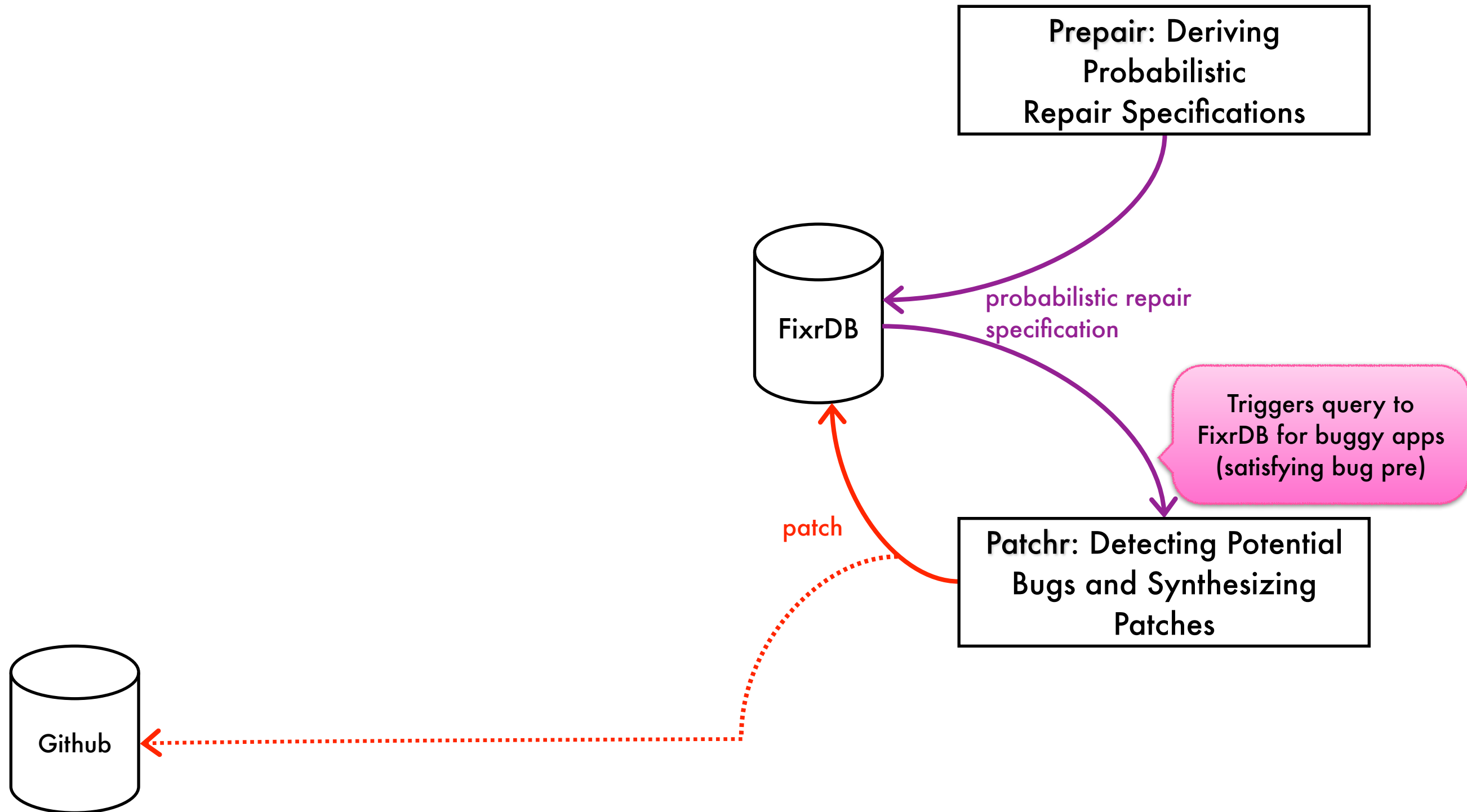
# Workflow 1c: Synthesizing patches



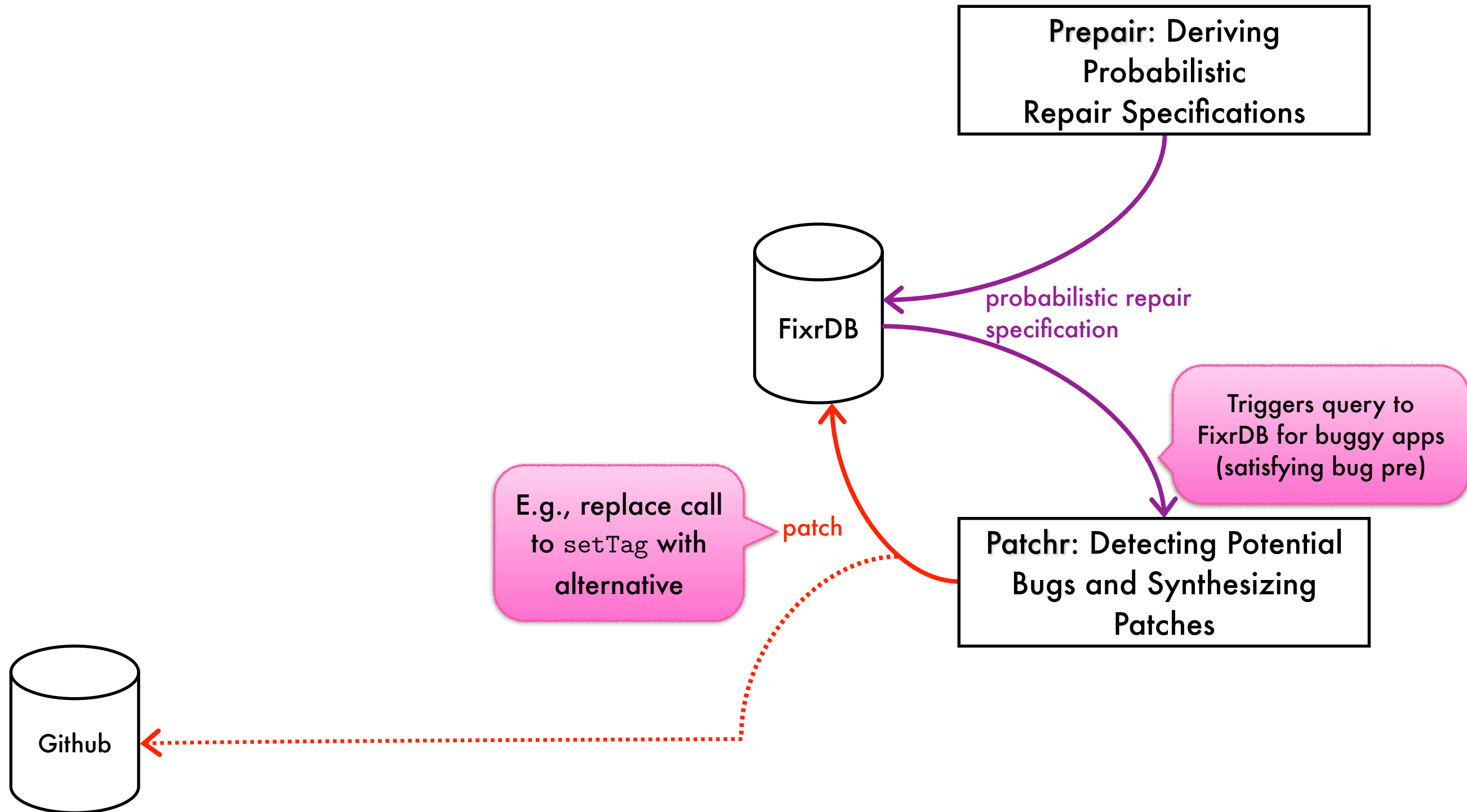
# Workflow 1c: Synthesizing patches



# Workflow 1c: Synthesizing patches

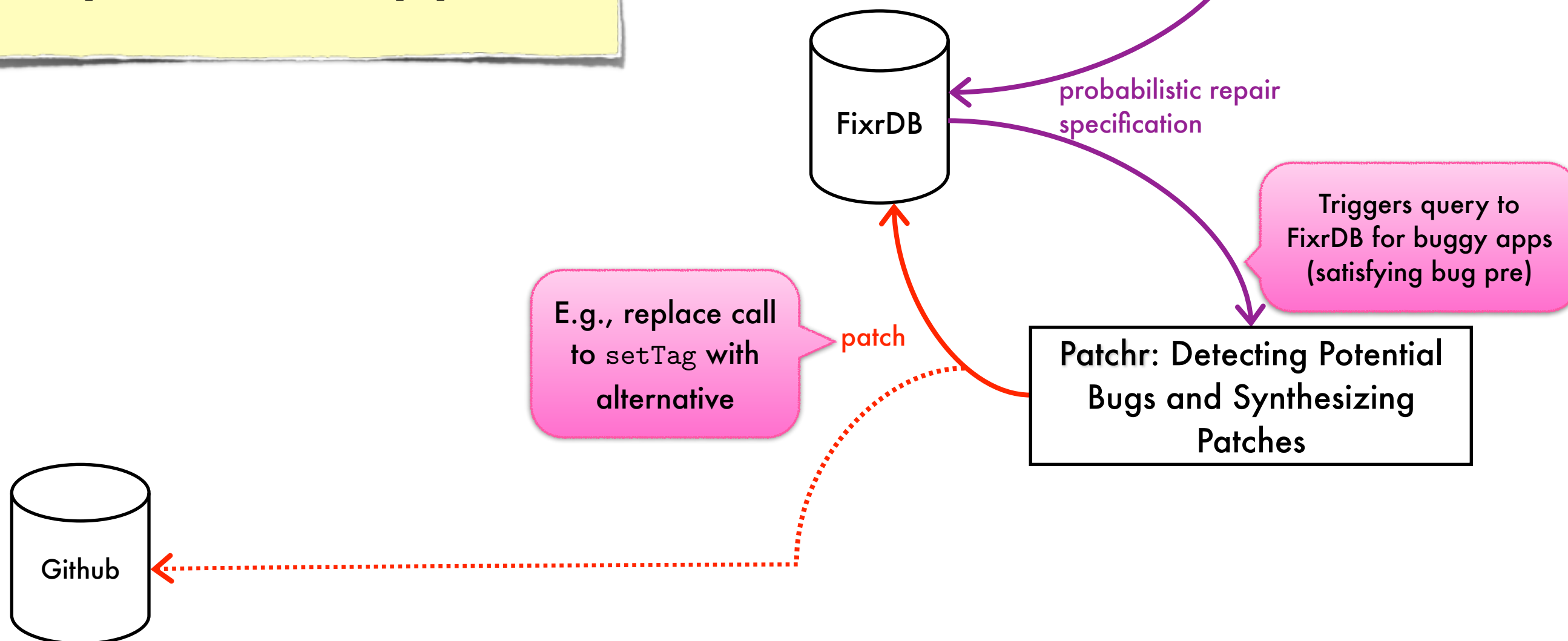


# Workflow 1c: Synthesizing patches



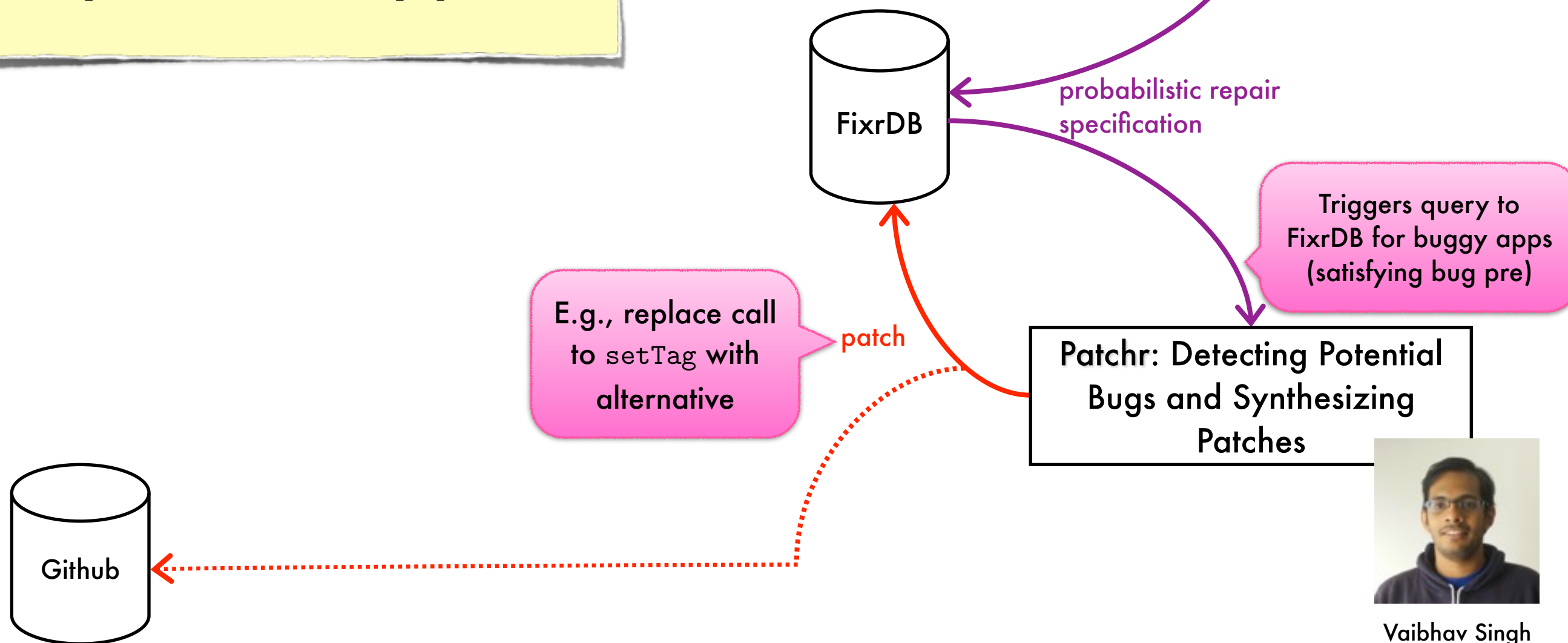
# Workflow 1c: Synthesizing patches

Component: **Patchr**  
*maps* buggy apps to  
patched apps



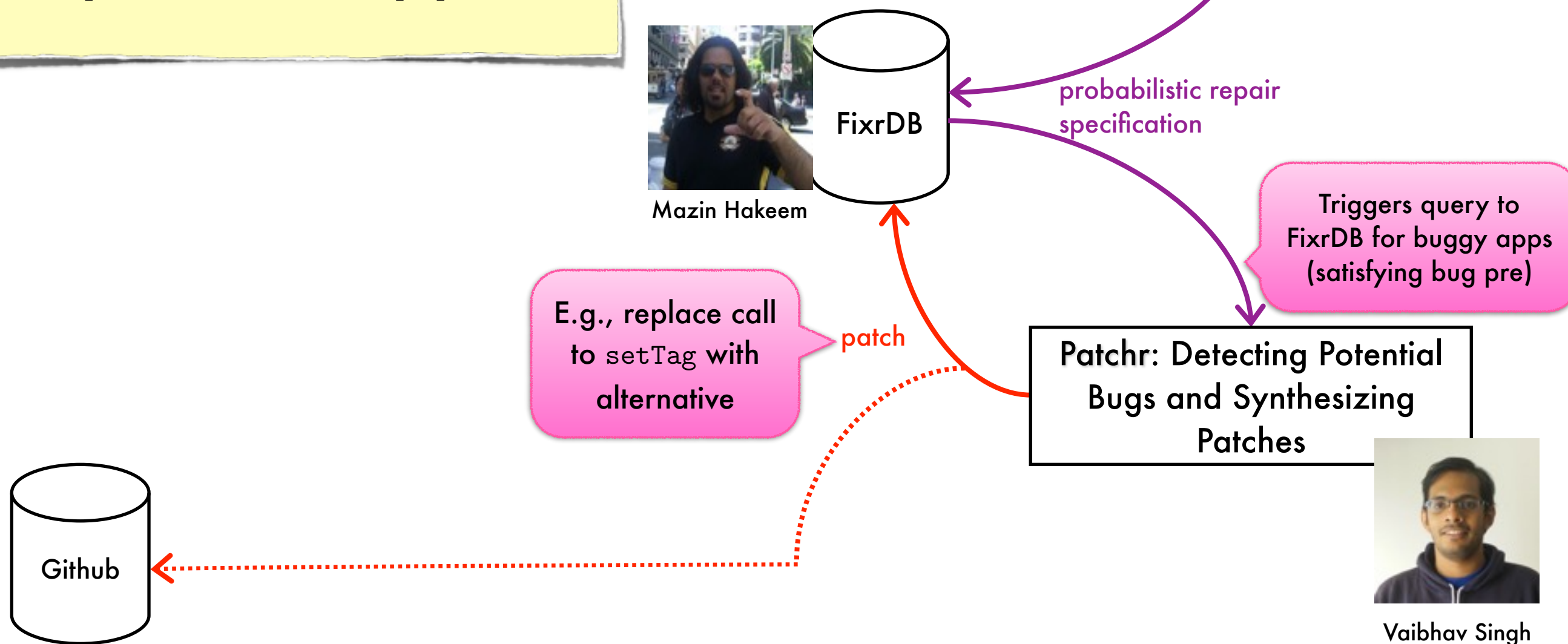
# Workflow 1c: Synthesizing patches

Component: **Patchr**  
*maps buggy apps to patched apps*



# Workflow 1c: Synthesizing patches

Component: **Patchr**  
*maps buggy apps to patched apps*

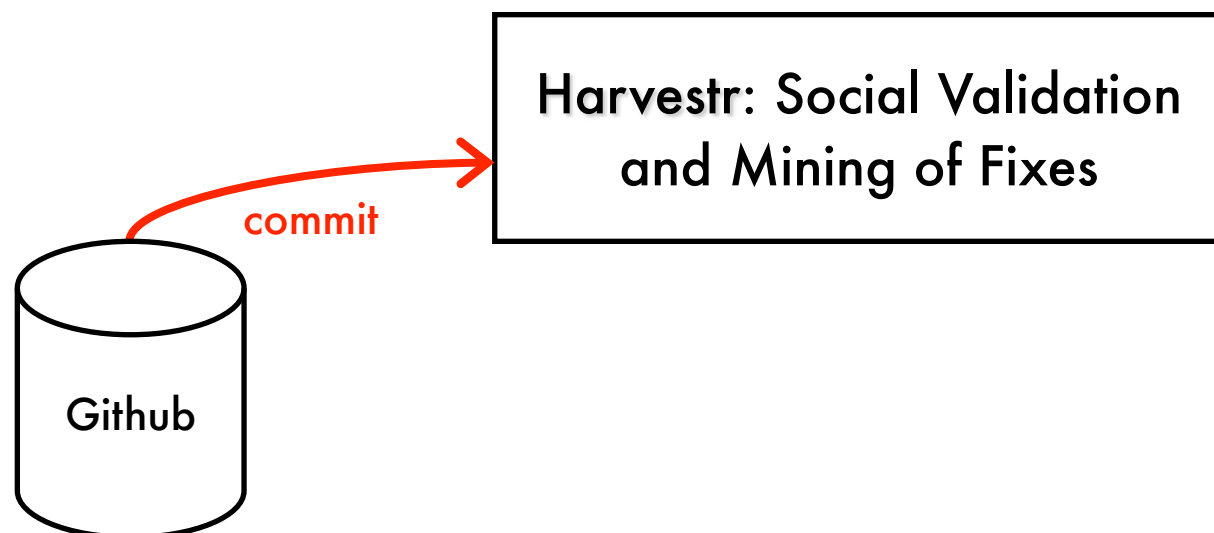




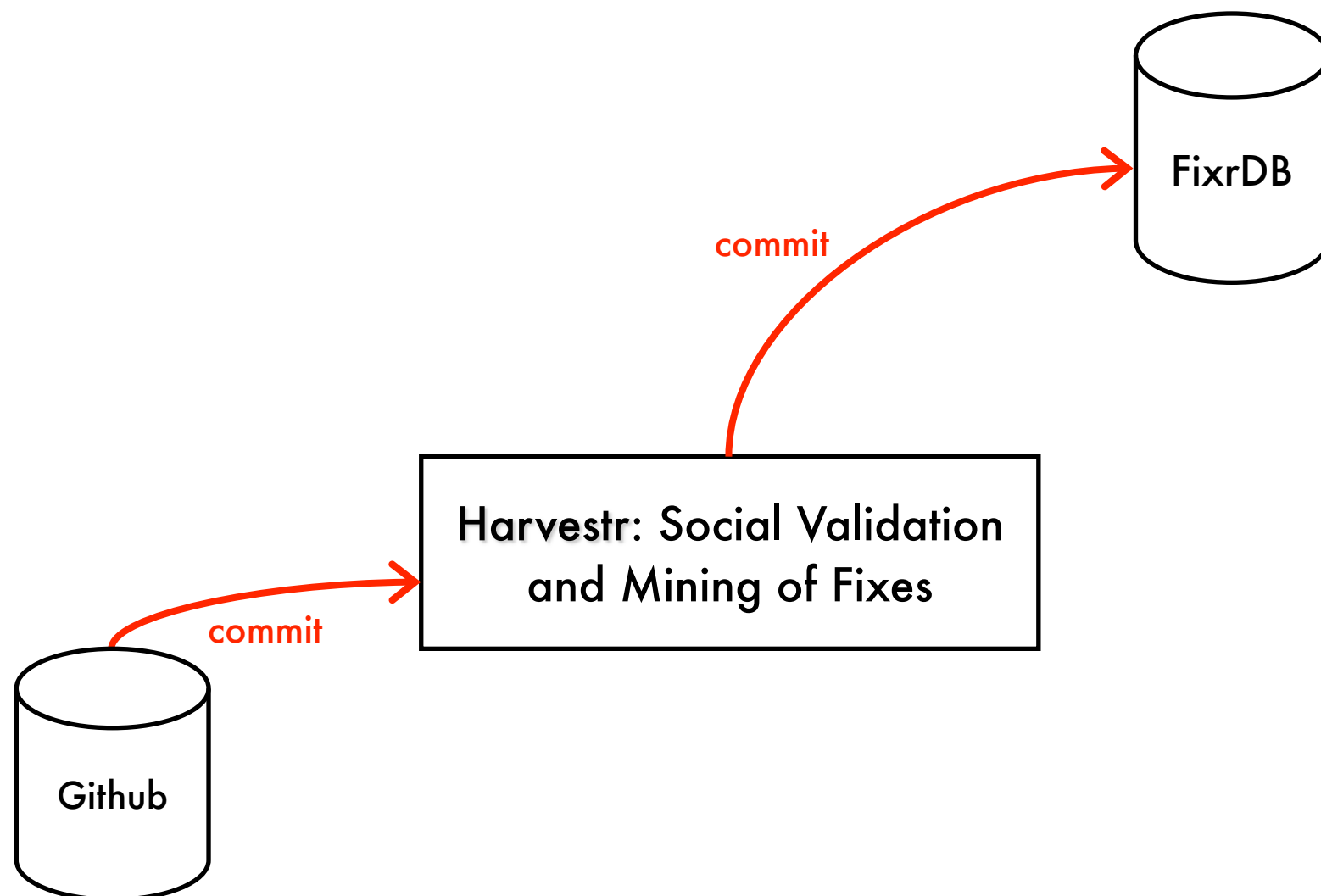
# Workflow 0: Continuous commit harvesting, buggy app patching, and social validation

**Harvestr: Social Validation  
and Mining of Fixes**

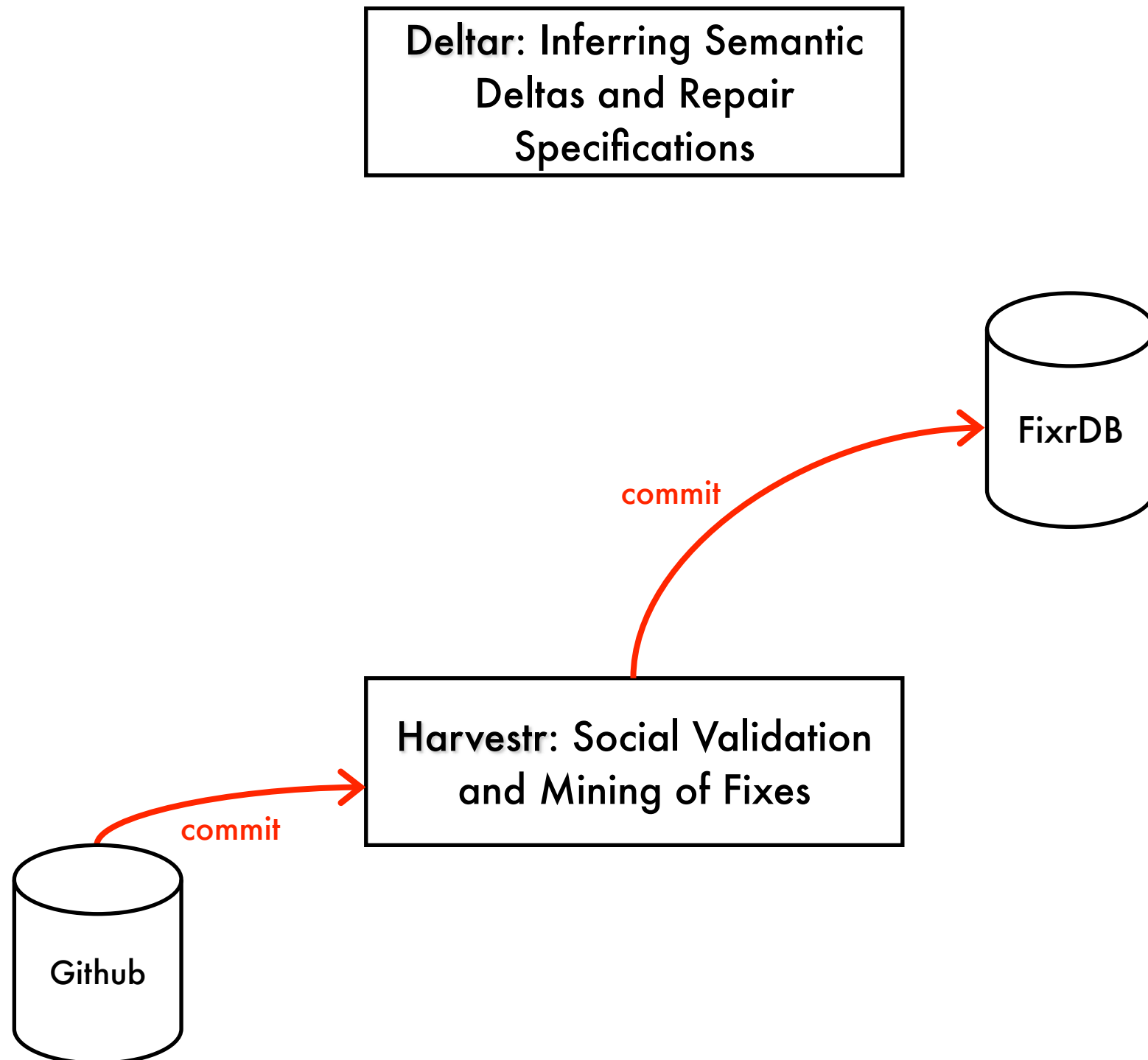
# Workflow 0: Continuous commit harvesting, buggy app patching, and social validation



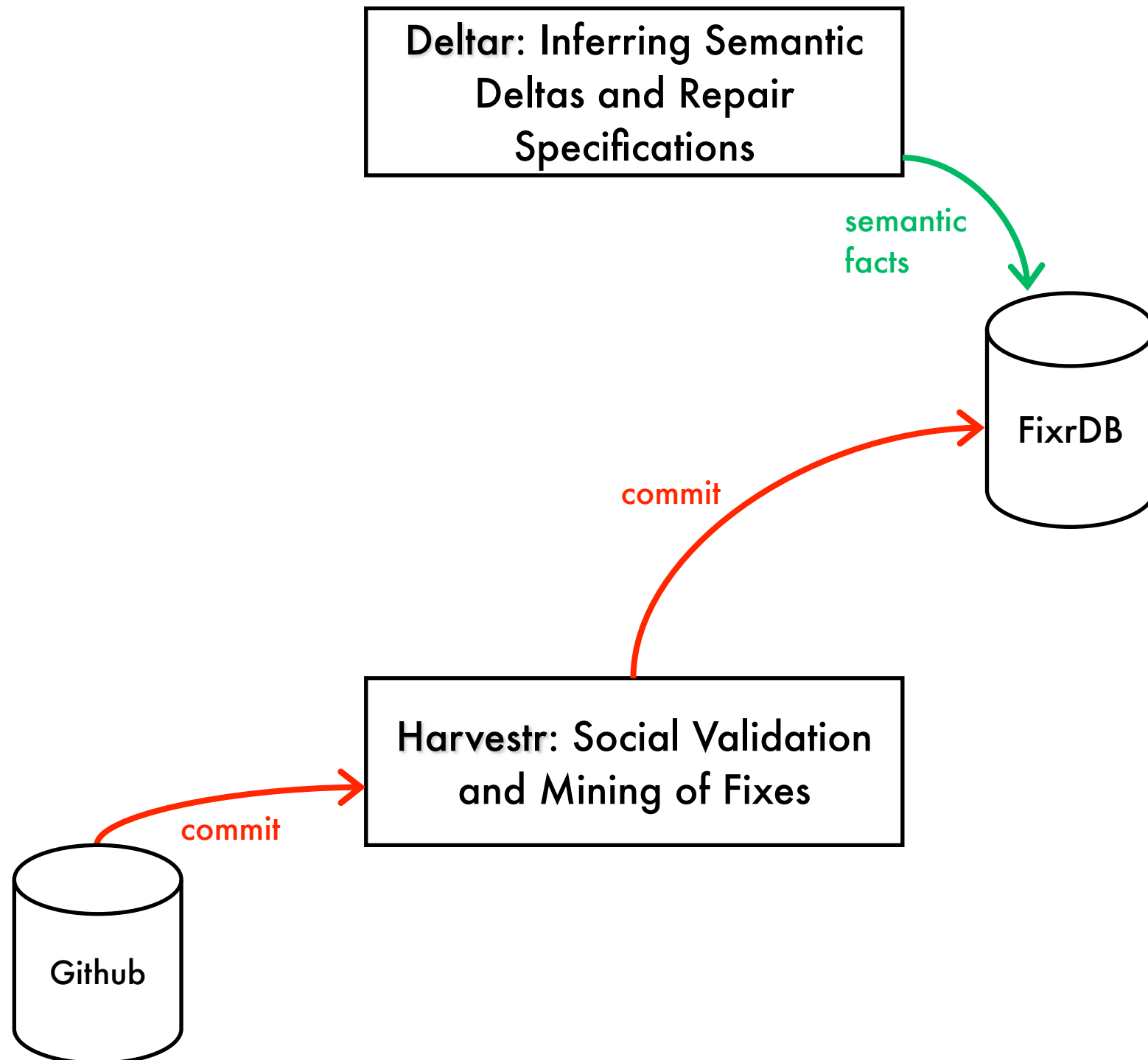
# Workflow 0: Continuous commit harvesting, buggy app patching, and social validation



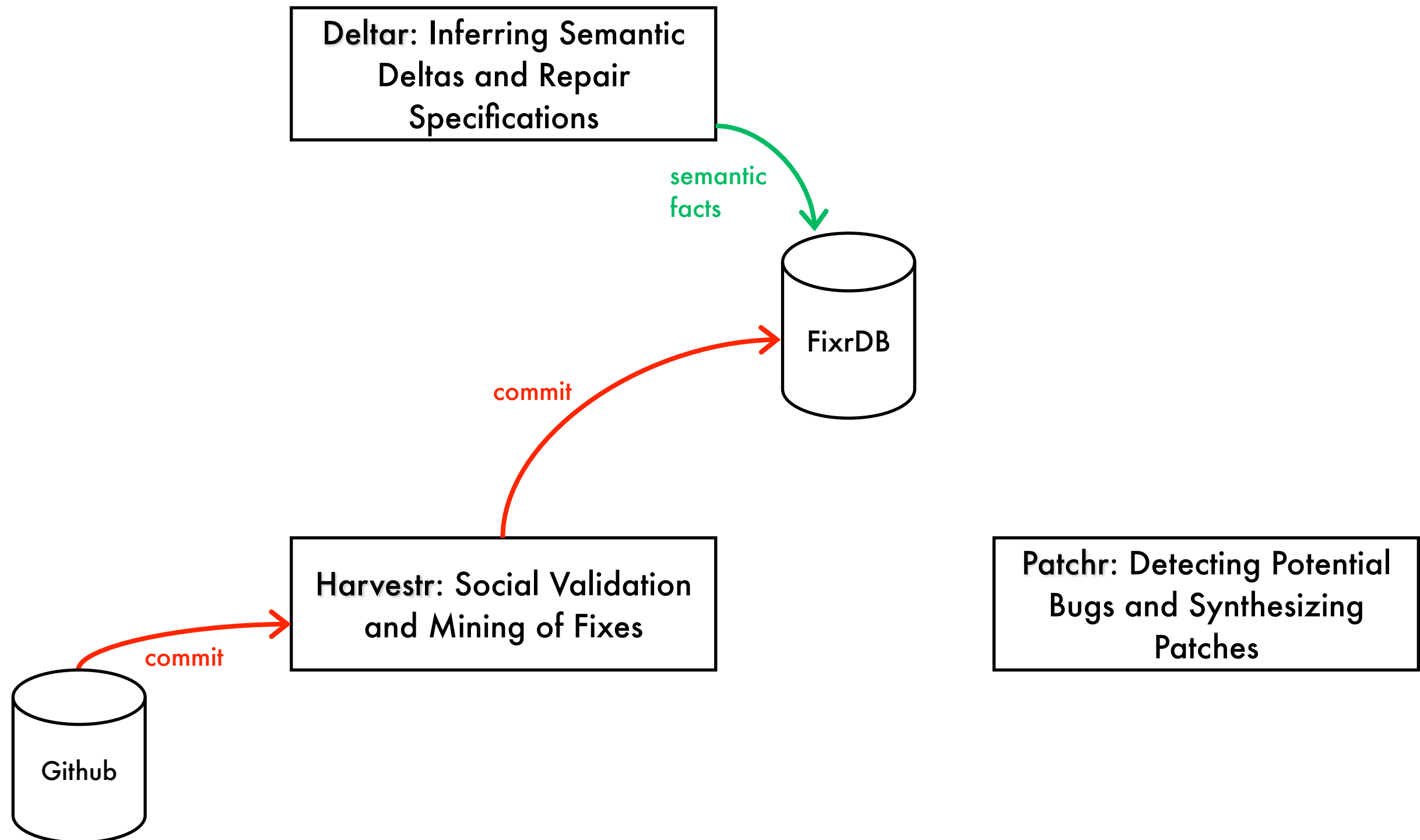
# Workflow 0: Continuous commit harvesting, buggy app patching, and social validation



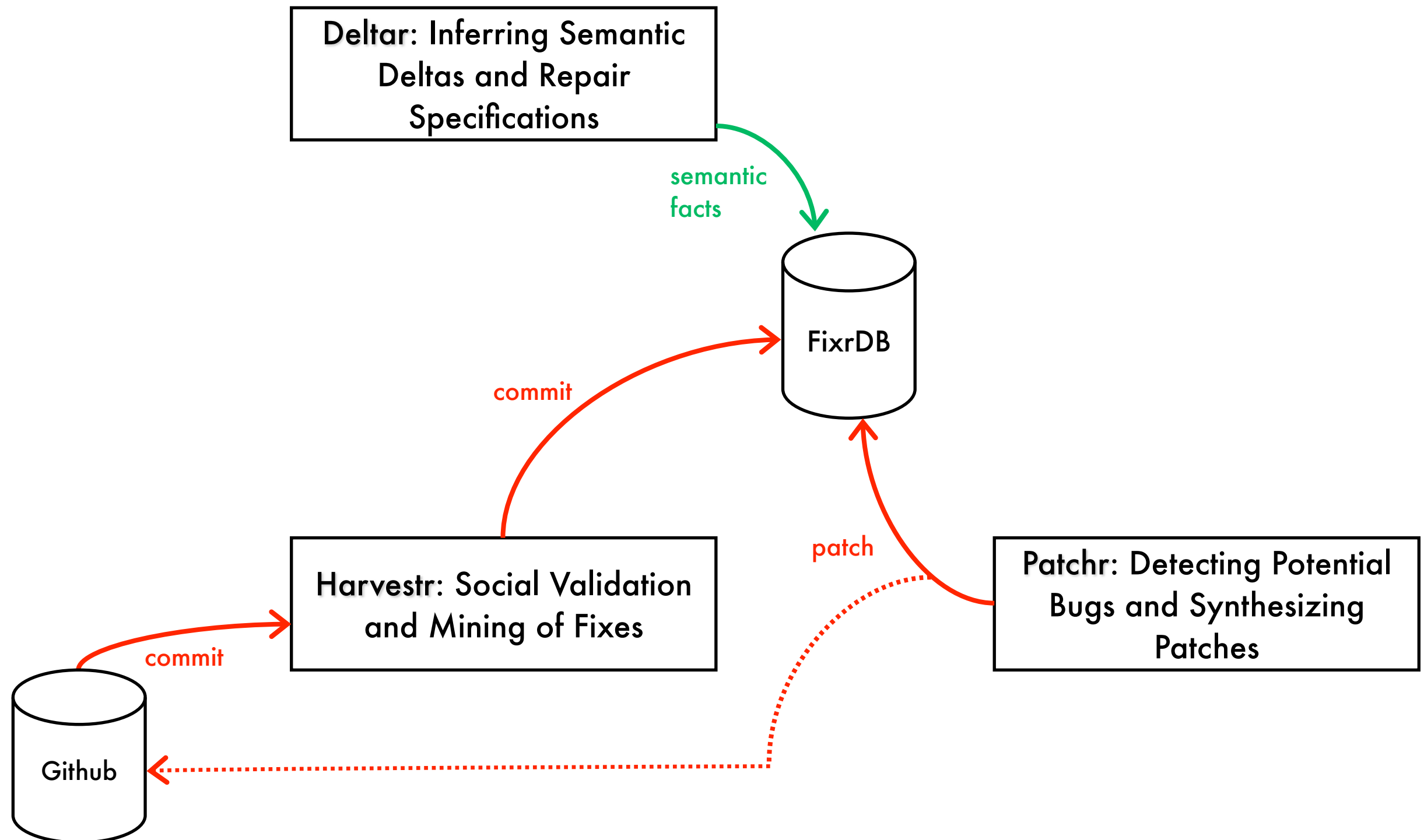
# Workflow 0: Continuous commit harvesting, buggy app patching, and social validation



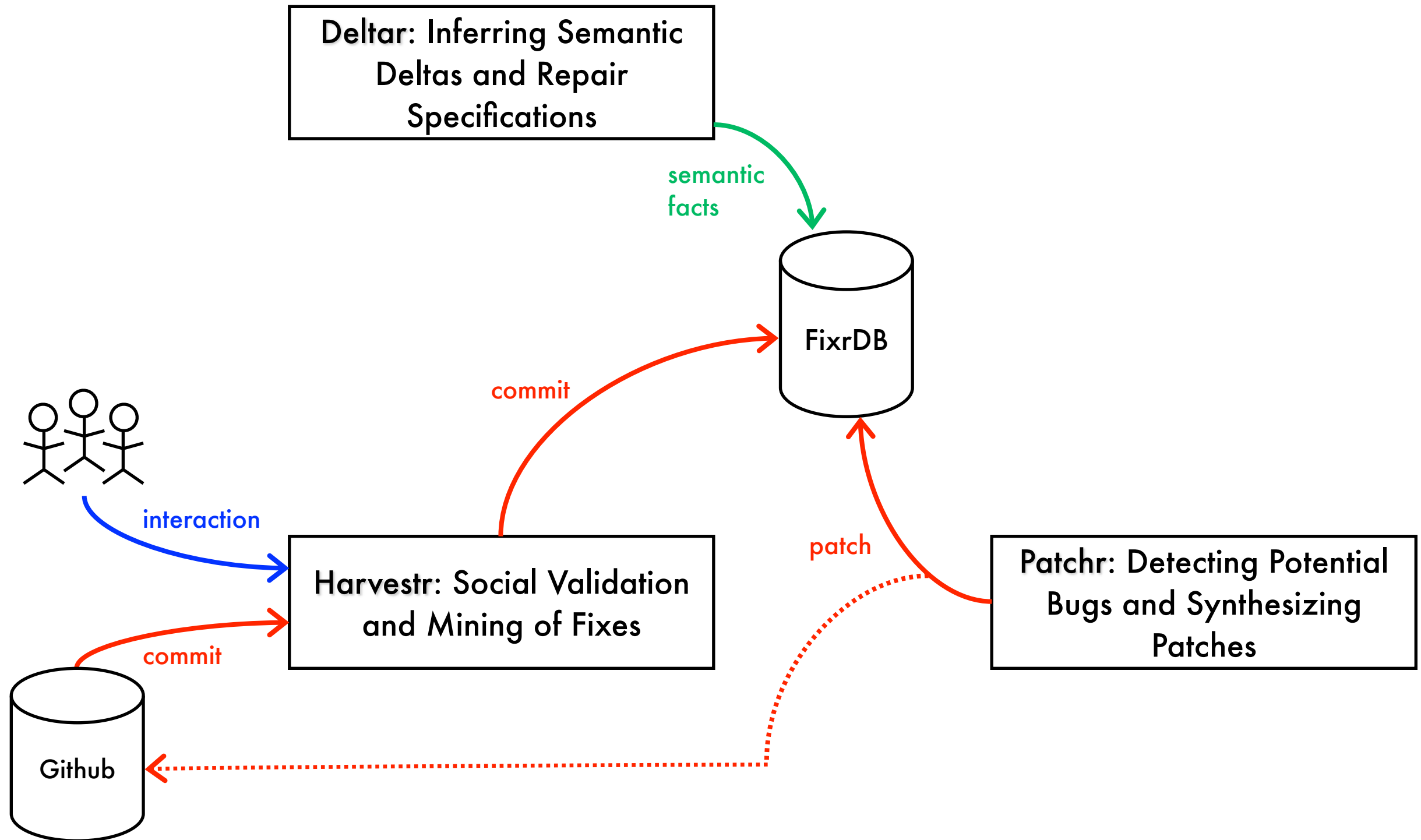
# Workflow 0: Continuous commit harvesting, buggy app patching, and social validation



# Workflow 0: Continuous commit harvesting, buggy app patching, and social validation

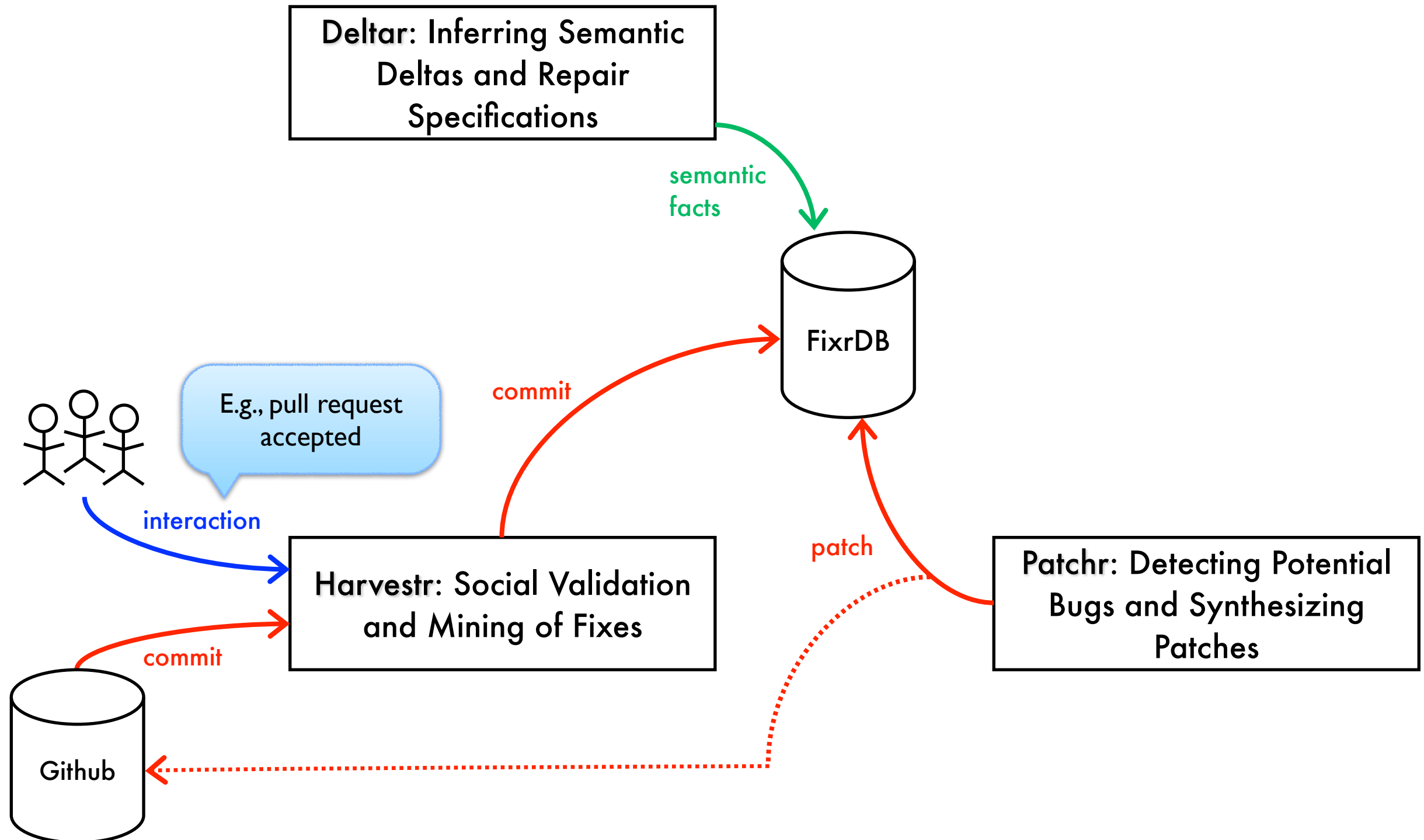


# Workflow 0: Continuous commit harvesting, buggy app patching, and social validation

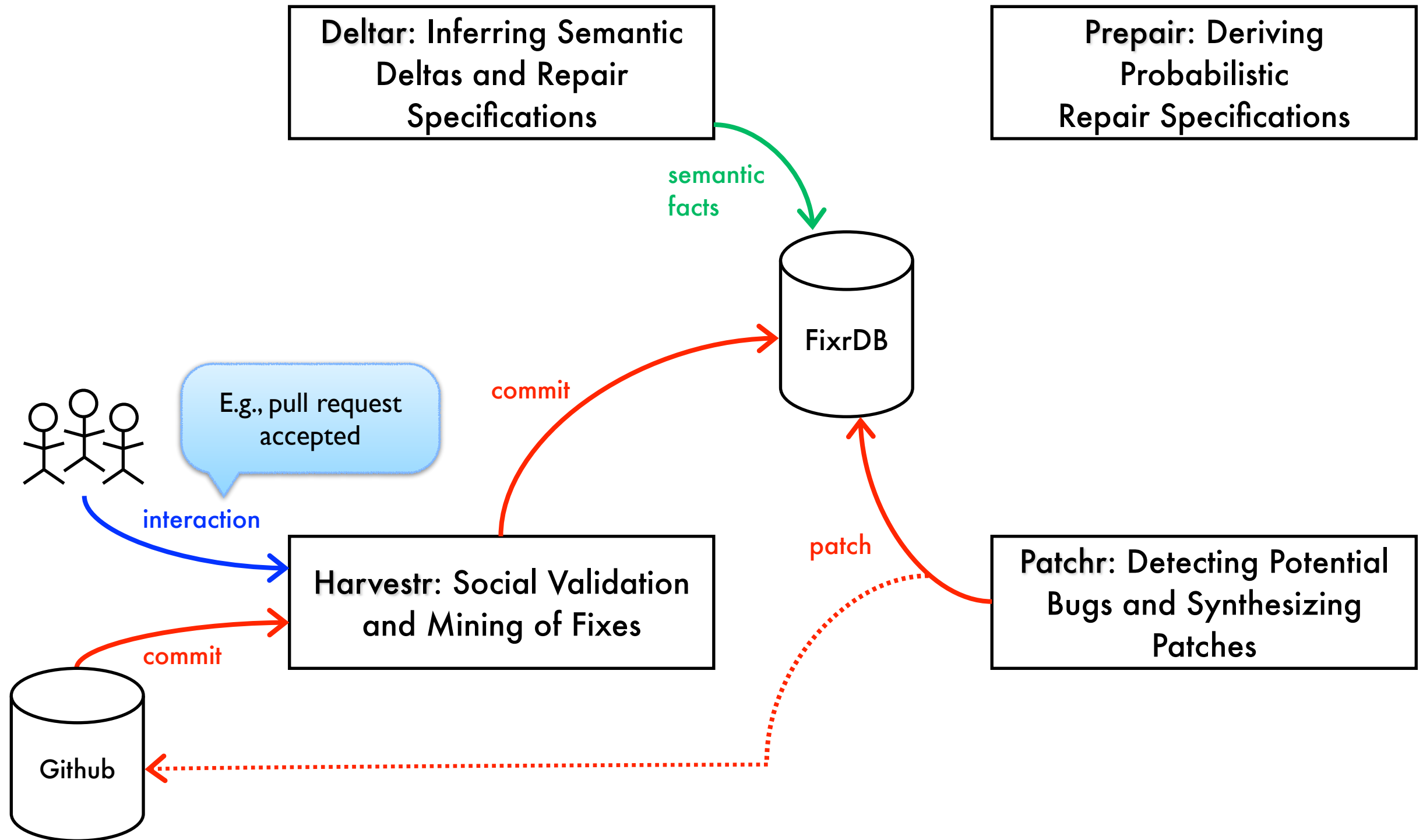




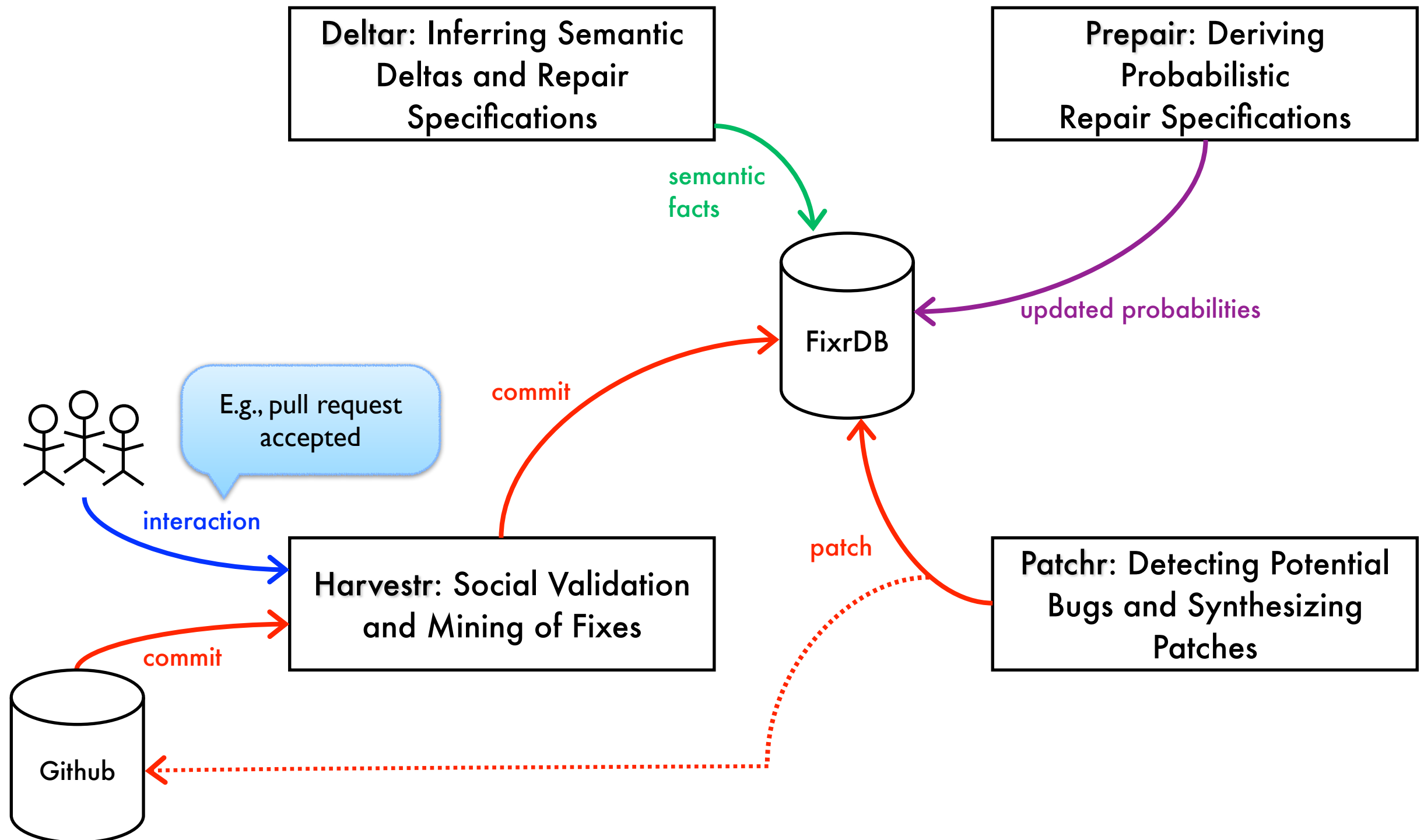
# Workflow 0: Continuous commit harvesting, buggy app patching, and social validation



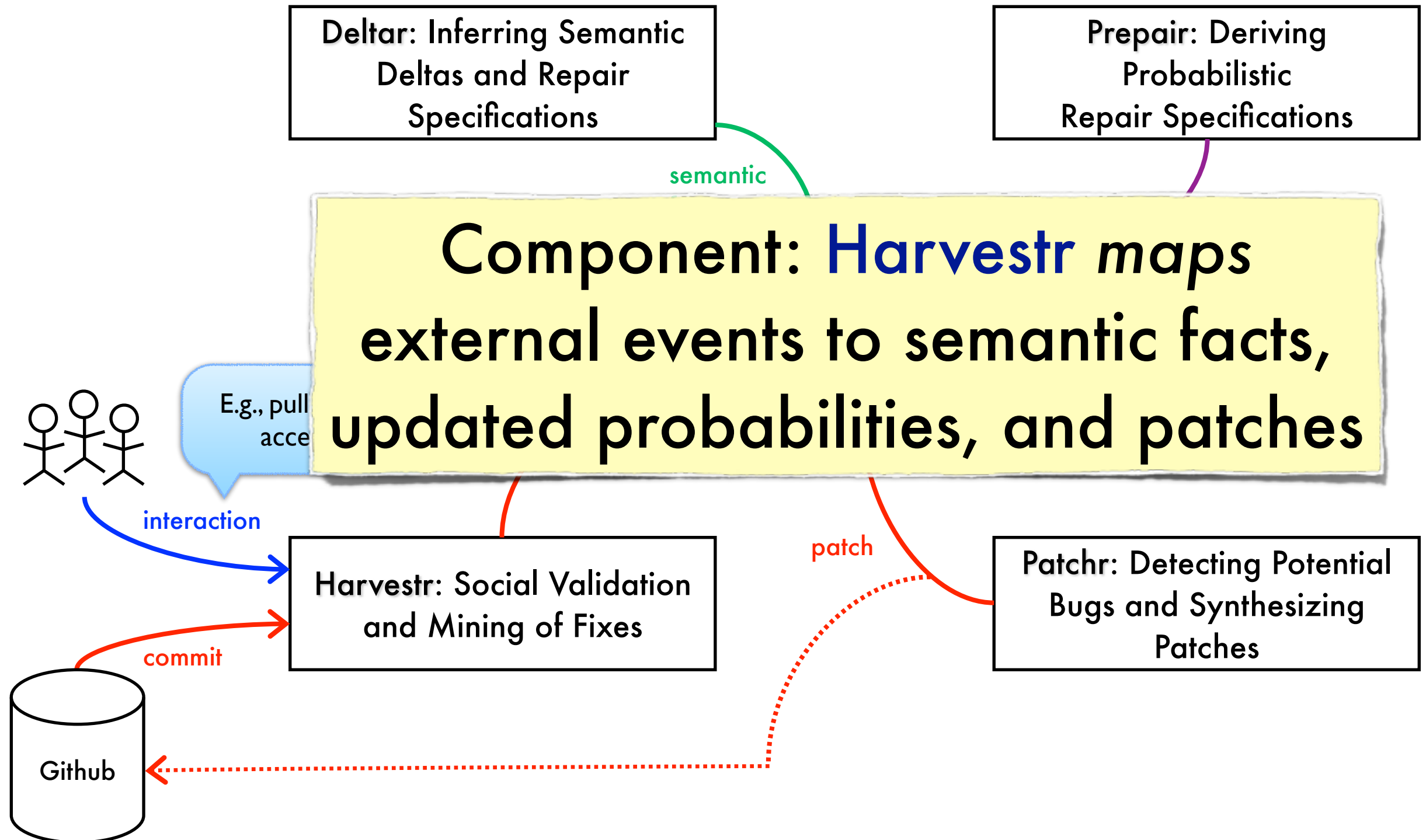
# Workflow 0: Continuous commit harvesting, buggy app patching, and social validation



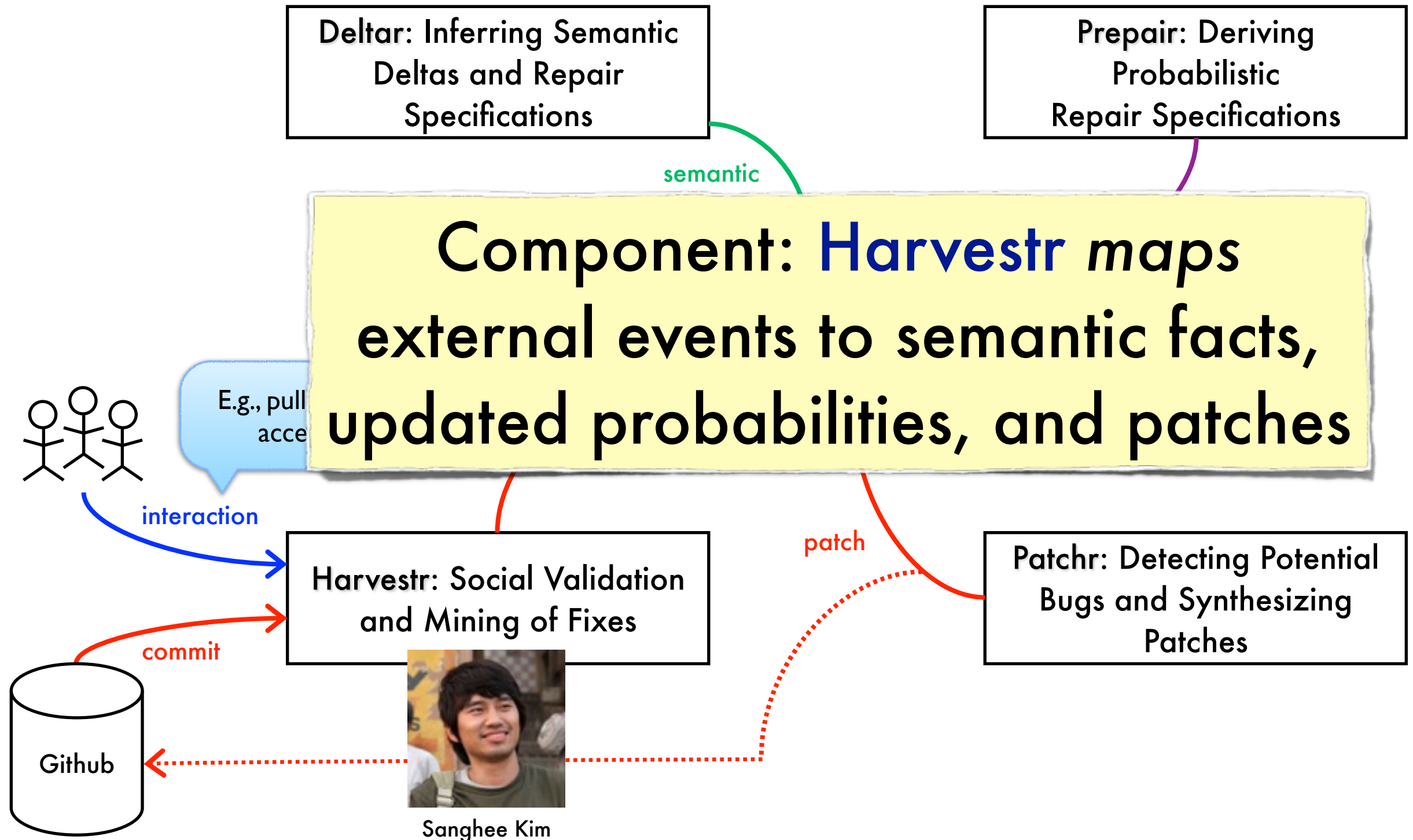
# Workflow 0: Continuous commit harvesting, buggy app patching, and social validation

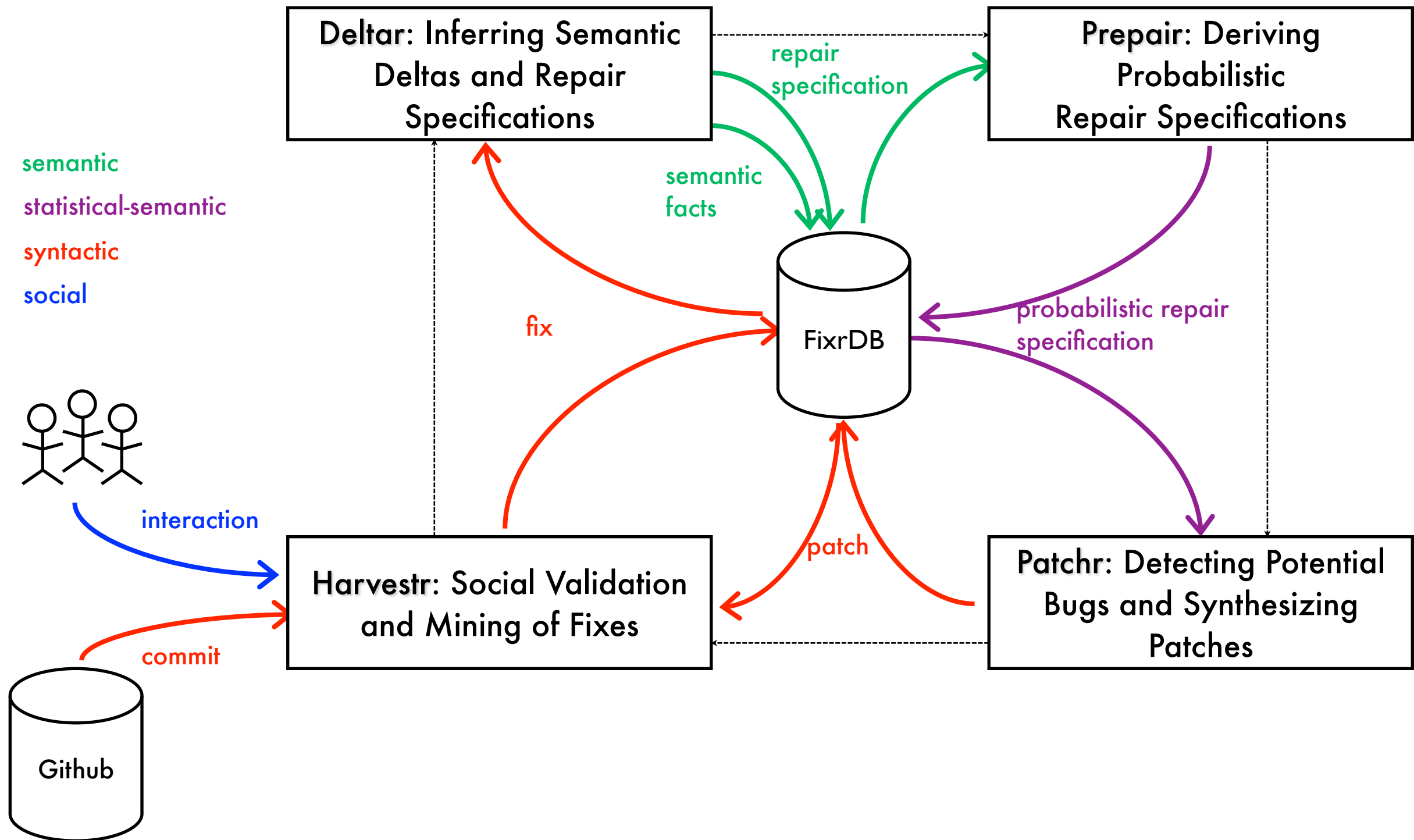


# Workflow 0: Continuous commit harvesting, buggy app patching, and social validation



# Workflow 0: Continuous commit harvesting, buggy app patching, and social validation





Bor-Yuh Evan Chang



symbolic  
program analysis

Sriram Sankaranarayanan



numerical-probabilistic  
program analysis

**Deltar: Inferring Semantic  
Deltas and Repair  
Specifications**

**Prepair: Deriving  
Probabilistic  
Repair Specifications**

semantic  
statistical-semantic  
syntactic  
social

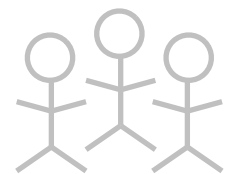
software engineering  
for big data *fix*



Ken Anderson



FixrDB



interaction

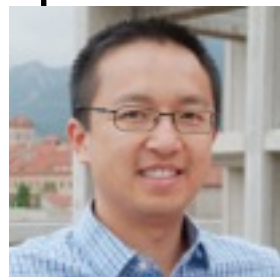
**Harvestr: Social Validation  
and Mining of Fixes**

**Patchr: Detecting Potential  
Bugs and Synthesizing  
Patches**



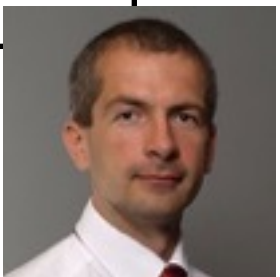
commit

user-centered  
big data analytics



Tom Yeh

program synthesis



Pavol Cerny

repair  
specification

semantic  
facts

probabilistic repair  
specification

patch



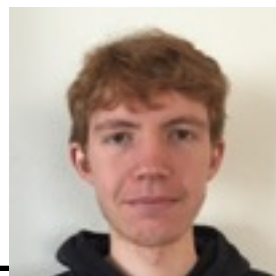
Bor-Yuh Evan Chang



Shawn Meier



Max Russek (UG)



Sriram Sankaranarayanan



**Deltar: Inferring Semantic Deltas and Repair Specifications**

**Prepair: Deriving Probabilistic Repair Specifications**

semantic  
statistical-semantic  
syntactic  
social

semantic facts

FixrDB

probabilistic repair specification

patch

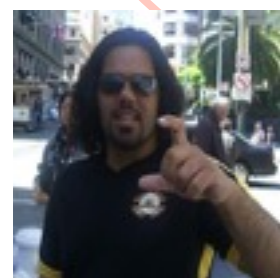
**Harvestr: Social Validation and Mining of Fixes**

**Patchr: Detecting Potential Bugs and Synthesizing Patches**



interaction

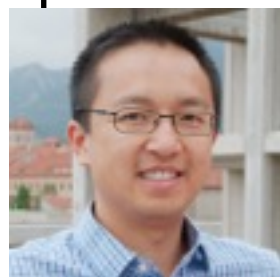
commit



Mazin Hakeem



Ken Anderson



Tom Yeh



Sanghee Kim



Brennan McConnell (UG)



Vaibhav Singh



Pavol Cerny