Always-On Programming Tools

Tom Lieber (me) — MIT CSAIL
Joel Brandt — Adobe Research
Robert C. Miller — MIT CSAIL

Cars Provide Feedback

- Procedure: turn key, step on pedal
- Output: car moves forward

Software Car Feedback?

```
Car.prototype = {
    ignition: function () { /* ... */ },
    rumble: function () { /* ... */ },
    accelerate: function () { /* ... */ },
    brake: function () { /* ... */ },
    honk: function () { /* ... */ },
    steer: function () { /* ... */ },
};
```

On-Demand = Hidden

Code

Internal State

on-demand
with debuggers

Output

Continuous feedback prepares us for trouble

```
Car.prototype = {
    ignition: function () { /* ... */ },
    rumble: function () { /* ... */ },
    accelerate: function () { /* ... */ },
    brake: function () { /* ... */ },
    honk: function () { /* ... */ },
    steer: function () { /* ... */ },
};
```

Always-On Interfaces

Code

integrated
with

Output

Research Direction

- Are “always-on” interfaces helpful to programmers?
- If so, how do they help people?
- How do we design and implement always-on interfaces well?

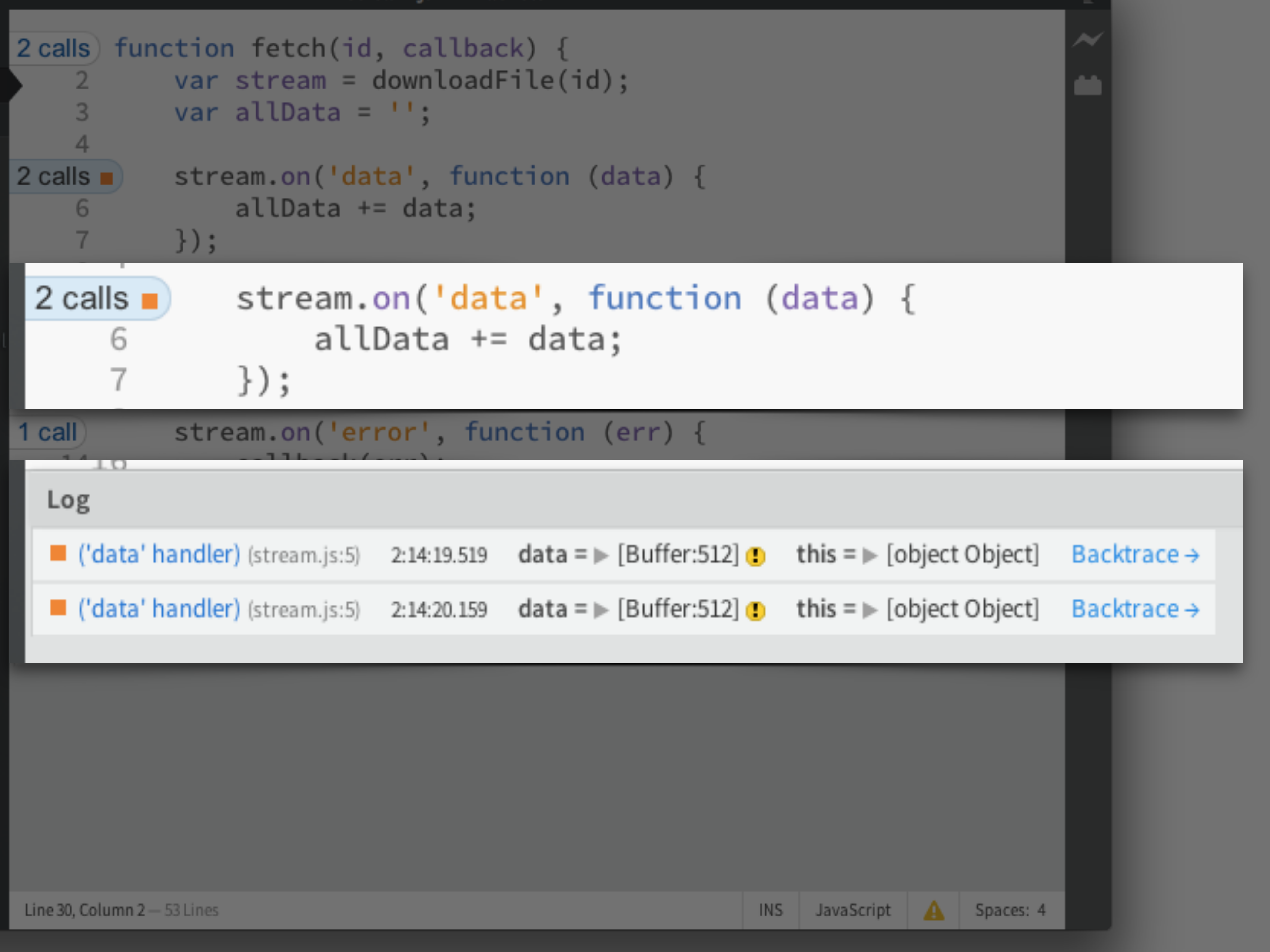
Theseus Design Goals

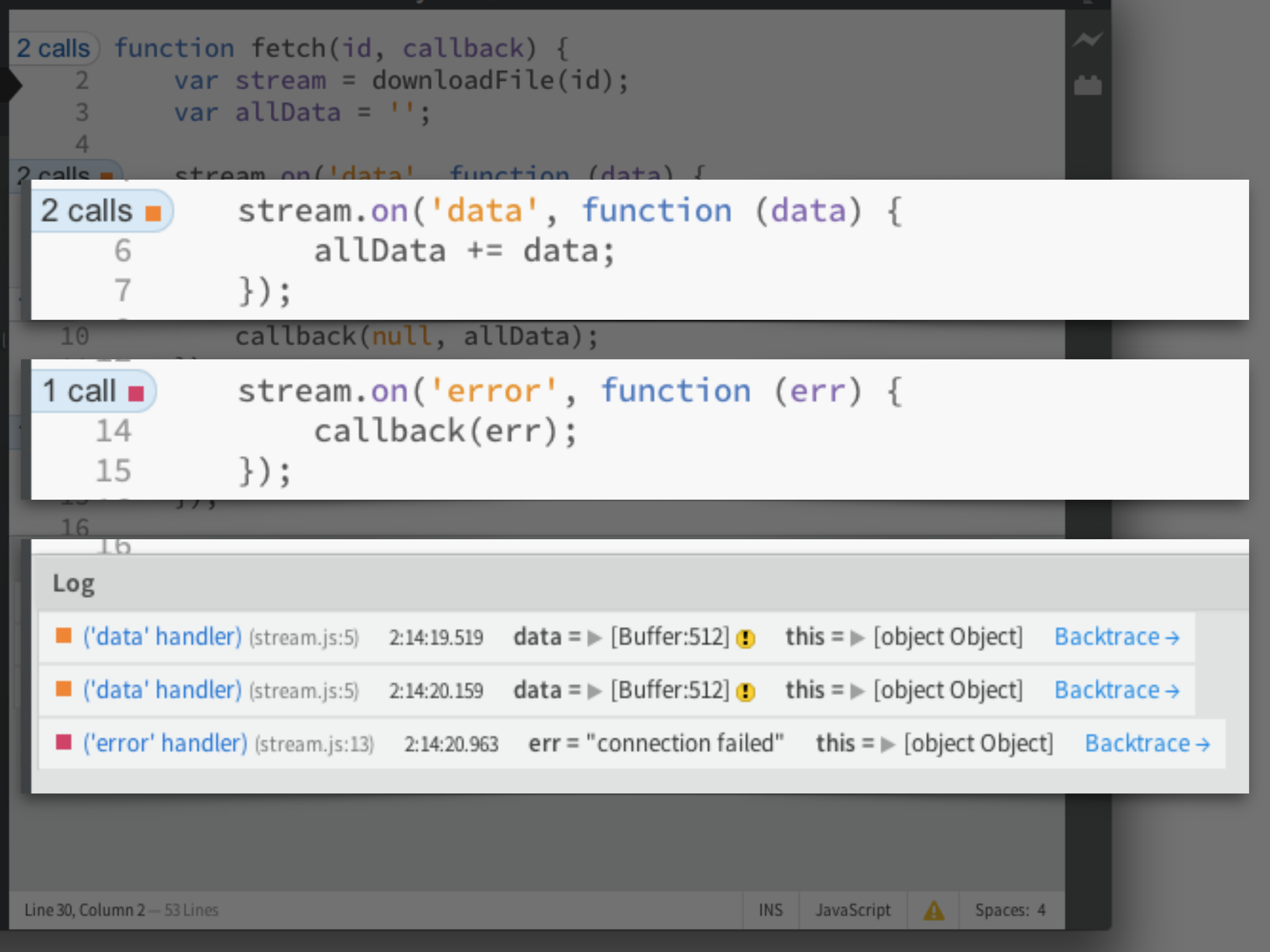
- Answer reachability questions
  (LaToza, Myers 2010)
- Low threshold, high ceiling
  - Power of breakpoints, ease of logging

```
function fetch(id, callback) {
    var stream = downloadFile(id);
    var allData = '';

    stream.on('data', function (data) {
        allData += data;
    });

    stream.on('end', function () {
        callback(null, allData);
    });

    stream.on('error', function (err) {
        callback(err);
    });

    return stream;
}
```

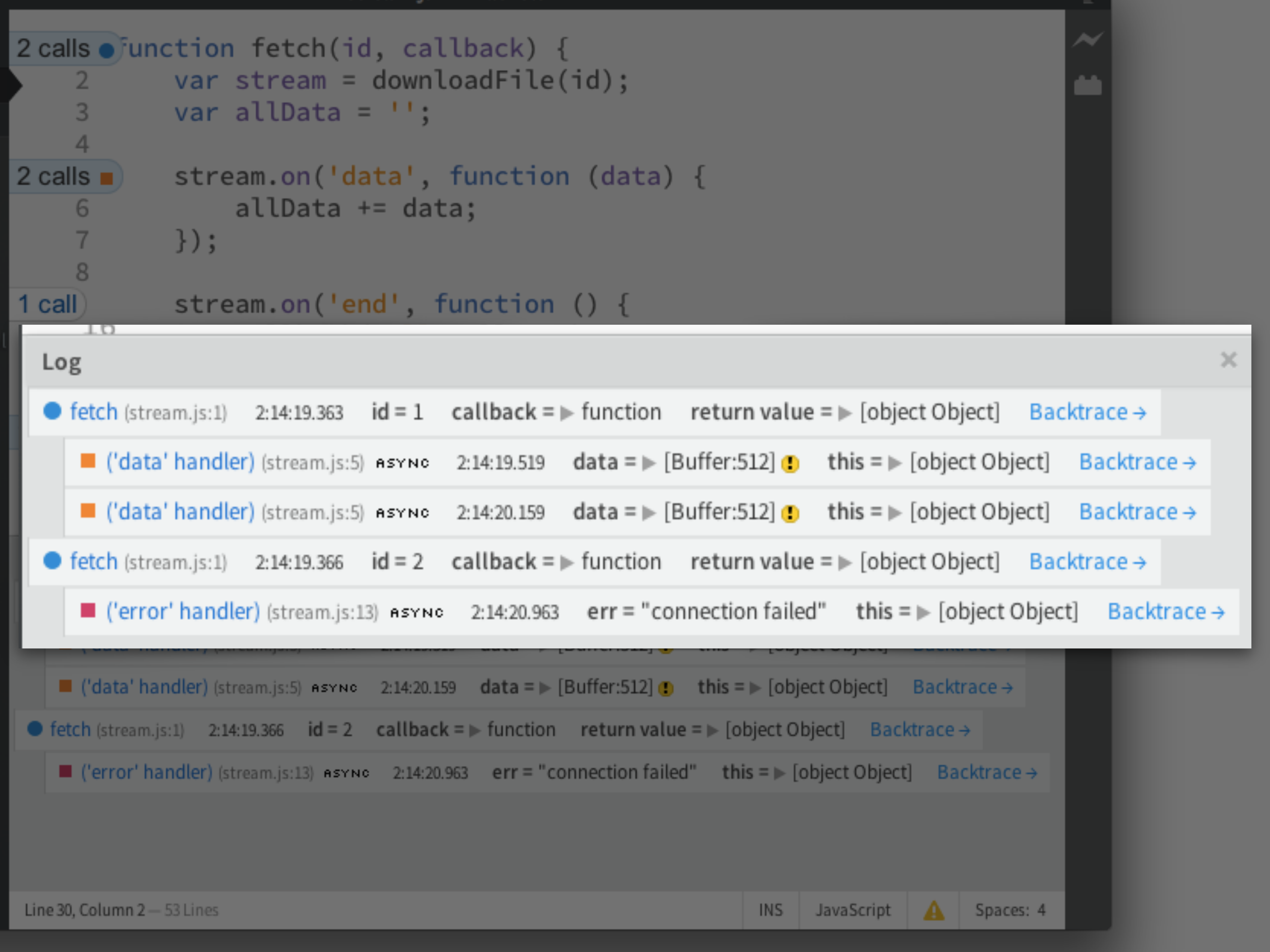
```javascript
function fetch(id, callback) {
    var stream = downloadFile(id);
    var allData = '';

    stream.on('data', function (data) {
        allData += data;
    });

    stream.on('end', function () {
        callback(null, allData);
    });

    stream.on('error', function (err) {
        callback(err);
    });

    return stream;
}
```

Line 30, Column 2 — 53 Lines | INS | JavaScript | Spaces: 4


2 calls function fetch(id, callback) {
2 var stream = downloadFile(id);
3 var allData = '';
4
2 calls stream.on('data', function (data) {
6 allData += data;
7 });
2 calls stream.on('data', function (data) {
6 allData += data;
7 });
1 call stream.on('error', function (err) {
Log
('data' handler) (stream.js:5) 2:14:19.519 data = [Buffer:512] this = [object Object] Backtrace →
('data' handler) (stream.js:5) 2:14:20.159 data = [Buffer:512] this = [object Object] Backtrace →
Line 30, Column 2 — 53 Lines
INS
JavaScript
Spaces: 4



2 calls function fetch(id, callback) {
2 var stream = downloadFile(id);
3 var allData = '';
4
2 calls stream.on('data', function (data) {
6 allData += data;
7 });
10 callback(null, allData);
1 call stream.on('error', function (err) {
14 callback(err);
15 });
16
Log
('data' handler) (stream.js:5) 2:14:19.519 data = [Buffer:512] this = [object Object] Backtrace →
('data' handler) (stream.js:5) 2:14:20.159 data = [Buffer:512] this = [object Object] Backtrace →
('error' handler) (stream.js:13) 2:14:20.963 err = "connection failed" this = [object Object] Backtrace →
Line 30, Column 2 — 53 Lines
INS
JavaScript
Spaces: 4


2 calls function fetch(id, callback) {
2 var stream = downloadFile(id);
3 var allData = '';
4
2 calls stream.on('data', function (data) {
6 allData += data;
7 });
8
1 call stream.on('end', function () {
Log
fetch (stream.js:1) 2:14:19.363 id = 1 callback = ▶ function return value = ▶ [object Object] Backtrace →
('data' handler) (stream.js:5) ASYNC 2:14:19.519 data = ▶ [Buffer:512] this = ▶ [object Object] Backtrace →
('data' handler) (stream.js:5) ASYNC 2:14:20.159 data = ▶ [Buffer:512] this = ▶ [object Object] Backtrace →
fetch (stream.js:1) 2:14:19.366 id = 2 callback = ▶ function return value = ▶ [object Object] Backtrace →
('error' handler) (stream.js:13) ASYNC 2:14:20.963 err = "connection failed" this = ▶ [object Object] Backtrace →
Line 30, Column 2 — 53 Lines
INS
JavaScript
Spaces: 4

Design Principles

Think about bandwidth

```
2 calls    stream.on('end', function () {
               callback(null, allData);
           });

0 calls    stream.on('error', function (err) {
               callback(err);
           });
```

Think about efficiency

- Can be used to open the full tool using the user's current context
- Might answer their questions without them having to click anything
- Might clue programmer into problems that are otherwise invisible

How does programmer behavior change with always-on tools?

Evaluation 1 Method

- 7 MIT grad student participants
  - 20-39 years old, male
- Two 20-minute tasks (A, B)
  - A: Fix bug in 2,000-line, 8-file JavaScript page
  - B: Calculate recursive file size with async API
- Three 5-minute tasks (C, D, E)

Evaluation 1 Results: Uses of Call Counts

Three uses
of call counts

Evaluation 1 Results: Use #1 of Call Counts

Notice incorrect call count changes

"I get 2 mouse up actions [every time I click]. Huh."

Evaluation 1 Results: Use #2 of Call Counts

Compare two call counts

"I'd expect the call counts to be the same for both of them, but they're not."

Evaluation 1 Results: Use #3 of Call Counts

Compare call counts to other data

17 files in directory, 17 calls to function

Evaluation 1 Results: Use of Call Counts?

Unclear whether call counts helped find initial focus points

- One user felt strongly that Theseus was useful for skimming, another the opposite

Evaluation 1 Results

With interactive code, programmers arranged windows to see code and app side-by-side

2/3 of the participants who started with task A (complicated web page) all used side-by-side technique on small tasks C and D

Evaluation 2 Method

- 9 participants, professional developers, male
- Used Theseus for a week in daily work
- Interview:
  - How they used Theseus during the week
  - Work on task A from the previous study

Programmers didn’t use Theseus until they got stuck

- Start by reading to “familiarize myself with where all the code is”
  - “I try to stay out of the debugger as much as possible because it’s a time suck.”
- But some did use it as part of finding initial focus points*

* Sillito. Asking and Answering Questions During a Programming Change Task. Thesis, 2006.

Call counts: weak, but sufficient evidence

- “So this was called 7 times. ... Seems about right. I didn’t draw that many things.”
- “This was called a bunch, 319 times... maybe they’re simulating dragging.”

Evaluation 2 Results

Programmers want more always-on displays

- Time spent in every function
- File-level counterpart for function call counts
- State changes on individual lines

Future Work

- Theseus: programmers occasionally had to memorize call counts
- Always-on interfaces: more diverse participant populations

Take-Aways

- Always-on displays enable interesting new types of debugging interactions that deserve exploration

- When creating a programming tool, consider an always-on component

- Call counts are surprisingly useful… what else?

Try It Yourself

- http://brackets.io/
  - File > Extension Manager, install “Theseus”
- Source: https://github.com/adobe-research/theseus
- Available since February 11, 2013
  - Installed >= 2,500 times as of December
  - 57 bug reports & feature requests as of today

Do It Yourself

- https://github.com/adobe-research/fondue
  - eval(fondue.instrument(src));
  - Real-time information: functions called, parameter values, etc
- tom@alltom.com

Thanks!

- Get it: http://brackets.io/ then install “Theseus”
- Fork it: https://github.com/adobe-research/theseus
- Make it: https://github.com/adobe-research/fondue