SOFTWARE ANALYSIS AND TESTING

Don't Call Us, We'll Call You:

Characterizing Callbacks in JavaScript

Keheliya Gallaba, Ali Mesbah, Ivan Beschastnikh
University of British Columbia

Why JavaScript?

On the client

@WalmartLabs

ebay

IBM

node

JS

Microsoft

LinkedIn

PayPal

Joyent

redhat

CYLON.JS

Espruino

TESSEL

Why JavaScript?

On the client

@WalmartLabs
Social + Mobile + Retail
ebay
IBM
node
TM
JS
Microsoft
LinkedIn
PayPal
Joyent

On the server

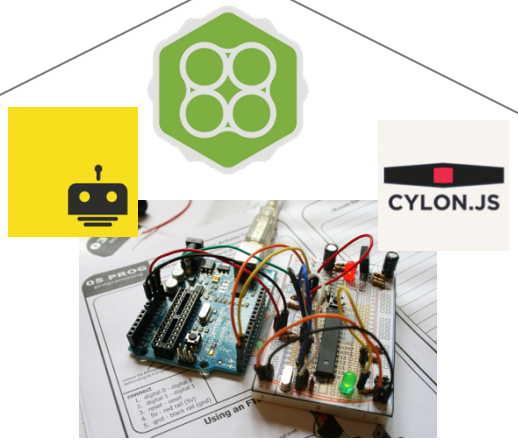
CYLON.JS
Espruino
TESSEL

Why JavaScript?

On the client

@WalmartLabs
Social + Mobile + Retail
ebay
IBM
node
TM
Microsoft
LinkedIn
PayPal
Joyent

On the server



Espruino
TESSEL

Even in hardware!

Why JavaScript?

JavaScript has become the modern lingua franca!

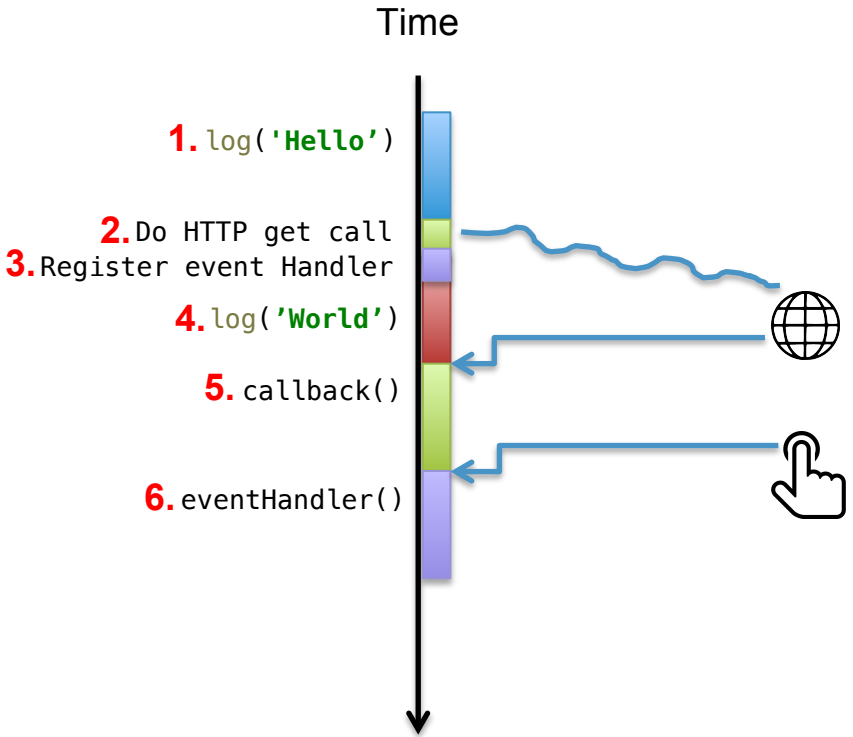
On the client

On the server

Even in hardware!

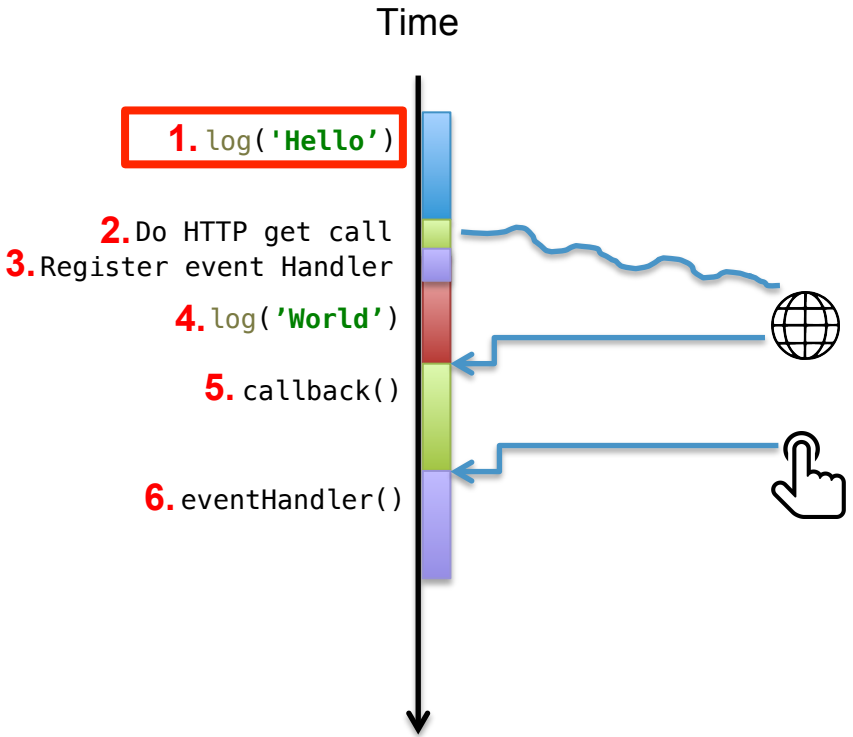
JavaScript Event-driven Model

```
console.log('Hello');

$.get('data.json', callback);

$("button").click(eventHandler);

console.log('World');
```



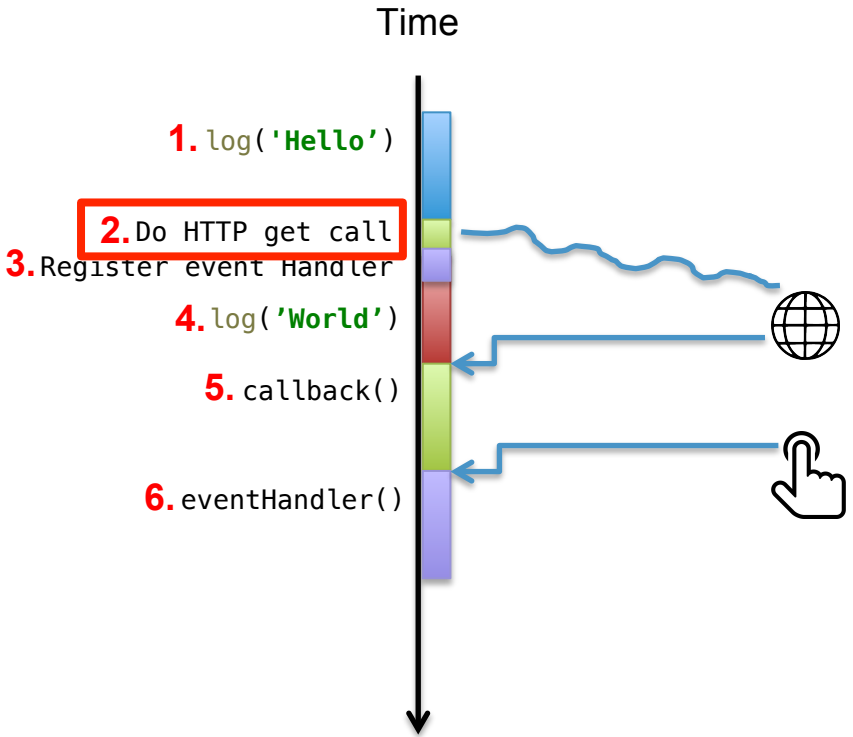
JavaScript Event-driven Model

```
console.log('Hello');

$.get('data.json', callback);

$("button").click(eventHandler);

console.log('World');
```



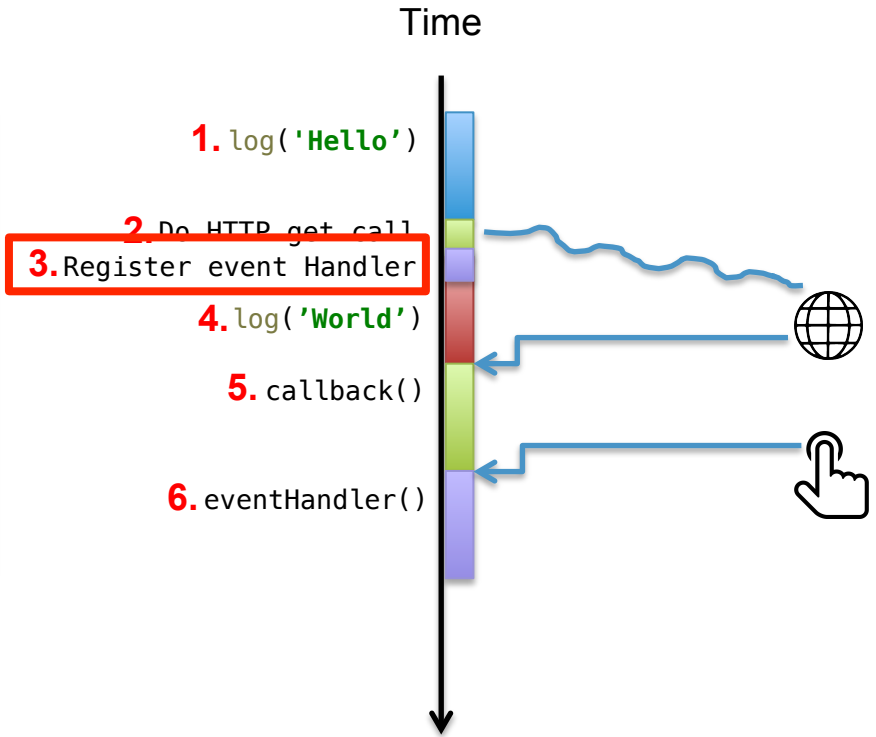
JavaScript Event-driven Model

```
console.log('Hello');

$.get('data.json', callback);

$("button").click(eventHandler);

console.log('World');
```



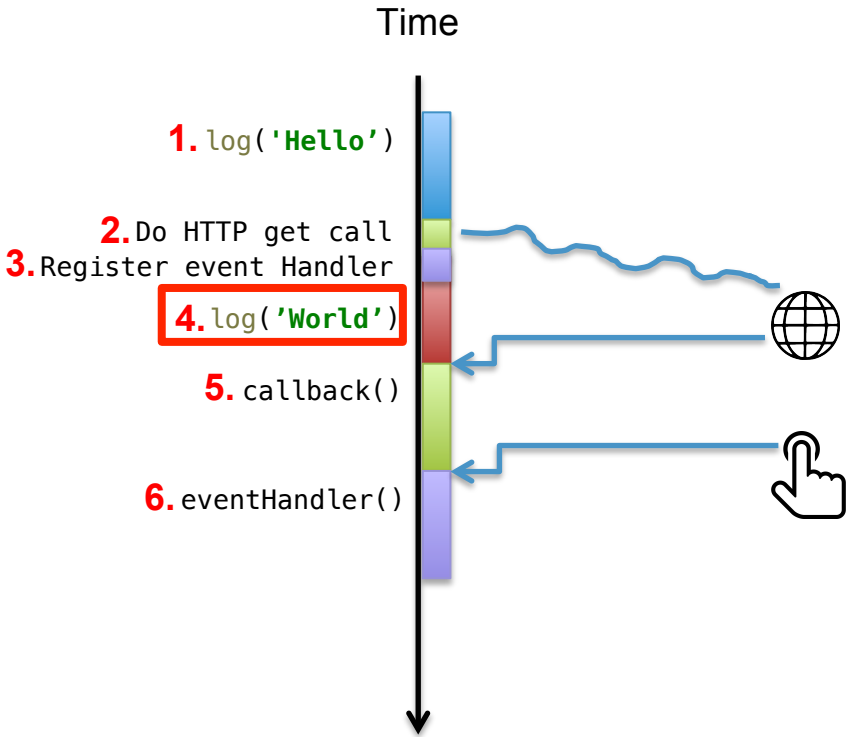
JavaScript Event-driven Model

```
console.log('Hello');

$.get('data.json', callback);

$("button").click(eventHandler);

console.log('World');
```



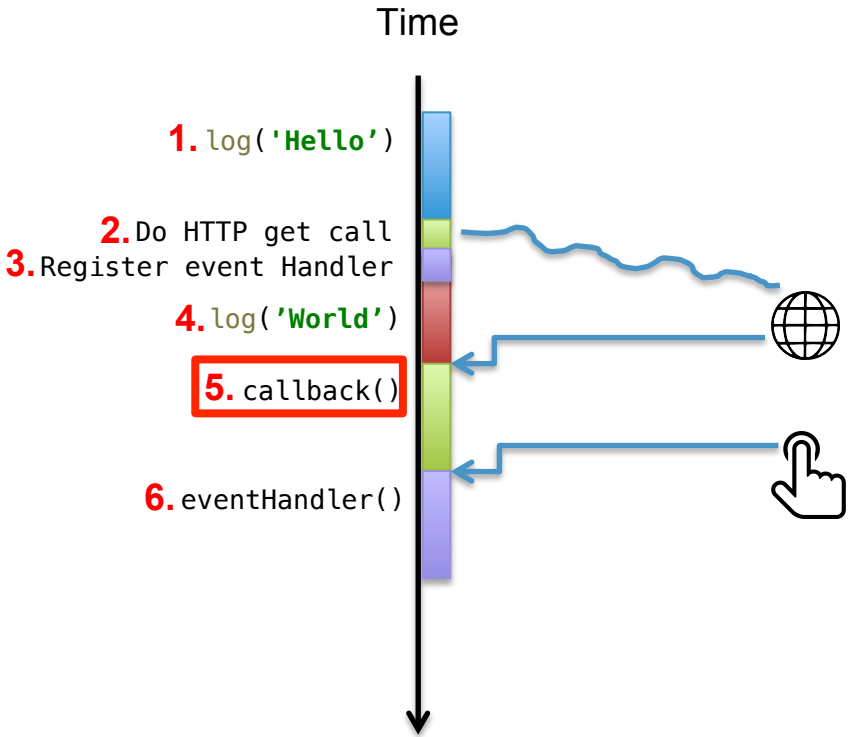
JavaScript Event-driven Model

```
console.log('Hello');

$.get('data.json', callback);

$("button").click(eventHandler);

console.log('World');
```



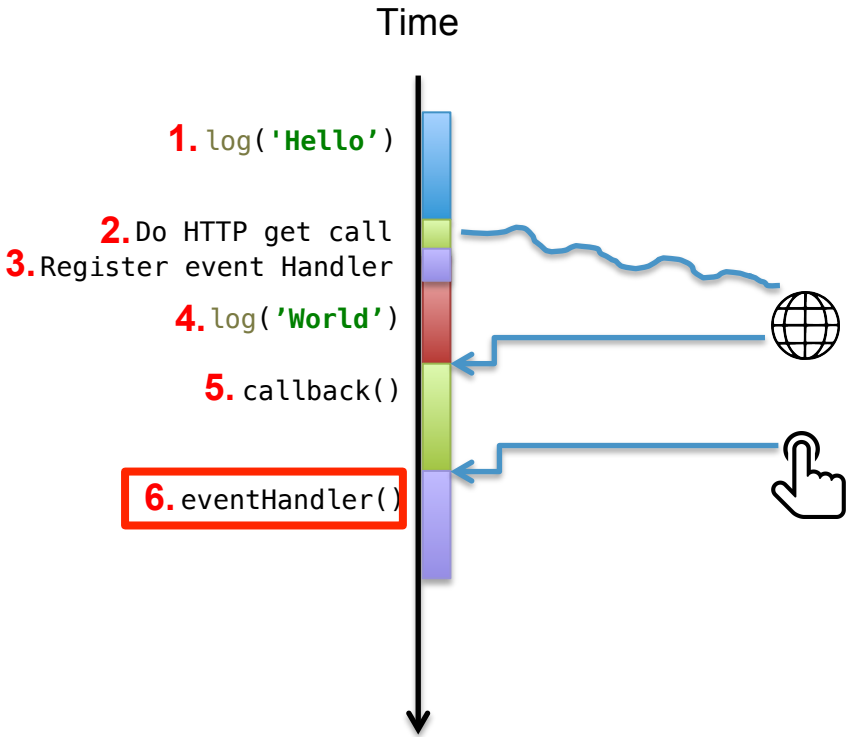
JavaScript Event-driven Model

```
console.log('Hello');

$.get('data.json', callback);

$("button").click(eventHandler);

console.log('World');
```



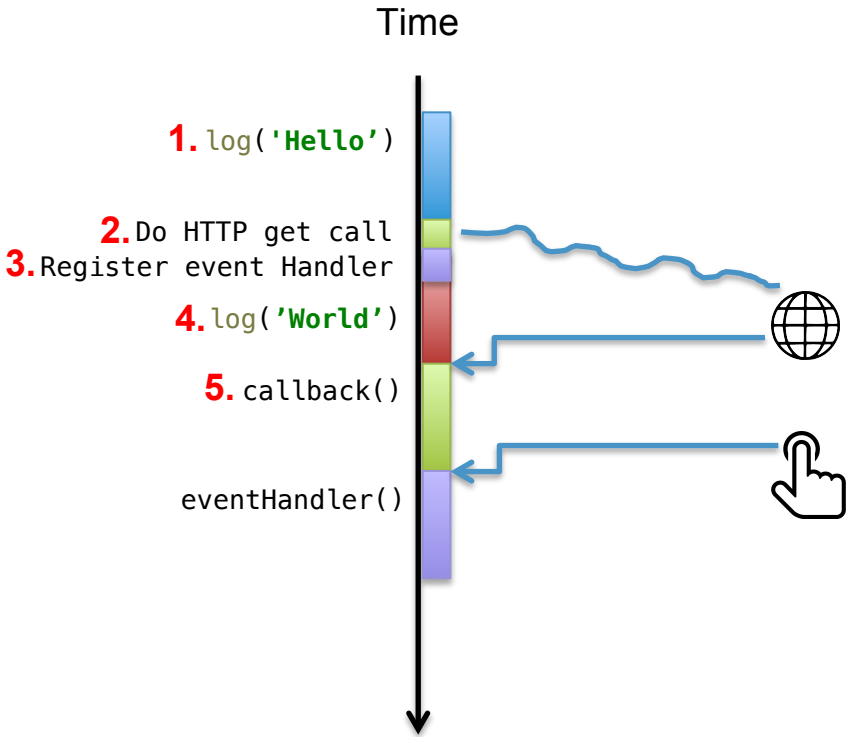
JavaScript Event-driven Model

```
console.log('Hello');

$.get('data.json', callback);

$("button").click(eventHandler);

console.log('World');
```



JavaScript Event-driven Model

```
console.log('Hello');

$.get('data.json', callback);

$("button").click(eventHandler);

console.log('World');
```



Why Callbacks?

```
$("button").click(buttonHandler);

function buttonHandler(event){
    alert("Button Clicked.");
}
```

Used For:

- HTTP Request/Response
- File I/O in Node.js
- Mouse click/drag events in the browser

Why Callbacks?

```
$("button").click(buttonHandler);

function buttonHandler(event){
    alert("Button Clicked.");
}
```

Used For:

- HTTP Request/Response
- File I/O in Node.js
- Mouse click/drag events in the browser

You cannot write an interesting program in JavaScript without using **callbacks!**

Motivation

- Callbacks are important in all non-trivial JS programs.
- But, how do developers use callbacks?
- No prior research on this topic.

Goal of this work

Outline

Motivation ✔
Methodology & Overview
Characterizing Problems
- Anonymous Callbacks
- Asynchronous Callbacks
- Nested Callbacks

Characterizing Solutions
- Error-first Protocol
- Async.js
- Promises

Conclusion

Outline

- Motivation ✔
- ► Methodology & Overview
- Characterizing Problems
  - Anonymous Callbacks
  - Asynchronous Callbacks
  - Nested Callbacks
- Characterizing Solutions
  - Error-first Protocol
  - Async.js
  - Promises
- Conclusion

Subject Systems

138 popular **open source** JavaScript subject systems,

from **6** distinct categories,

with both **Client-side** & **Server-side** code.

Category	Subject systems	Client side	Server side	Total files	Total LOC
NPM Modules	86		✓	4,983	1,228,271
Web Apps.	16	✓	✓	1,779	494,621
Game Engines	16	✓	✓	1,740	1,726,122
Frameworks	8	✓		2,374	711,172
DataViz Libs.	6	✓		3,731	958,983
Games	6	✓		347	119,279
Total	**138**	✓	✓	**14,954**	**5,238,448**

What is a callback?

A **callback** is a function that is passed as an argument to another function, which is expected to invoke it either *immediately* or at some point *in the future*.

```
function getInput (options, callback)
   allUserData.push (options);
   callback (options);
}

var logStuff = function () { ... }
getInput ({name:"Rich", speciality:"JavaScript"}, logStuff);
```

What is a callback?

Accepts a callback

```
function getInput (options, callback) {
   allUserData.push (options);
   callback (options);
}

var logStuff = function () { ... }
getInput ({name:"Rich", speciality:"JavaScript"}, logStuff);
```

Invokes the callback

Passing a callback as an argument to a function

A **callback** is a function that is passed as an argument to another function, which is expected to invoke it either *immediately* or at some point *in the future*.

Detecting callbacks: An example

```
1 function getRecord(id, callback) {
2   http.get('http://foo/' + id, function (err, doc) {
3     if (err) {
4       return callback(err);
5     }
6       return callback(null, doc);
7   });
8 }

10 var logStuff = function () { ... }
11 getRecord('007', logStuff);
```

`getRecord()` accepts `logStuff()` as a callback because there exist a path...

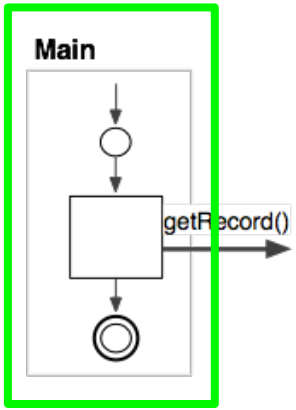
`getRecord(cb)` ➡ `http.get()` ➡ `Anonymous()` ➡ `logStuff()`

Detecting callbacks: An example

```
1 function getRecord(id, callback) {
2   http.get('http://foo/' + id, function (err, doc) {
3     if (err) {
4       return callback(err);
5     }
6       return callback(null, doc);
7   });
8 }

10 var logStuff = function () { ... }
11 getRecord('007', logStuff);
```

`getRecord()` accepts `logStuff()` as a callback because there exist a path...

`getRecord(cb)` ➡ `http.get()` ➡ `Anonymous()` ➡ `logStuff()`

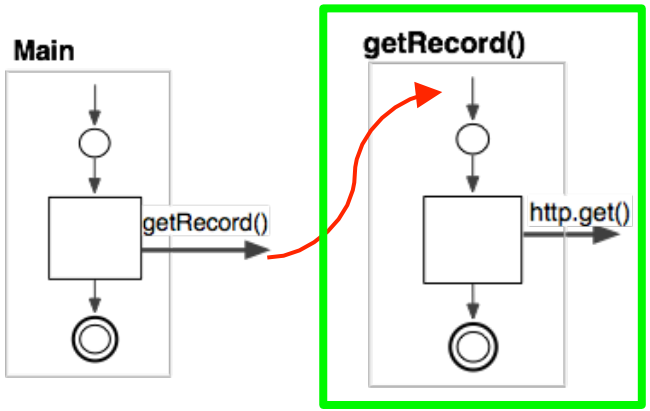


Detecting callbacks: An example

```
1 function getRecord(id, callback) {
2   http.get('http://foo/' + id, function (err, doc) {
3     if (err) {
4       return callback(err);
5     }
6       return callback(null, doc);
7   });
8 }

10 var logStuff = function () { ... }
11 getRecord('007', logStuff);
```

`getRecord()` accepts `logStuff()` as a callback because there exist a path...

`getRecord(cb)` ➡ `http.get()` ➡ `Anonymous()` ➡ `logStuff()`

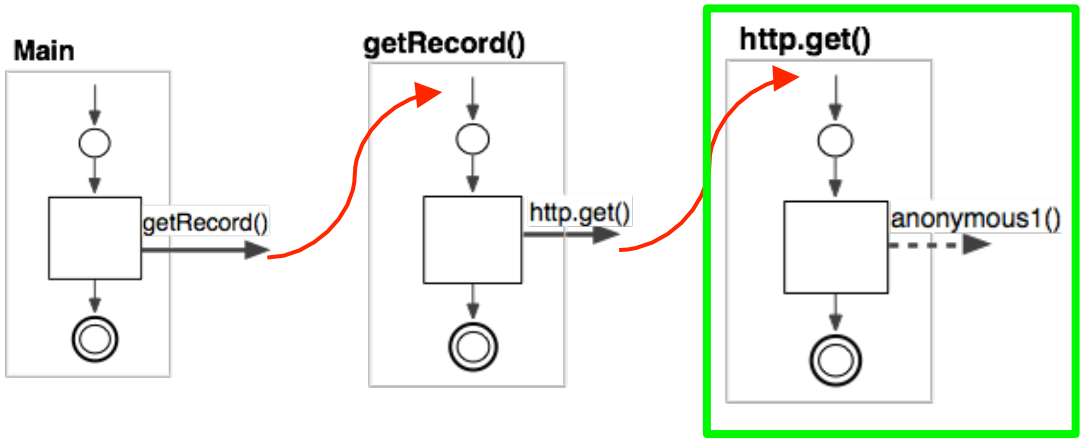


Detecting callbacks: An example

```
1 function getRecord(id, callback) {
2   http.get('http://foo/' + id, function (err, doc) {
3     if (err) {
4       return callback(err);
5     }
6       return callback(null, doc);
7   });
8 }

10 var logStuff = function () { ... }
11 getRecord('007', logStuff);
```

`getRecord()` accepts `logStuff()` as a callback because there exist a path...

`getRecord(cb)` ➡ `http.get()` ➡ `Anonymous()` ➡ `logStuff()`

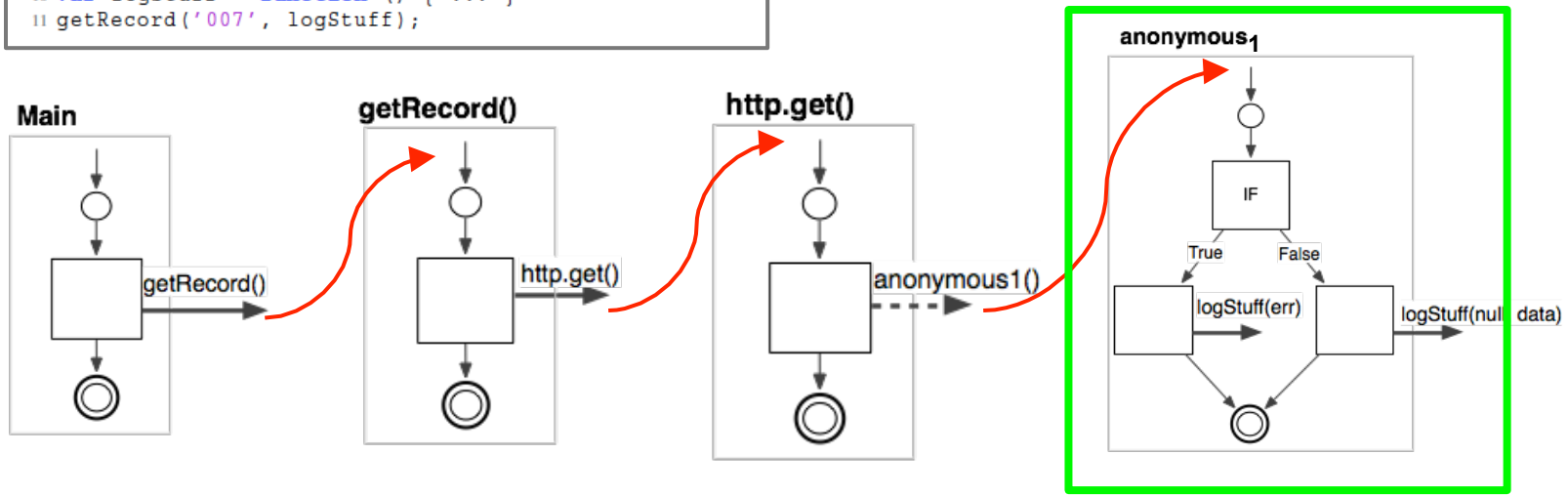


Detecting callbacks: An example

```
1 function getRecord(id, callback) {
2   http.get('http://foo/' + id, function (err, doc) {
3     if (err) {
4       return callback(err);
5     }
6       return callback(null, doc);
7   });
8 }

10 var logStuff = function () { ... }
11 getRecord('007', logStuff);
```

`getRecord()` accepts `logStuff()` as a callback because there exist a path...

`getRecord(cb)` ➡ `http.get()` ➡ `Anonymous()` ➡ `logStuff()`

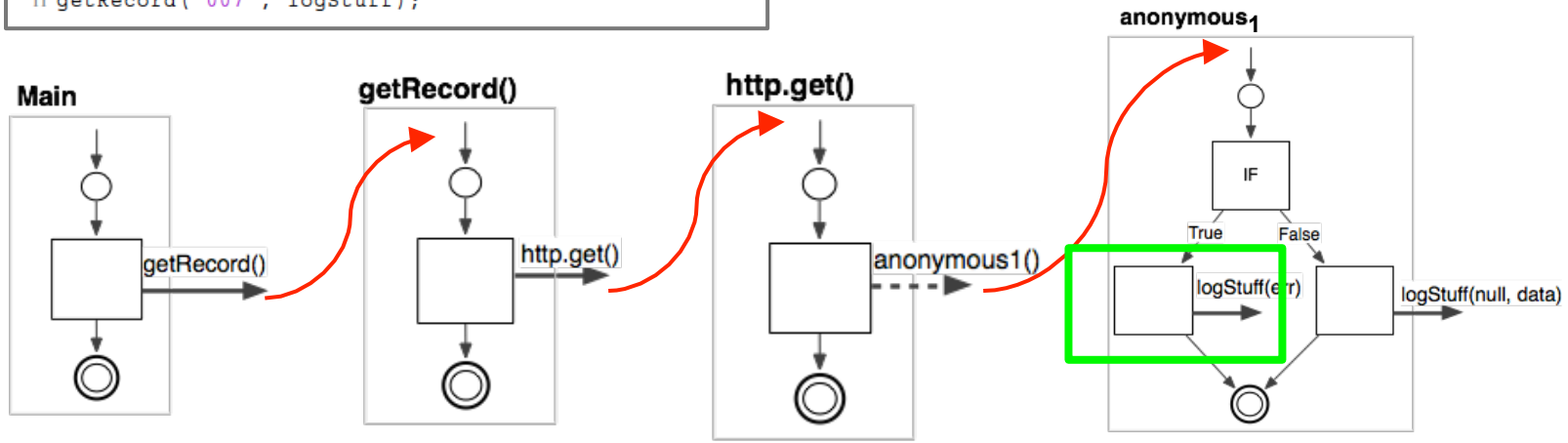


Detecting callbacks: An example

```
1 function getRecord(id, callback) {
2   http.get('http://foo/' + id, function (err, doc) {
3     if (err) {
4       return callback(err);
5     }
6       return callback(null, doc);
7   });
8 }

10 var logStuff = function () { ... }
11 getRecord('007', logStuff);
```

`getRecord()` accepts `logStuff()` as a callback because there exist a path...

`getRecord(cb)` ➡ `http.get()` ➡ `Anonymous()` ➡ `logStuff()`

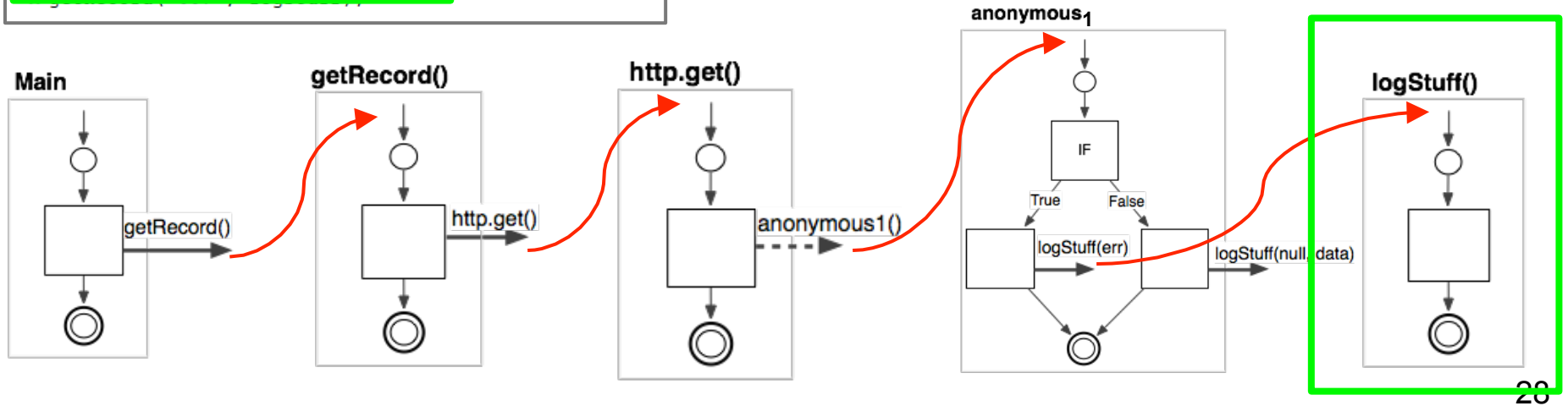


Detecting callbacks: An example

```
1 function getRecord(id, callback) {
2   http.get('http://foo/' + id, function (err, doc) {
3     if (err) {
4       return callback(err);
5     }
6       return callback(null, doc);
7   });
8 }

10 var logStuff = function () { ... }
```

`getRecord()` accepts `logStuff()` as a callback because there exist a path...

`getRecord(cb)` ➡ `http.get()` ➡ `Anonymous()` ➡ `logStuff()`

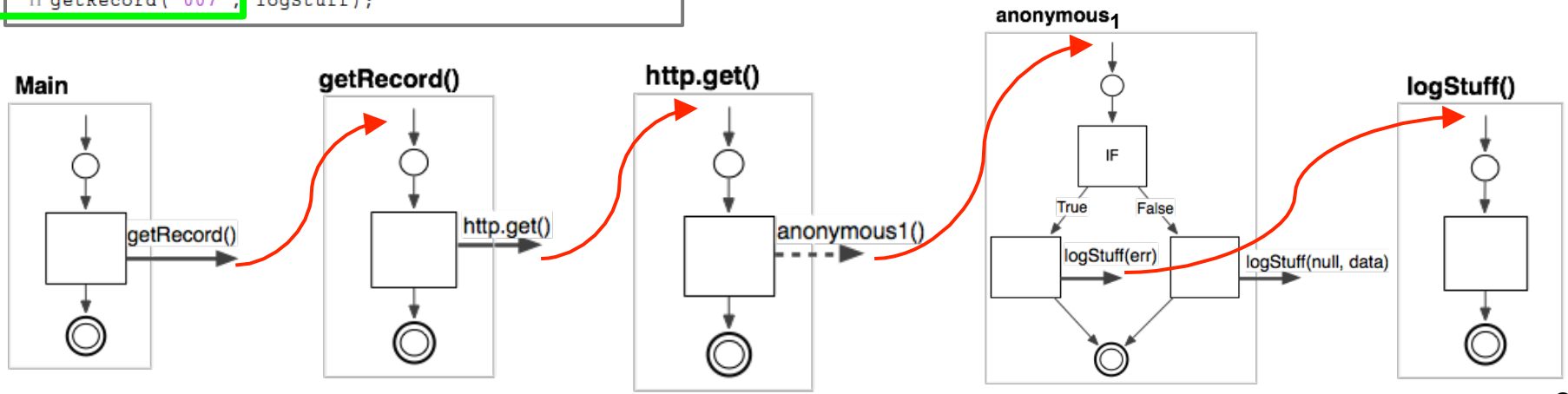


Detecting callbacks: An example

```
1  function getRecord(id, callback) {
2    http.get('http://foo/' + id, function (err, doc) {
3      if (err) {
4        return callback(err);
5      }
6        return callback(null, doc);
7    });
8  }

10 var logStuff = function () { ... }
11 getRecord('007', logStuff);
```

`getRecord()` accepts `logStuff()` as a callback because there exist a path...

`getRecord(cb)` ➡ `http.get()` ➡ `Anonymous()` ➡ `logStuff()`

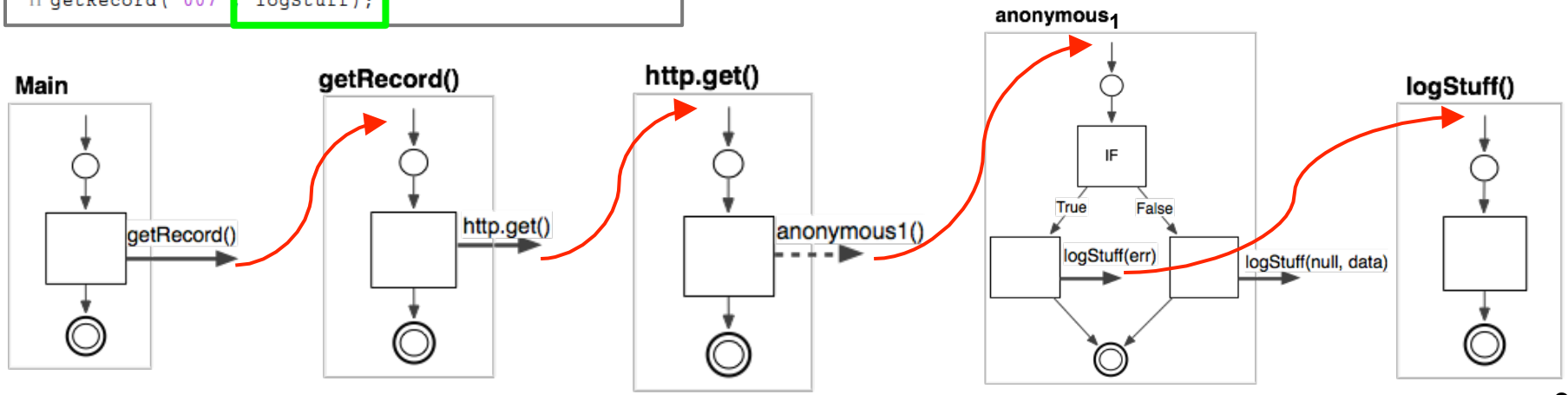


Detecting callbacks: An example

```
1 function getRecord(id, callback) {
2   http.get('http://foo/' + id, function (err, doc) {
3     if (err) {
4       return callback(err);
5     }
6       return callback(null, doc);
7   });
8 }

10 var logStuff = function () { ... }
11 getRecord('007', logStuff);
```

`getRecord()` accepts `logStuff()` as a callback because there exist a path...

`getRecord(cb)` ➡ `http.get()` ➡ `Anonymous()` ➡ `logStuff()`



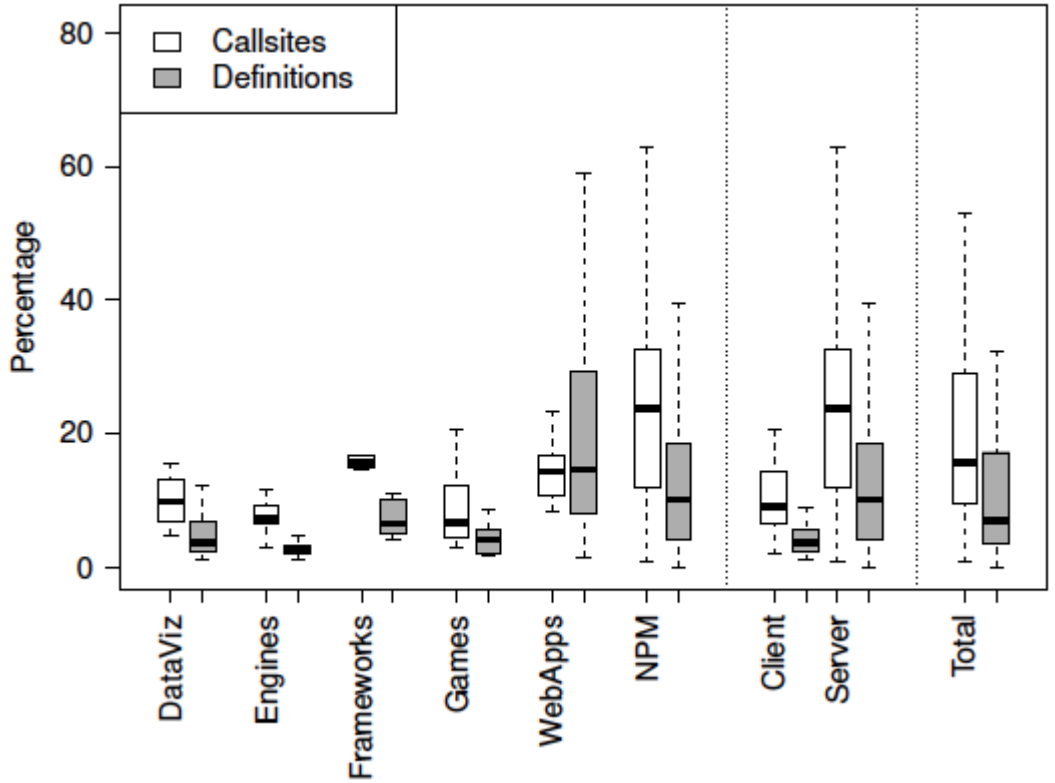
How prevalent are callbacks?

Callback-accepting function definitions

- On average, **10%** of all function definitions take callback arguments.
- They are more prevalent in server-side code **(10%)** than in client- side code **(4.5%)**.

Callback-accepting function call-sites

- **19%** of all function callsites take callback arguments.
- Callback-accepting function call-sites are more prevalent in server-side code **(24%)** than in client-side code **(9%)**.



How prevalent are callbacks?

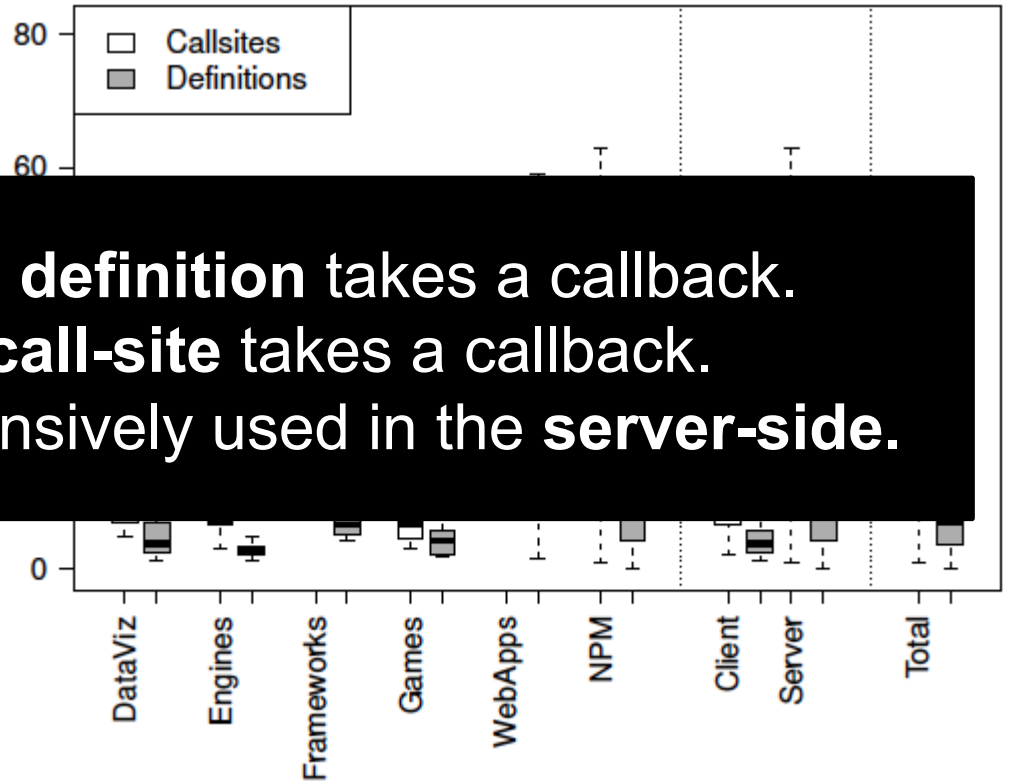
Callback-accepting function definitions

- On average, **10%** of all function definitions take callback arguments.
- They are more prevalent in

Calll

take callback arguments.

- Callback-accepting function call-sites are more prevalent in server-side code **(24%)** than in client-side code **(9%)**.



- Every 10th function **definition** takes a callback.
- Every 5th function **call-site** takes a callback.
- Callbacks are extensively used in the **server-side.**

Outline

Motivation ✔
Methodology & Overview ✔
Characterizing Problems
- Anonymous Callbacks
- Asynchronous Callbacks
- Nested Callbacks

Characterizing Solutions
- Error-first Protocol
- Async.js
- Promises

Conclusion

Outline

- Motivation ✔
- Methodology & Overview ✔
- ► Characterizing Problems
  - Anonymous Callbacks
  - Asynchronous Callbacks
  - Nested Callbacks
- Characterizing Solutions
  - Error-first Protocol
  - Async.js
  - Promises
- Conclusion

JavaScript in the wild...

```
define('admin/general/dashboard', 'semver', function(semver
    ) {
  var Admin = {};

  $('#logout-link').on('click', function() {
    $.post(RELATIVE_PATH + '/logout', function() {
      window.location.href = RELATIVE_PATH + '/';
    });
  });

  ...

  $('.restart').on('click', function() {
    bootbox.confirm('Are you sure you wish to restart
        NodeBB?', function(confirm) {
      if (confirm) {
        $(window).one('action:reconnected', function() {
          app.alert({ alert_id: 'instance_restart', });
        });

        socket.emit('admin.restart');
      }
    });
  });
  return Admin;
});
```

Anonymous callbacks

```
1  define('admin/general/dashboard', 'semver', function(semver
       ) {
2    var Admin = {};

4    $('#logout-link').on('click', function() {
5      $.post(RELATIVE_PATH + '/logout', function() {
6        window.location.href = RELATIVE_PATH + '/';
7      });
8    });

10   ...

12   $('.restart').on('click', function() {
13     bootbox.confirm('Are you sure you wish to restart
           NodeBB?', function(confirm) {
14       if (confirm) {
15         $(window).one('action:reconnected', function() {
16           app.alert({ alert_id: 'instance_restart', });
17         });

19         socket.emit('admin.restart');
20       }
21     });
22   });
23   return Admin;
24 });
```

Notorious for:
Difficulty to debug, maintain, test, or reuse

Anonymous Vs. Named Callbacks

If a function callsite is ..

- callback-accepting and
- has an anonymous function expression as an argument

..it is an instance of an anonymous callback.

```
1 function getRecord(id, callback) {
2   http.get('http://foo/' + id, function (err, doc) {
3     if (err) {
4       return callback(err);
5     }
6       return callback(null, doc);
7   });
8 }

10 var logStuff = function () { ... }
11 getRecord('007', logStuff);
```

Anonymous Vs. Named Callbacks

If a function callsite is ..

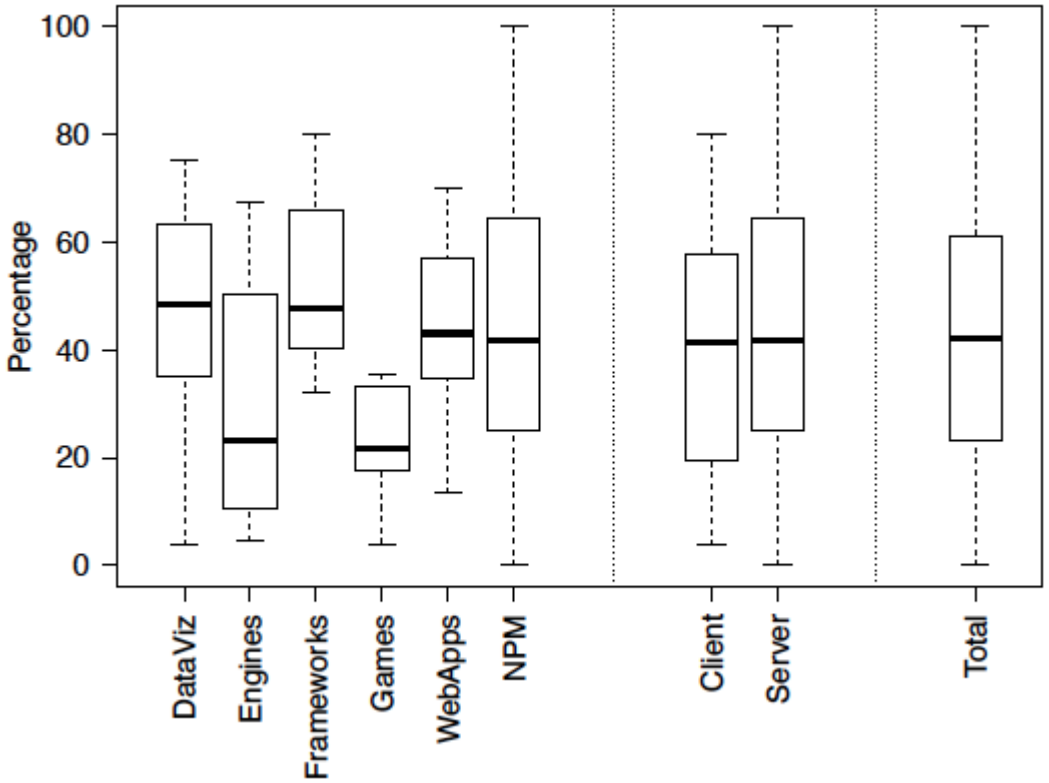
- callback-accepting and
- has an anonymous function expression as an argument

..it is an instance of an anonymous callback.

```
1 function getRecord(id, callback) {
2   http.get('http://foo/' + id, function (err, doc) {
3     if (err) {
4       return callback(err);
5     }
6       return callback(null, doc);
7   });
8 }

10 var logStuff = function () { ... }
11 getRecord('007', logStuff);
```

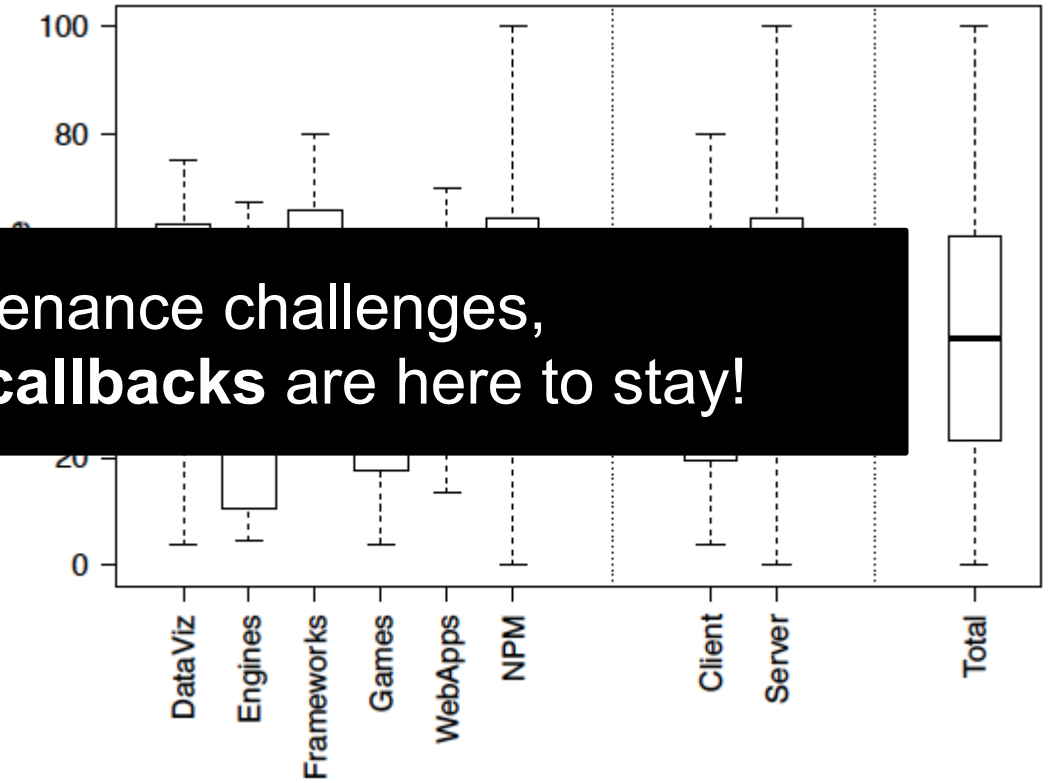
Anonymous Vs. Named Callbacks - Results

- **43%** of all callback-accepting function callsites are invoked with at least one anonymous callback.
- There is little difference between client-side and server-side code in how they use anonymous callbacks.



Anonymous Vs. Named Callbacks - Results

- **43%** of all callback-accepting fu
  wi
  ca
- Th
  between client-side and server-side code in how they use anonymous callbacks.



In spite of maintenance challenges,
Anonymous callbacks are here to stay!

Asynchronous callbacks

```
1  define('admin/general/dashboard', 'semver', function(semver
       ) {
2    var Admin = {};

4    $('#logout-link').on('click', function() {
5      $.post(RELATIVE_PATH + '/logout', function() {
6        window.location.href = RELATIVE_PATH + '/';
7      });
8    });

10   ...

12   $('.restart').on('click', function() {
13     bootbox.confirm('Are you sure you wish to restart
           NodeBB?', function(confirm) {
14       if (confirm) {
15         $(window).one('action:reconnected', function() {
16           app.alert({ alert_id: 'instance_restart', });
17         });

19         socket.emit('admin.restart');
20       }
21     });
22   });
23   return Admin;
24 });
```

Notorious for:
Making it hard to reason about the execution.

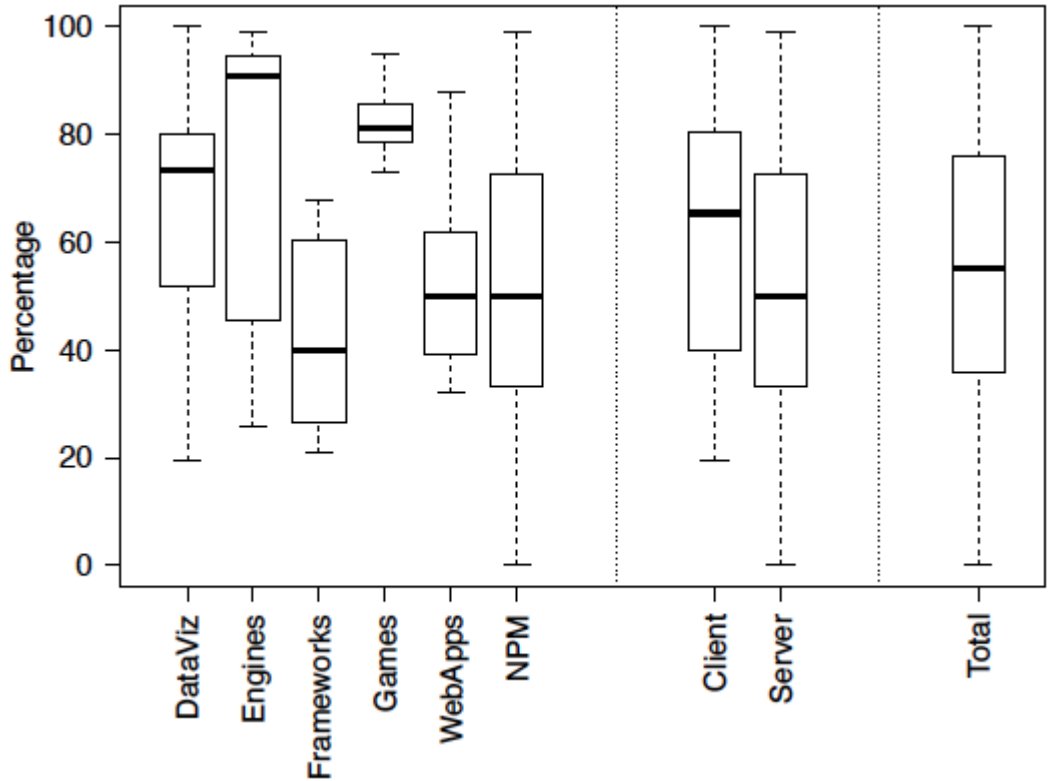
Asynchronous Callbacks

Category	Examples	Availability
DOM events	`addEventListener`, `onclick`	Browser
Network calls	`XMLHTTPRequest.open`	Browser
Timers (macro-Task)	`setImmediate()`, `setTimeout()`, `setInterval()`	Browser, Node.js
Timers (micro-task)	`process.nextTick()`	Node.js
I/O	APIs of `fs`, `net`	Node.js

Some Asynchronous APIs in JavaScript

Asynchronous Callbacks – Results

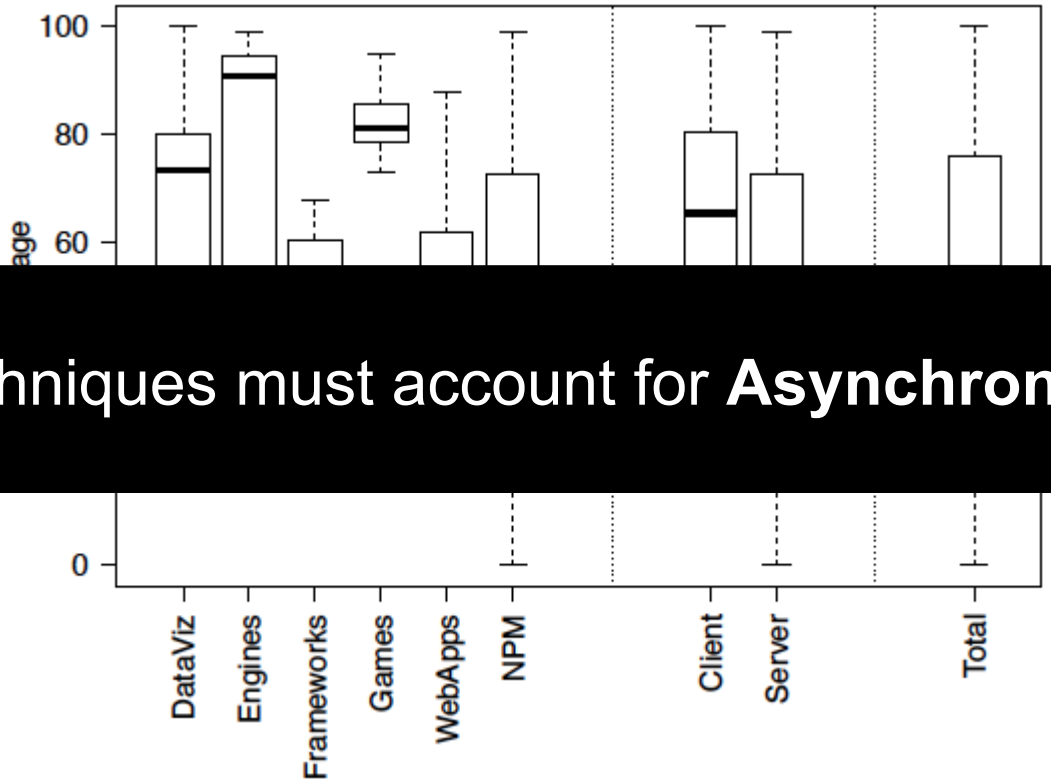
- More than half **(56%)** of all callbacks are Asynchronous.
- Asynchronous callbacks, on average, appear more frequently in client-side code **(72%)** than in server-side code **(55%)**.



Asynchronous Callbacks – Results

- More than half **(56%)** of all

frequently in client-side code **(72%)** than in server-side code **(55%)**.



Program analyses techniques must account for **Asynchrony.**

Nested Callbacks

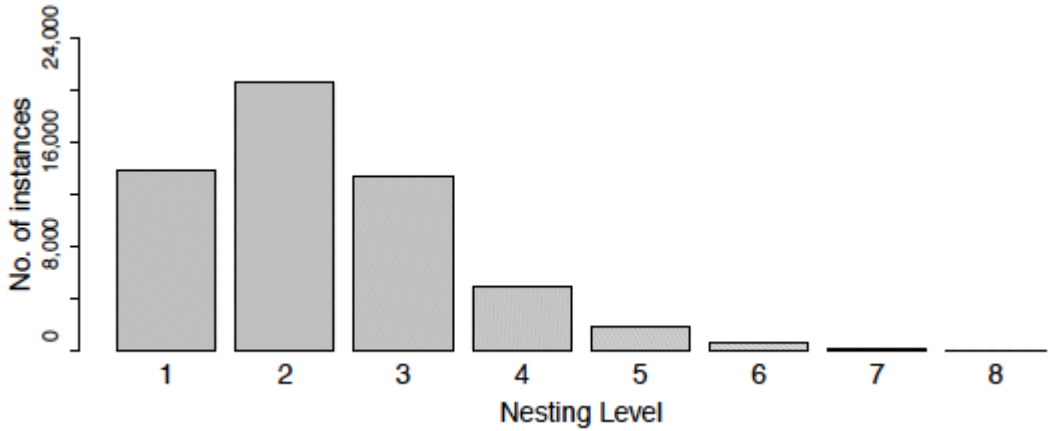
Nesting Level

```
1  define('admin/general/dashboard', 'semver', function(semver
       ) {
2    var Admin = {};

4    $('#logout-link').on('click', function() {
5      $.post(RELATIVE_PATH + '/logout', function() {
6        window.location.href = RELATIVE_PATH + '/';
7      });
8    });

10   ...

12   $('.restart').on('click', function() {
13     bootbox.confirm('Are you sure you wish to restart
         NodeBB?', function(confirm) {
14       if (confirm) {
15         $(window).one('action:reconnected', function() {
16           app.alert({ alert_id: 'instance_restart', });
17         });

19         socket.emit('admin.restart');
20       }
21     });
22   });
23   return Admin;
24 });
```

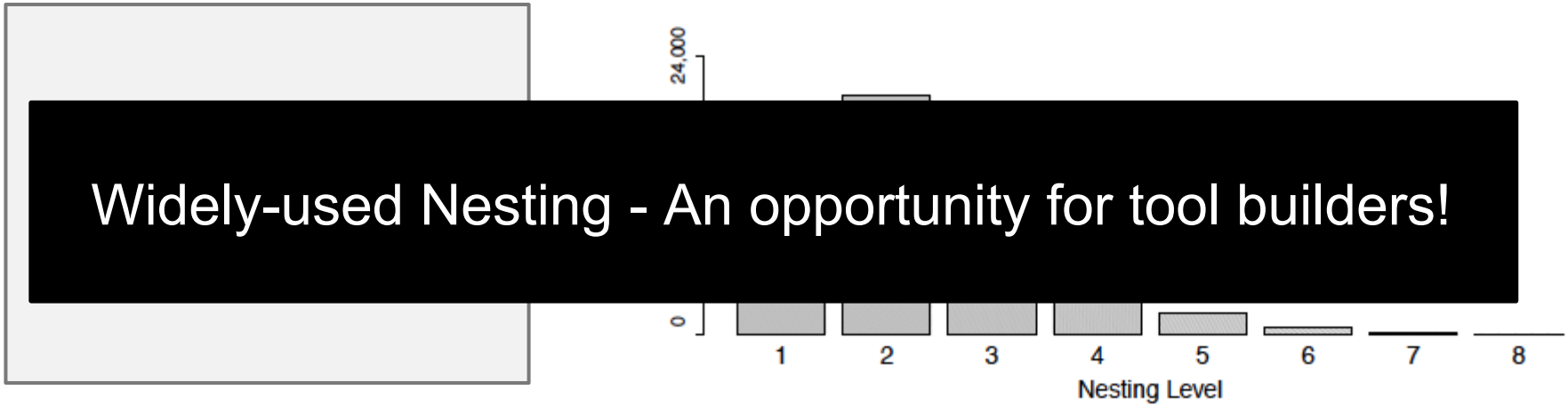
Notorious for:
Callback hell aka Pyramid of Doom

Nested Callbacks - Results

- Callbacks are nested up to a depth of 8.
- There is a peak at nesting level of 2.



Nested Callbacks - Results



Widely-used Nesting - An opportunity for tool builders!

Outline

Motivation ✔
Methodology & Overview ✔
Characterizing Problems ✔
- Anonymous Callbacks ✔
- Asynchronous Callbacks ✔
- Nested Callbacks ✔

Characterizing Solutions
- Error-first Protocol
- Async.js
- Promises

Conclusion

Outline

Motivation ✔
Methodology & Overview ✔
Characterizing Problems ✔
- Anonymous Callbacks ✔
- Asynchronous Callbacks ✔
- Nested Callbacks ✔

- Characterizing Solutions
  - Error-first Protocol
  - Async.js
  - Promises

Conclusion

Error-first Protocol

- JS has no explicit language support for asynchronous error-signaling
- Developer community has the convention:

Dedicate the 1st argument in the callback to be a permanent place-holder for error-signalling

```
1  var fs = require('fs');
2  // read a file
3  function read_the_file(filename, callback) {
4      fs.readFile(filename, function (err, contents) {
5          if (err) return callback(err);
6
7          // if no error, continue
8          read_data_from_db(null, contents, callback);
9      });
10 }
11
12 function read_data_from_db(err, contents, callback) {
13     //some long running task
14 }
15
16 read_the_file('/some/file', function (err, result) {
17     if (err) {
18         //handle the error
19         console.log(err);
20         return;
21     }
22     // do something with the result
23 });
```

Error-first Protocol

- JS has no explicit language support for asynchronous error-signaling
- Developer community has the convention:

Dedicate the 1st argument in the callback to be a permanent place-holder for error-signalling

```
1  var fs = require('fs');
2  // read a file
3  function read_the_file(filename, callback) {
4      fs.readFile(filename, function (err, contents) {
5          if (err) return callback(err);
6
7          // if no error, continue
8          read_data_from_db(null, contents, callback);
9      });
10 }
11
12 function read_data_from_db(err, contents, callback) {
13     //some long running task
14 }
15
16 read_the_file('/some/file', function (err, result) {
17     if (err) {
18         //handle the error
19         console.log(err);
20         return;
21     }
22     // do something with the result
23 });
```

Error-first Protocol - Results

To detect error-first protocol we checked

- if the first parameter **p** of a function definition **f** has the name ‘error’ or ‘err’
- if **f**’s callsites also contain ‘error’ or ‘err’ as their first argument.

We found

- **20%** of function definitions follow the error-first protocol
- The error-first protocol is used twice as often in server-side code than in client-side code (**30%** Vs **16%**)
- **73%** (63 out of 86) NPM modules and **93%** (15 out of 16) web applications had instances of it.

Error-first Protocol - Results

To detect error-first protocol we
- If F's callsites also contain 'error' or 'err' as their first argument.

We found
- **20%** of function definitions follow the error-first protocol
- **73%** (63 out of 86) NPM modules and **93%** (15 out of 16) web applications had instances of it.

Cannot depend on APIs/libraries to enforce **error-first protocol**.

Usage of Async.js

Async.js: popular library to manage asynchronous control flow, and to help with functional programming. For example:

Without Async.js

```
var users = [];
fetchUsers(function() {
  renderUsersOnPage(function() {
    fadeInUsers(function) {
      loadUserPhotos(function() {
         // do something
      });
    });
  });
});
```

With Async.js

```
var users = [];
async.series([
  function(callback) {
    fetchUsers(callback);
  },
  function(callback) {
    renderUsersOnPage(callback);
  },
  function(callback) {
    fadeInusers(callback);
  }
  function(callback) {
    loadUserPhotos(callback);
  }
]);
```

Usage of Async.js - Results

- More than half of the web applications (**56%**) use Async.js.
- Usage is much lower (**11%**) in the NPM modules.
- Async.js library is used differently in these 2 categories of subject systems.

Top 10 Async.js invoked methods in JavaScript Web Applications (Left) and NPM modules (Right).

Rank	Method	Count	Rank	Method	Count
1	nextTick	18	1	parallel	189
2	queue*	16	2	apply	81
3	each	14	3	waterfall	72
3	setImmediate*	14	4	series	61
3	series	14	5	each	48
6	auto*	11	6	map	37
6	waterfall	11	7	eachSeries*	20
6	parallel	11	8	eachLimit*	12
9	map	10	9	whilst*	10
9	apply	10	9	nextTick	10

The * symbol denotes calls that do not appear in both tables.

Usage of Async.js - Results

- More than half of the web applications (**56%**) use Async.js
- Async.js library is used differently in these 2 categories of subject systems.

Top 10 Async.js invoked methods in JavaScript Web Applications (Left) and NPM modules (Right).

Rank	Method	Count	Rank	Method	Count
					89
					81
					72
					61
					48
6	auto*	11	6	map	37
6	waterfall	11	7	eachSeries*	20
6	parallel	11	8	eachLimit*	12
9	map	10	9	whilst*	10
9	apply	10	9	nextTick	10

Difference in API usage across categories indicate different underlying concerns in callback management!

The * symbol denotes calls that do not appear in both tables.

Conclusions

- Callbacks are extensively used in the **server-side**
- Program analyses techniques must account for **Asynchrony**
- In spite of maintenance challenges, **Anonymous callbacks** are here to stay
- Widely-used Nesting - An opportunity for tool builders.
- Cannot depend on APIs/libraries to enforce **error-first protocol**.

```
http://salt.ece.ubc.ca/callback-study
```

Extra Slides

Usage of Promises

Promises: a native language feature for solving the Asynchronous composition problem. For example:

Without Promises

```
getUser('mjackson', function (error, user) {
  if (error) {
    handleError(error);
  } else {
    getNewTweets(user, function (error, tweets) {
      if (error) {
        handleError(error);
      } else {
        updateTimeline(tweets, function (error) {
          if (error) handleError(error);
        });
      }
    });
  }
});
```

With Promises

```
getUser('mjackson')
        .then(getNewTweets,null)
        .then(updateTimeline)
        .catch(handleError);
```

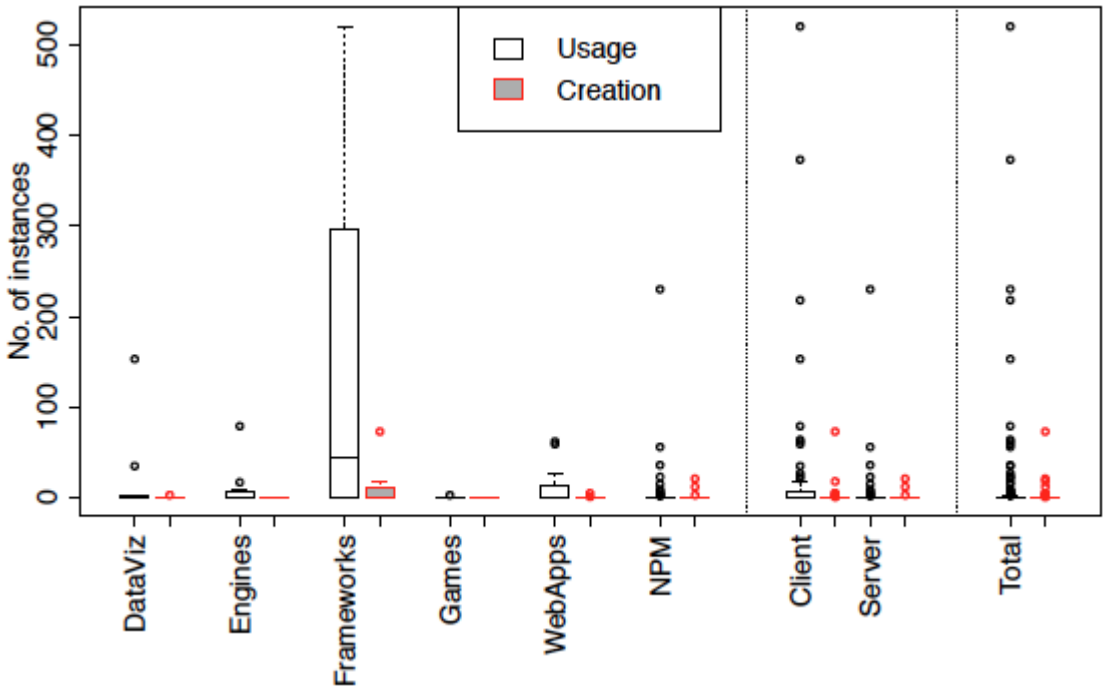
Usage of Promises - Results

27%

37 of 138
subject systems
use Promises.

Category	Subjects creating Promises (%)	Subjects using Promises (%)
DataViz libraries	6	31
Game Engines	0	25
Frameworks	50	75
Games	0	17
Web Applications	13	50
NPM Modules	3	12
Total	8	26

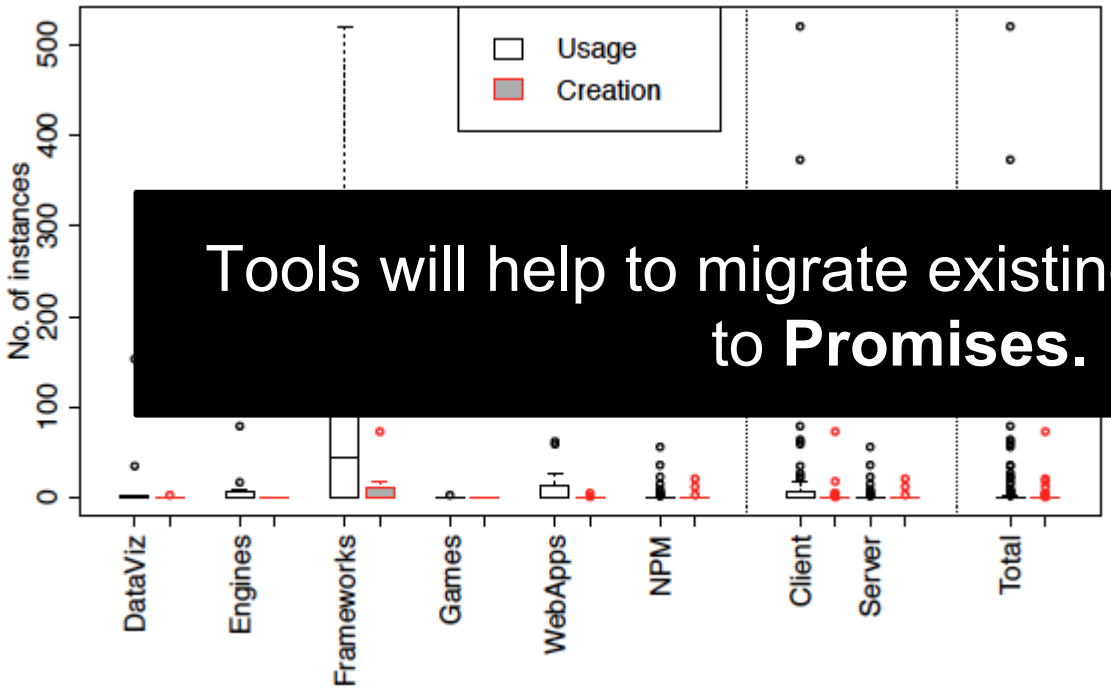
Usage of Promises - Results



Promises are prominently used on the Client-Side

Max: 513 usage instances

Usage of Promises - Results



Tools will help to migrate existing large projects to **Promises.**

Promises are
on the
de

Max: 513 usage instances

Callback Accepting Functions

- **f** is a *callback-accepting function definition* if at least one argument to **f** is used as a callback.

- A parameter **p** of a function **f** definition is a callback..
  - if **p** is invoked as a function in the body of **f**
  - if **p** is passed to a known callback-accepting function (e.g., `setTimeout()`)
  - if **p** is used as an argument to an unknown function **f ′**
    - and then we can recursively determine if **p** is a callback parameter in **f′**

- a *callsite* is callback-accepting if it is of
  - a function that was detected to be callback-accepting, as above
  - a function known a-priori to be callback-accepting (e.g., `setTimeout()`)