Server execution of JavaScript: What could possibly go wrong?

Brian Geffon
Staff Software Engineer

Hello!

Brian Geffon

Staff Software Engineer at LinkedIn

San Francisco Bay Area | Computer Software

500+ connections

Current: LinkedIn, The Apache Software Foundation

Recommendations: 4 people have recommended **Brian**

Contributor, Apache Traffic Server

The Apache Software Foundation

2011 – Present (3 years)

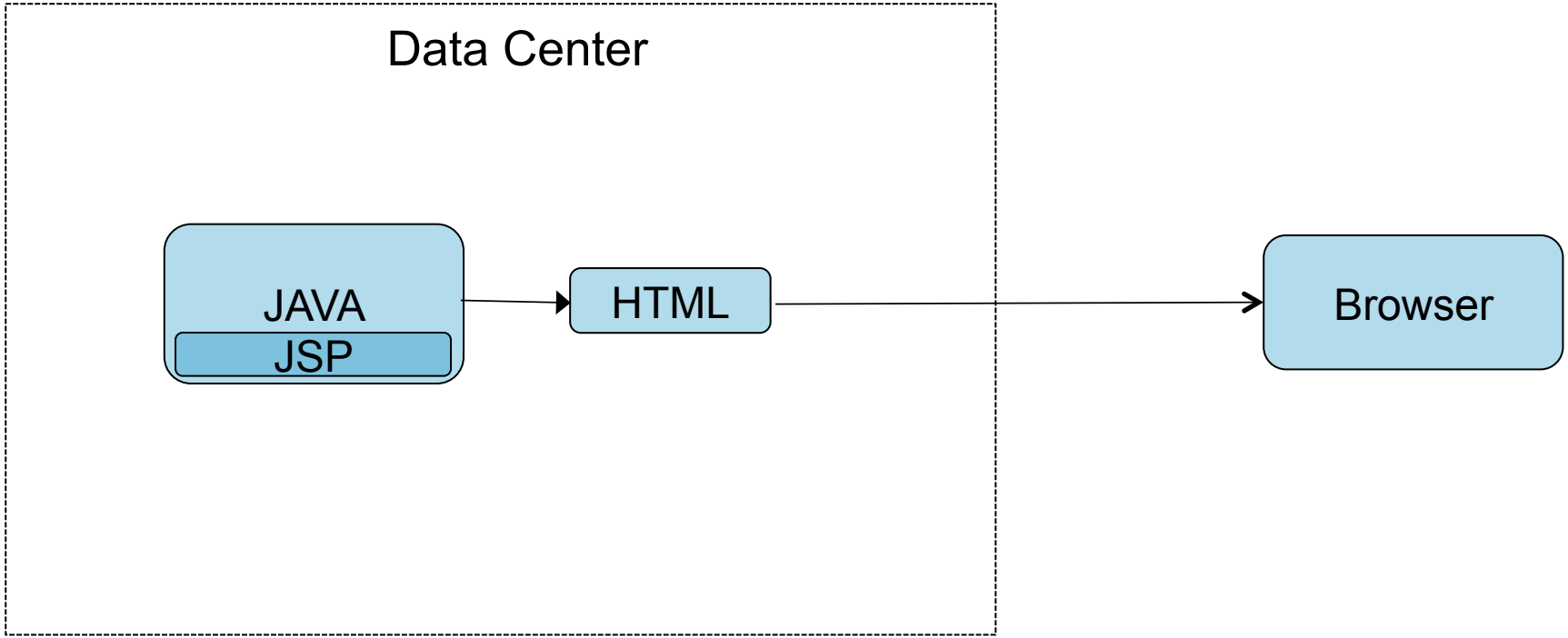
Committer and PMC member on Apache Traffic Server.

Apache Traffic Server is fast, scalable and extensible HTTP/1.1 compliant caching proxy server. It scales well on modern SMP hardware, handling 10s of thousands of requests per second.

Outline

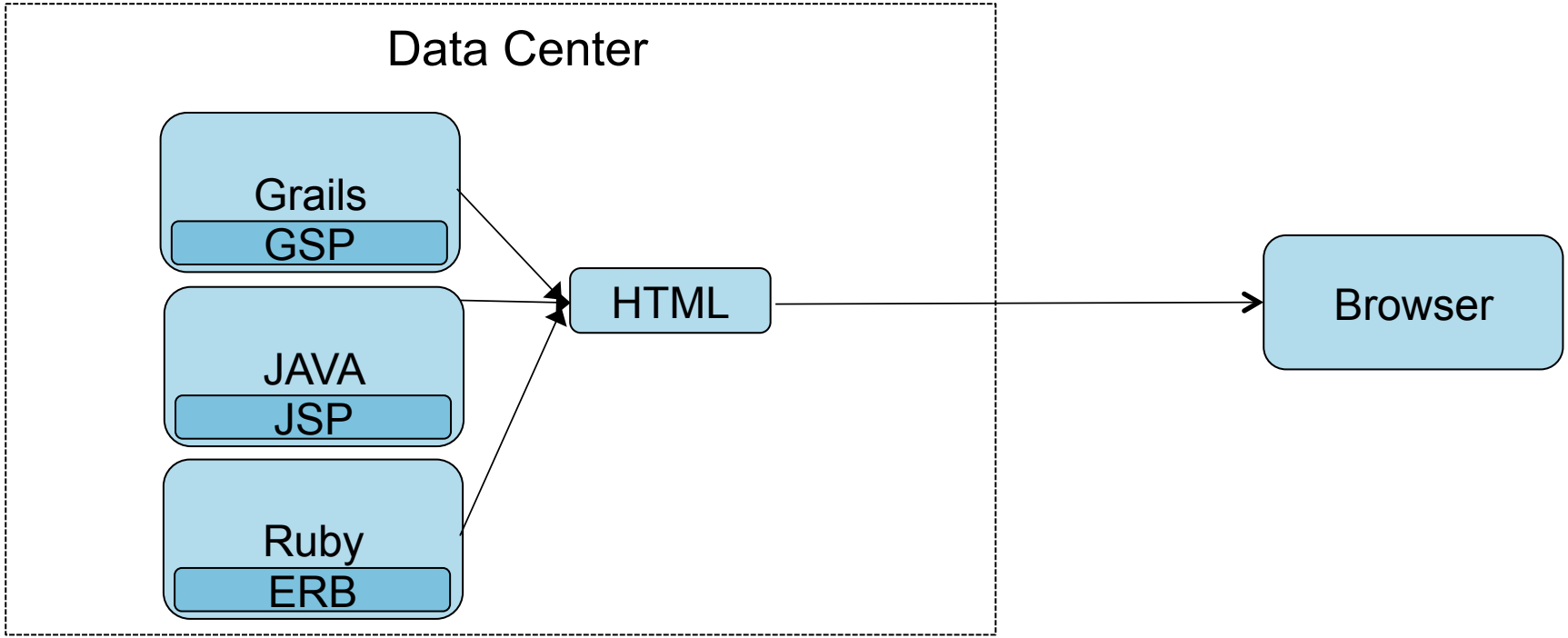
- Introductions
- **Brief History**
- The paradigm shift
- Problems!
- Where we are today
- Closing thoughts and Questions

LinkedIn in 2003



A single monolithic web application

LinkedIn in 2010



New frameworks: productivity boost.

New Frameworks: Added productivity
Added complexity

- Difficult to maintain numerous versions of the same template

- Make it difficult to share content between apps

Solution: a single templating language

- Do these web app frameworks share anything?
- How can we ensure that we remain D.R.Y.
- What language can be supported across each architecture?

Solution: client side templating

- Web applications return JSON data
- Templates are compiled to JavaScript
- JSON Data is consumed by JavaScript templates which will execute on the client side.

Solution: client side templating, contd.

- Webapps can share UI!
- Ability to cache templates on the client
  - Better performance?

So many options!

The winner: Dust.js

- Dust is a logicless JavaScript templating language
- Dust is extensible
- Dust is inherently D.R.Y.

https://github.com/linkedin/dustjs

Dust.js example

```
<h2>{first} {last} is a {occupation}.</h2>
```

gets compiled into a JavaScript function

```
(function() {
  dust.register("demo", body_0);

  function body_0(chk, ctx) {
    return chk.write("<h2>").reference(ctx.get("first").write(" ")
     .reference(ctx.get("last").write(" is a ").reference(ctx.get("occupation")
     .write("."), ctx, "h").write("</h2>");
  }
  return body_0;
})();
```

+ `{"name": "Fizz Bang"}` = `<h2>Hello Fizz Bang!</h2>`

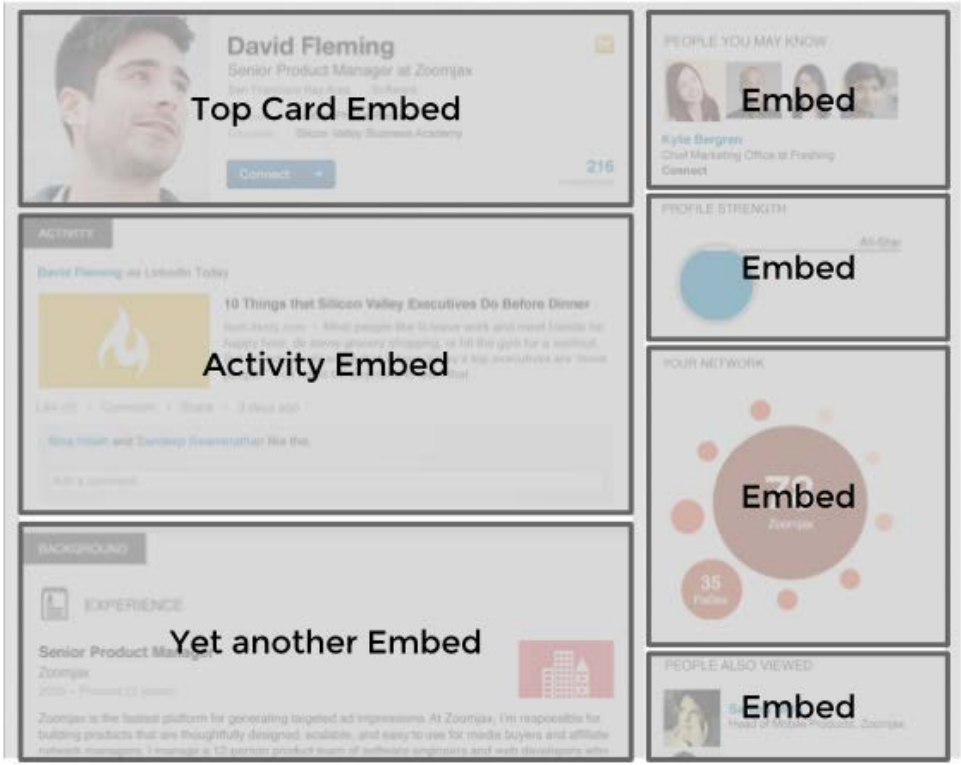
Outline

- Introductions
- Brief History
- **The paradigm shift**
- Problems!
- Where we are today
- Closing thoughts and Questions

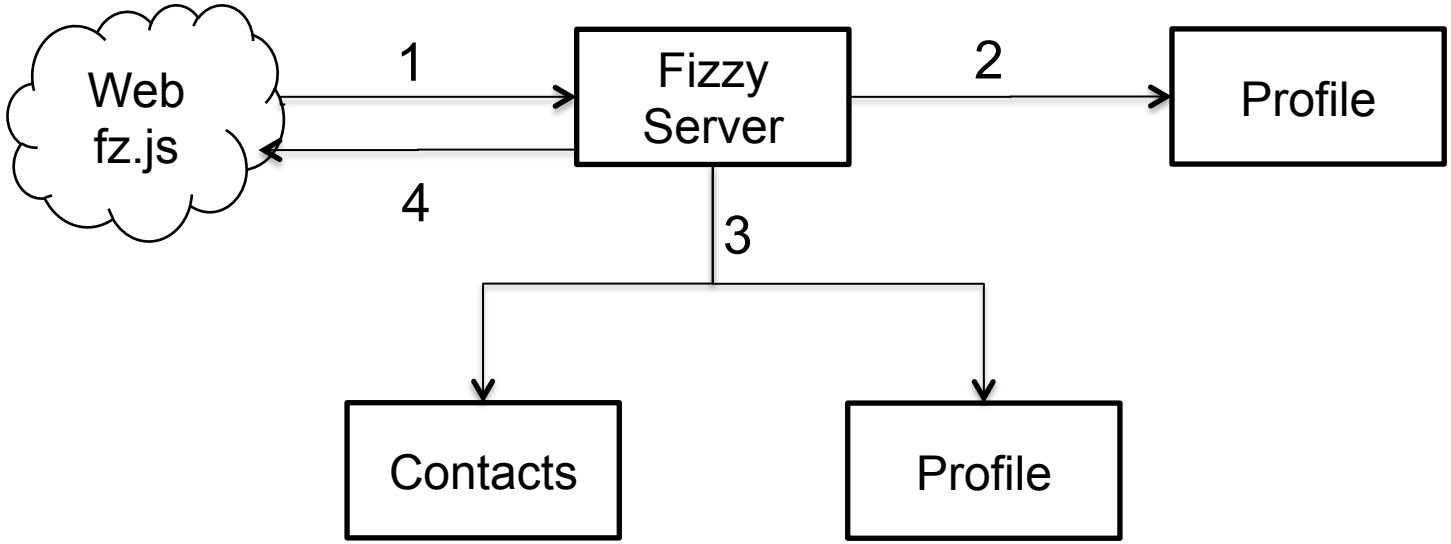
The paradigm shift

- Reusable UI gives rise to component sharing across apps
- Components are now separated from data models
- Ability to avoid RTT for components embedded in page.

The paradigm shift: Fizzy

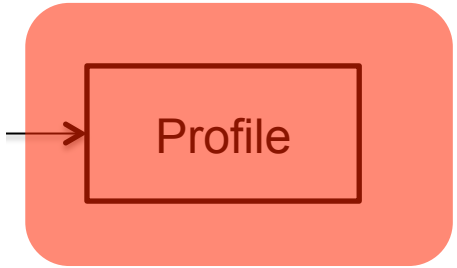


What's going on here?



What's does the application return?

```
HTTP/1.1 200 OK
Content-Type: text/html
X-FS-Page-Parse: 1
X-FS-Page-Id: profile-view-fs
X-FS-Host-Id: ela4-appxxxx.prod
```

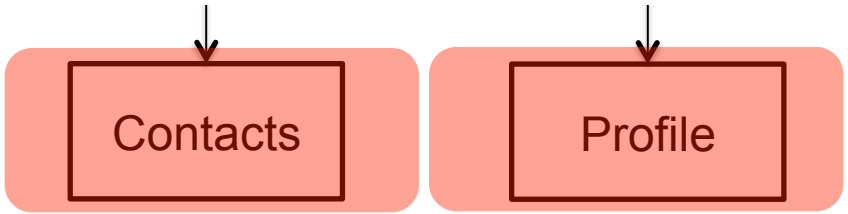



```
1 <html>
2   <body>
3     <script type="fs/embed"
4             fs-uri="http://www.linkedin.com/profile/summary"
5             fs-id="example-id">
6     </script>
7 </html>
```

What do these embedded components return?

```
HTTP/1.1 200 OK
Content-Type: application/json
X-FS-Page-Id: profile-activity
X-FS-Host-Id: ela4-appxxxx.prod
X-FS-TL: http://cdn-host/hash-of-template1.js
X-FS-Template-Keys: __default__=hash-of-template1
```

```
{"first": "Brian", "last": "Geffon", "Occupation": "Software Engineer"}
```



What does the browser see?

```
1  <html>
2    <head>
3      <script type="text/javascript” src=”/static/fz.js"></script>
4    </head>
5    <body>
6      <script type="text/javascript"
7              src="http://cdn-host/hash-of-template1.js"></script>
8      <code id="example-id-content">
9        <!-- {"first":"Brian","last":"Geffon", "occupation": "..."} -->
10     </code>
11     <script type="text/javascript">
12       fs.embed("example-id" , "hash-of-template1");
13     </script>
14   </body>
15 </html>
```

Yay! A fancy new web architecture

- Components are now stand alone
- Nice UI separation
- Reusability

What could possibly go wrong?

- Large JSON payloads caused many problems with IE7
  - IE7 doesn't have a native JSON parser!

What could possibly go wrong?

- Some older browsers would take a very long time executing JS
  - Many browsers didn’t have optimized JS engines

What could possibly go wrong?

- Search Engine Optimization
  - JS in GoogleBot Yes, many others: No

Server Side Rendering (SSR)

- Unfortunately we need a way to execute JavaScript on the server
- Could potential performance improvements been seen across the board?

The Pieces of SSR

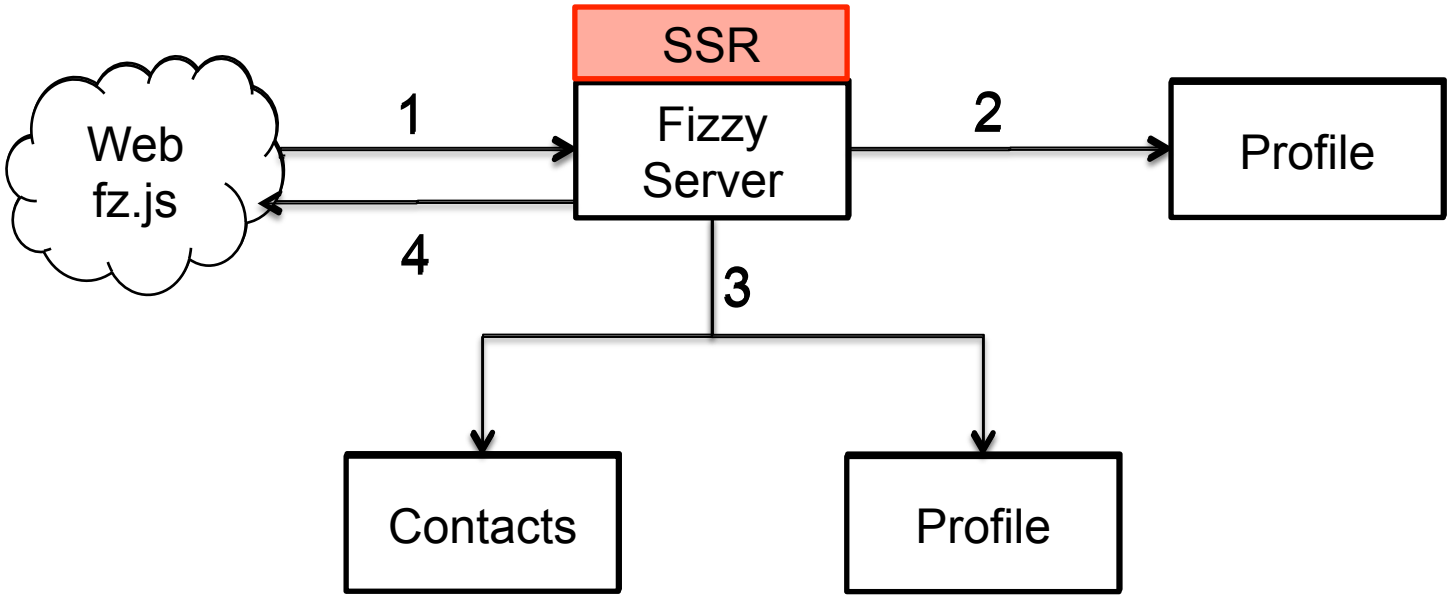
traffic server

- High performance caching HTTP proxy

Google V8 JS Engine

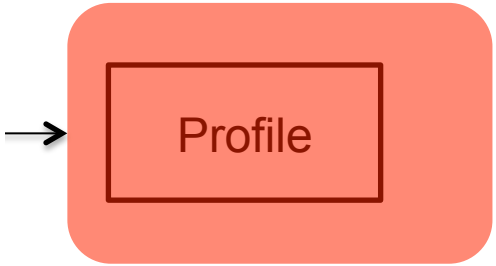
- High performance embeddable JavaScript Engine

Server Side Rendering: What's going on here?



What's does the application return?

```
HTTP/1.1 200 OK
Content-Type: text/html
X-FS-Page-Parse: 1
X-FS-Page-Id: profile-view-fs
X-FS-Host-Id: ela4-appxxxx.prod
```




```
1  <html>
2    <body>
3      <script type="fs/embed"
4              fs-uri="http://www.linkedin.com/profile/activity"
5              fs-id="example-id"
6              fs-render-control="server">
7      </script>
8  </html>
```

What does the browser see?

```
1 <html>
2   <head>
3     <script type="text/javascript" src="/static/fz.js"></script>
4   </head>
5   <body>
6         <h2>Brian Geffon is a Software Engineer.</h2>
7   </body>
8 </html>
```

Yay! A fancy new web architecture

- We can now support old web browsers
- We can now gracefully handle SEO
- It turns out that even for modern browsers sometimes we can execute JavaScript faster!

Outline

- Introductions
- Brief History
- The paradigm shift
- **Problems**
- Where we are today
- Closing thoughts and Questions

What could possibly go wrong?

- A shared JS engine gives rise to issues and vulnerabilities that don’t affect browsers that execute JS.

What could possibly go wrong?

- Context Pollution
  - One malicious request can poison the context of another
  - This issue exists with any dynamic language

Context Pollution

```
1 Math.pow = function(x,y) {
2     return x + y;
3 }
```

Silly example but illustrates the need for isolation.

What if we leave off **var** in JavaScript?

Context Pollution: The solution

- Each request requires it's own context
  - Completely reload the environment and bootstrap code
- Performance Hits?

What could possibly go wrong?

- Poorly written JavaScript can take forever to execute!

Poorly Written JavaScript: Infinite Loops, Recursion, etc.

```
1 var fib = function(x) {
2     if (x < 2) return 1;
3     return fib(x - 1) + fib(x - 2);
4 }
5 console.log(fib(1000)); // Surely a stack overflow
```

Although this is tail recursion and a silly example, It illustrates the need for stack protection and time limitations.

Long Running JavaScript: The solution

- Enforce stack size limits that allow you to gracefully kill a VM
- Sandbox: accept that apps will misbehave and allow them to only hurt themselves.

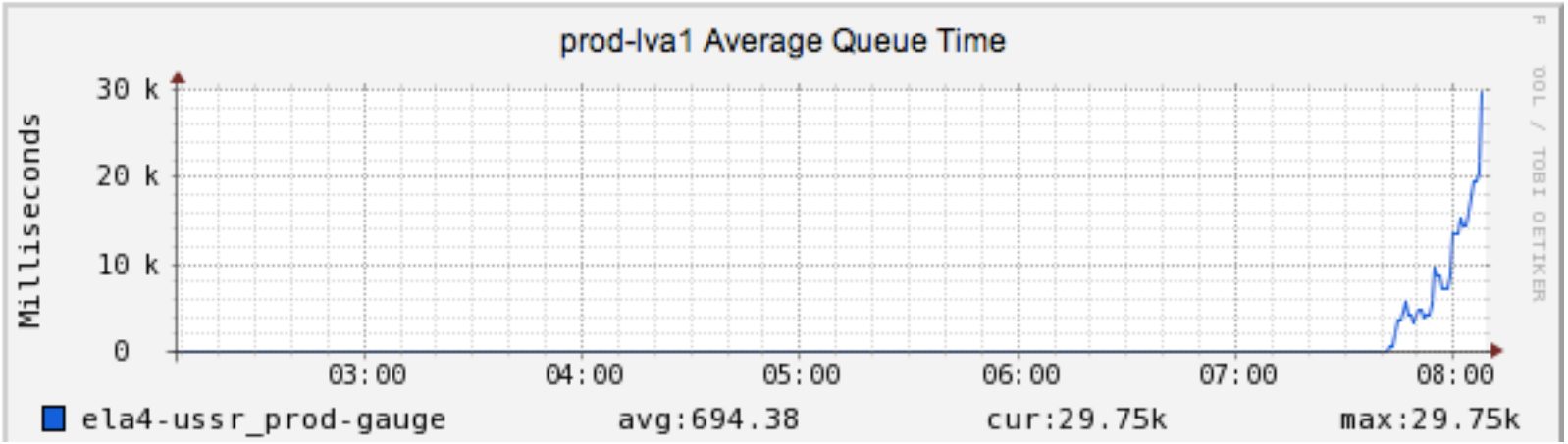
Long Running JavaScript: The solution

- Execution limits (we use 1000ms)
- Exponentially decay the execution limit to prevent taking down the entire site!

What could possibly go wrong?

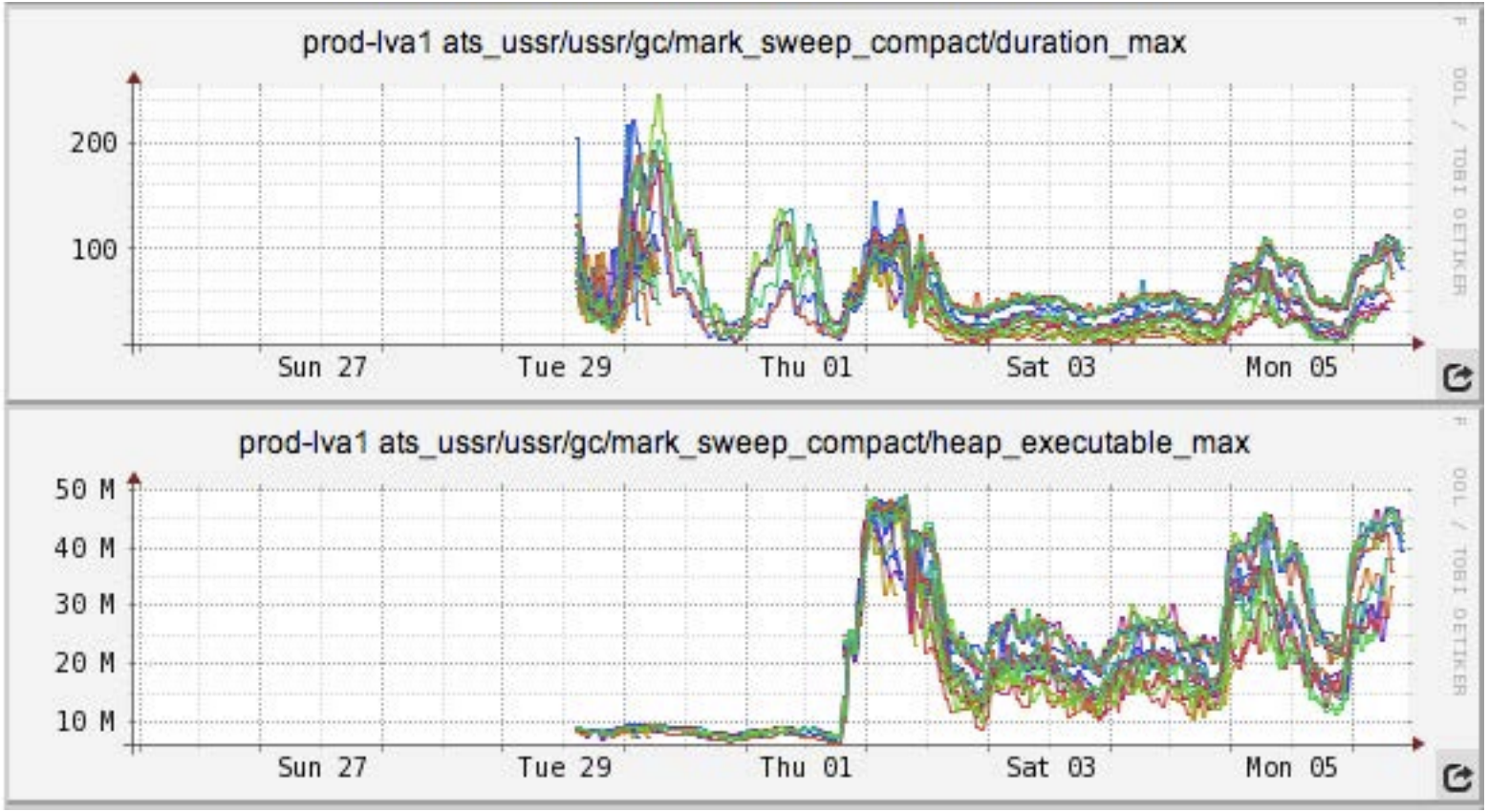
- Garbage Collection!

Garbage Collection!

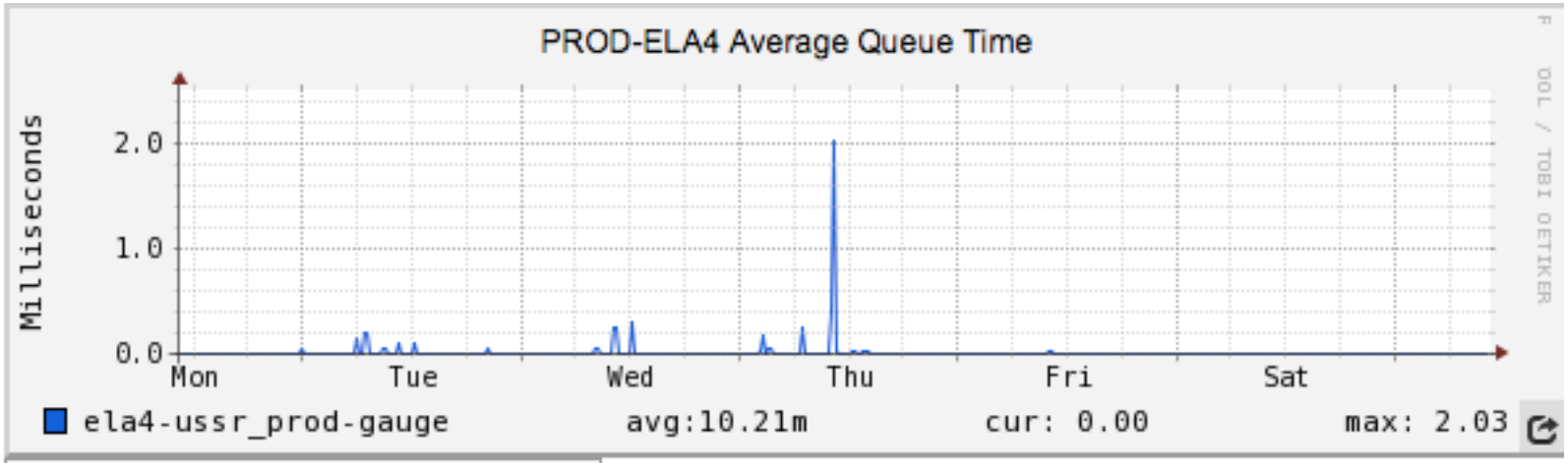


Queue times going through the roof!

The culprit: Garbage Collection!



GC tuning: it takes practice.



Avg Queue times < 0.3ms, P99.99 < 2ms.

GC Tuning

- Adjust old generation to be several order of magnitudes less than new generation
- New generation is critical because of the short lived jobs and contexts.
- More Threads!

Ideas for the future

- User load times are actually improved with SSR: do it 100% of the time.
- A generic JS engine: allow apps to return any JavaScript, not just Dust.js

Questions?