



---

# **Design and Implementation of a Real-Time Cloud Analytics Platform**

OSCON Data 2011

David Pacheco (@dapsays)

Brendan Gregg (@brendangregg)

## **The Problem**

Cloud Analytics

Demo

Experiences

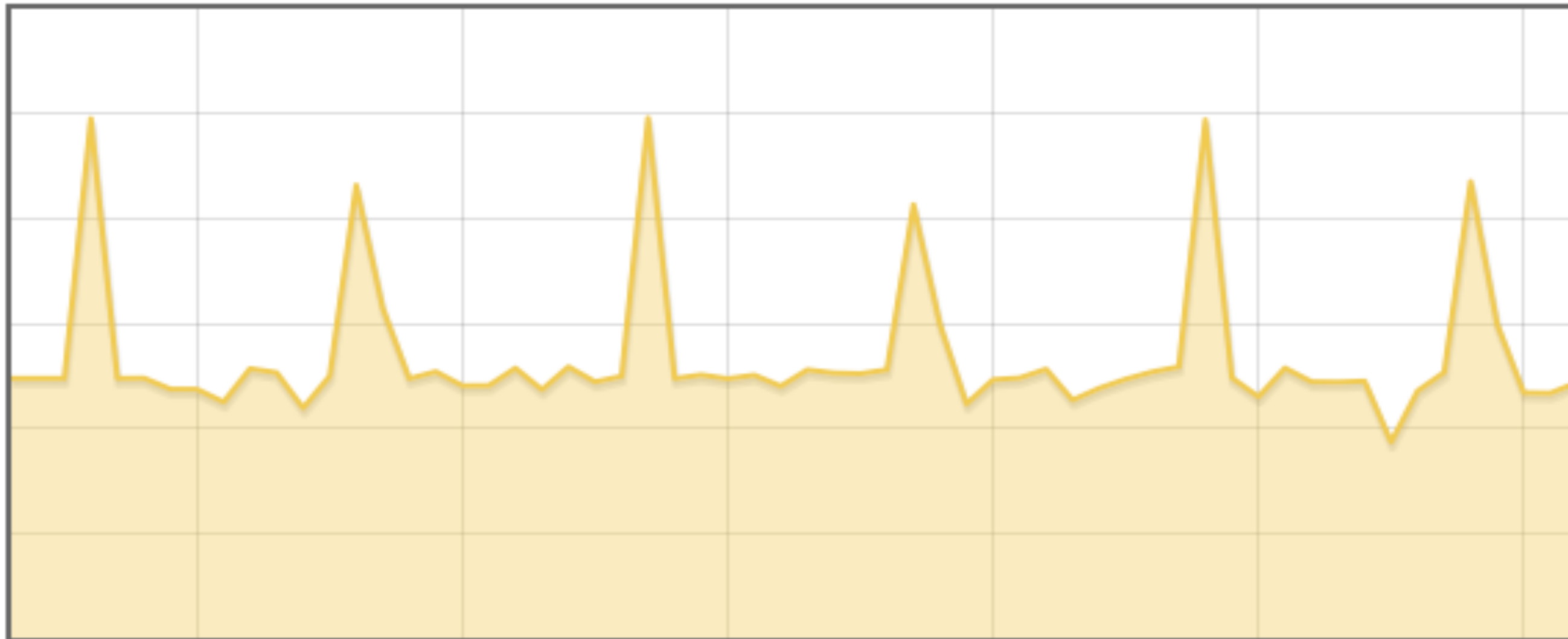
- The problem: we've deployed our software, and now performance sucks.
  - How do we figure out why?
- Focus on source of the pain: **latency**
  - How long a synchronous operation takes
  - ... while a client is waiting for data
  - ... while a user is waiting for a page to load
- How do you **summarize** the latency of thousands of operations?
  - ...without losing important details?
- How do you summarize that **across a distributed system**?
- How do you do this **in real time**?

# Latency: event-by-event

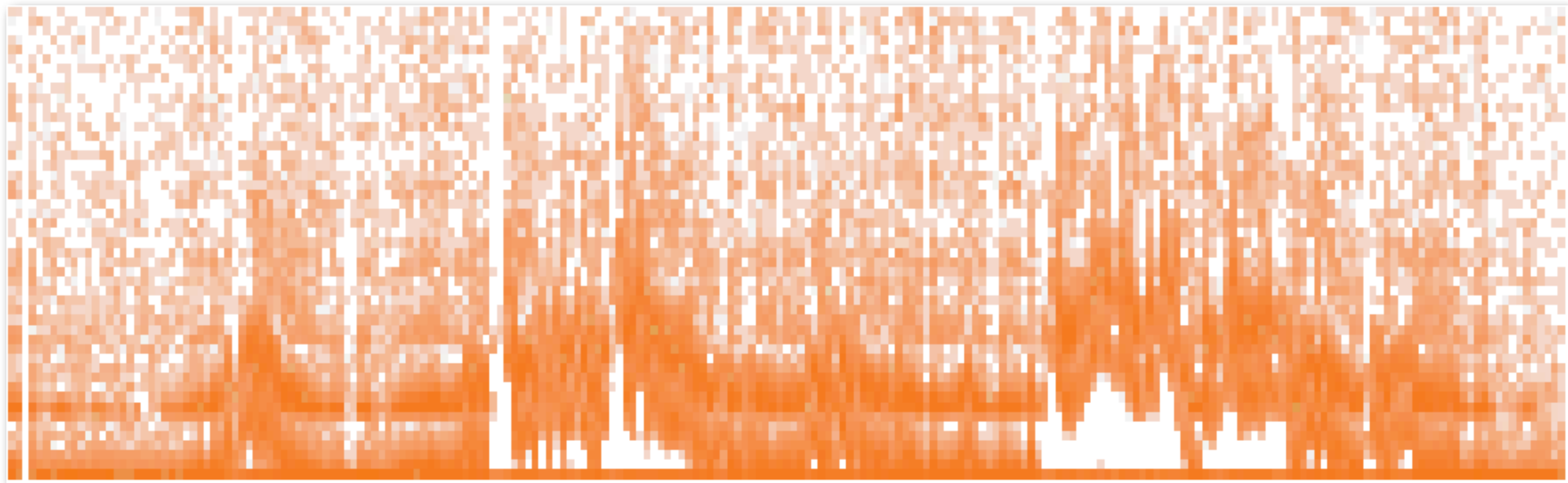


```
# ./iosnoop -Dots
STIME          TIME          DELTA  DTIME        UID   PID D   BLOCK    SIZE    COMM  PATHNAME
949417936651  949417948636  11984  11999        104  29008 R   99310470  16384  mysqld <none>
949418267667  949418275701  8033   8052         104  29008 R   1947809   16384  mysqld <none>
949418843669  949418843808  139    156          0     3   W    29024     2048  fsflush /var/log/...
949418873385  949418873488  103    121          0     3   W    6695855   2048  fsflush <none>
949418873564  949418873617  52     57           0     3   W    1829584   512   fsflush <none>
949418921970  949418932931  10960  10976        104  29008 R   95362430  16384  mysqld <none>
949419684613  949419692319  7706   7723         104  29952 R   81475146  16384  mysqld <none>
949419693574  949419699461  5886   5906         104  29952 R   60593276  16384  mysqld <none>
949422857833  949422857981  148    168          0     3   W    26720     4096  fsflush /var/adm/...
949423846191  949423846355  163    181          0     3   W    1990648   4096  fsflush /var/log/...
949426420134  949426420265  130    151          0     0   R     400      8192  sched <none>
949426420346  949426420423  77     85           0     0   W     65      512   sched <none>
949426420367  949426420459  92     35           0     0   W    129      512   sched <none>
949426420386  949426420490  103    30           0     0   W    146      512   sched <none>
949426420404  949426420566  161    76           0     0   W    193      512   sched <none>
949426420530  949426420604  73     37           0     0   W    206      512   sched <none>
949426420547  949426420679  131    75           0     0   W    210      512   sched <none>
[...thousands of lines...]
```

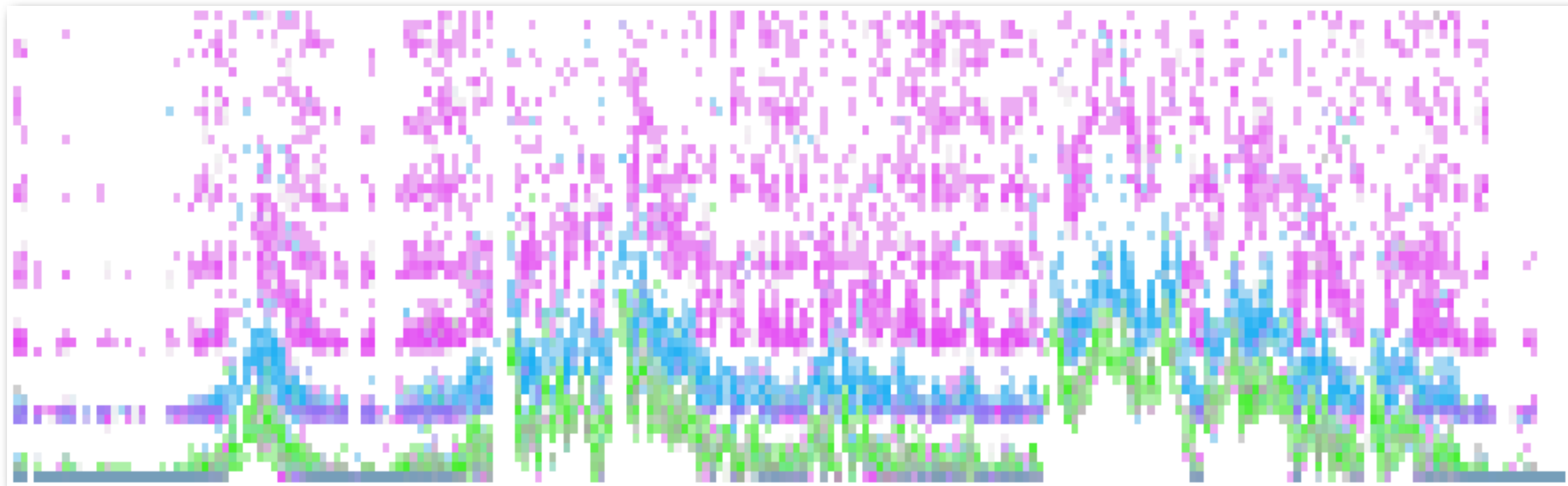
- Lots of of data to sift through; effective as a last resort



- Some patterns more visible; outlier hidden
- x-axis = time, y-axis = average latency



- Great! This example is MySQL query latency
- x-axis = time, y-axis = latency, **z-axis (color saturation) = count of queries**



- Even better! 4th dimension (color hue) represents different database tables
- x-axis = time, y-axis = latency, **color hue = table**, color saturation = count

The Problem

**Cloud Analytics**

Demo

Experiences

- Key building blocks
  - DTrace
  - OS-level virtualization
  - Node.js

- Facility for dynamic instrumentation of production systems
  - Originally developed circa 2003 for Solaris 10, then open-sourced in 2005
  - Available on Solaris-derived OSes (SmartOS, Illumos, etc.)
  - Available on Mac OSX 10.5+, FreeBSD 7.1+, Linux? (<http://crtags.blogspot.com>)
- Supports arbitrary actions and predicates, in situ data aggregation, dynamic and static tracing of both **userland** and **kernel**.
- Designed for safe, ad hoc use in production: concise answers to arbitrary questions

# DTrace example: MySQL query latency



- MySQL query latency can be measured with a (long) one-liner:

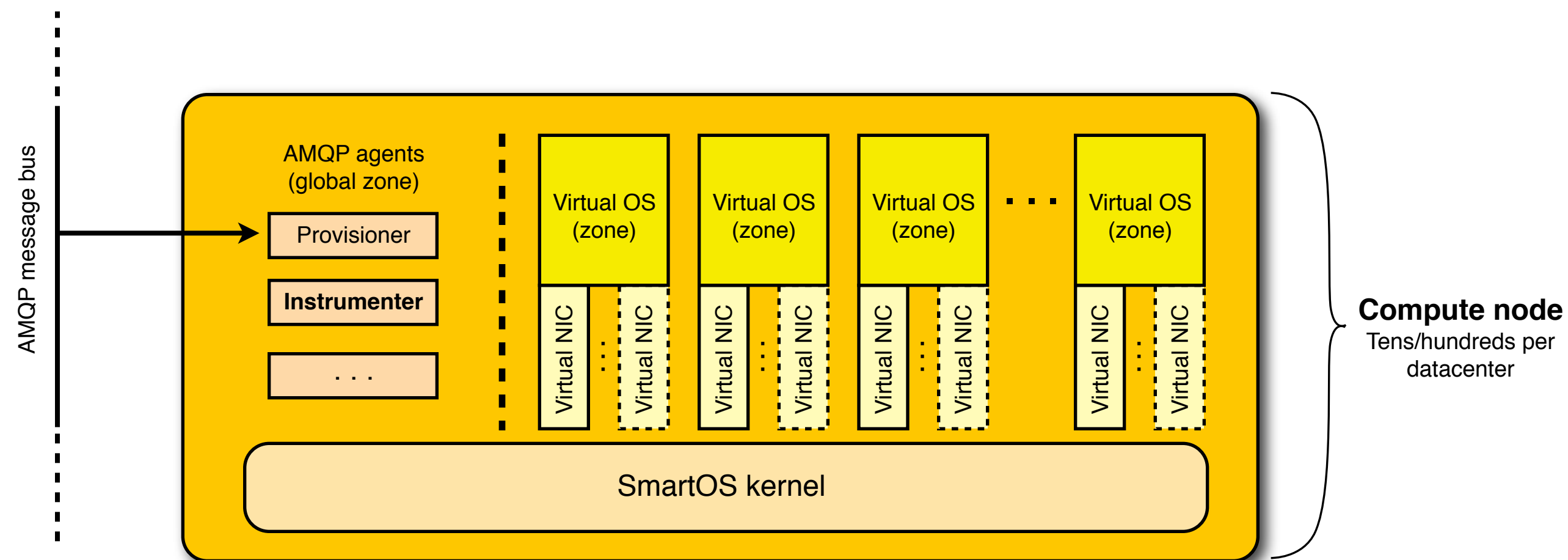
```
# dtrace -n '  
mysql*:::query-start { self->start = timestamp; } mysql*:::query-done /self->start/ {  
  @["nanoseconds"] = quantize(timestamp - self->start);  
  self->start = 0;  
}'
```

nanoseconds

value	----- Distribution -----	count
1024		0
2048		16
4096	@	93
8192		19
16384	@@@	232
32768	@@	172
65536	@@@@@	532
131072	@@@@@@@@@@@@@@@@	1513
262144	@@@@@	428
524288	@@@	258
1048576	@	127
2097152	@	47
4194304		20
8388608		33
16777216		9
33554432		0

## Building blocks: OS-level Virtualization

- The Joyent cloud uses OS-level virtualization to achieve high levels of tenancy on a single kernel without sacrificing performance:



- Allows for transparent instrumentation of *all* virtual OS instances using DTrace

- node.js is a JavaScript-based framework for building event-oriented servers:

```
var http = require('http');

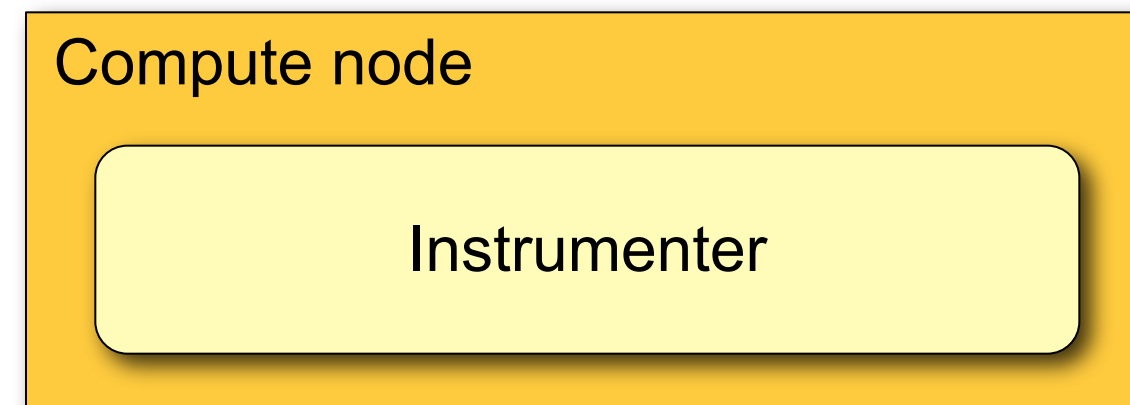
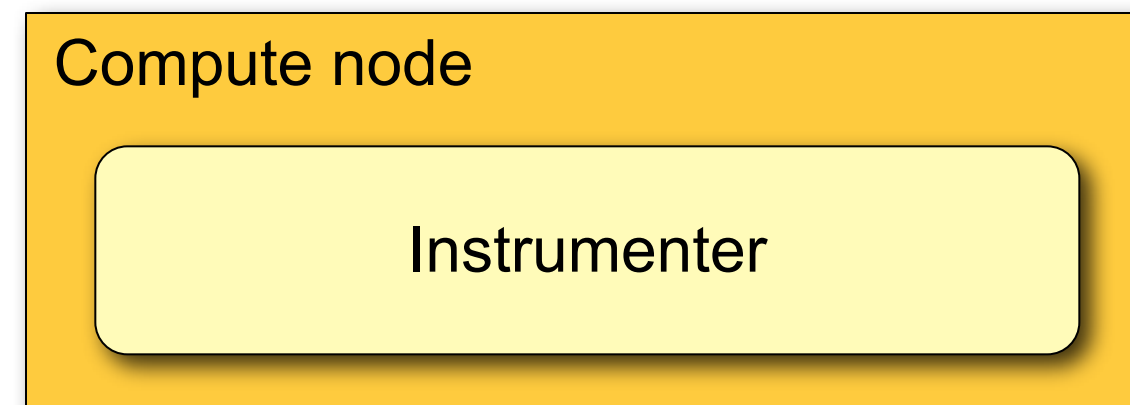
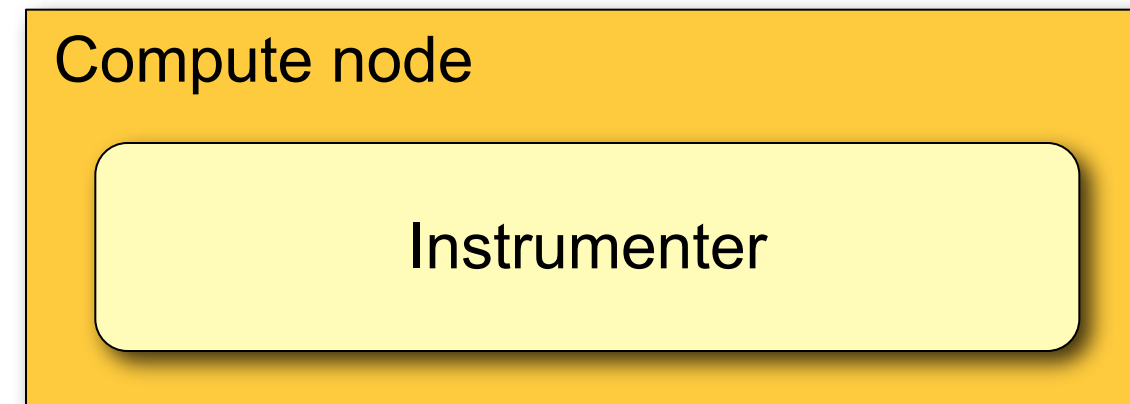
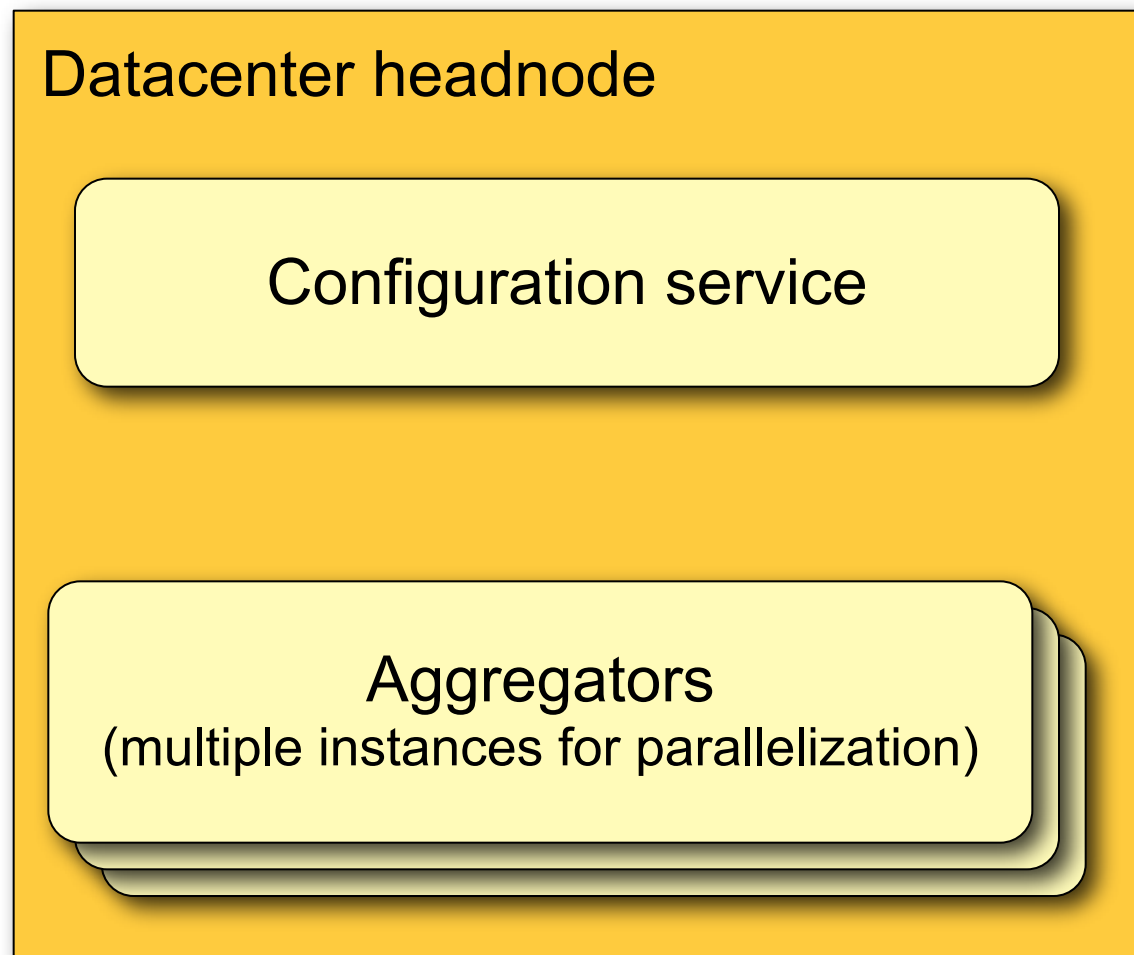
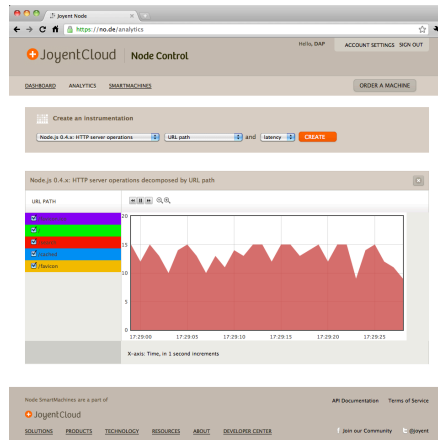
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(8124, "127.0.0.1");

console.log('Server running at http://127.0.0.1:8124/');
```

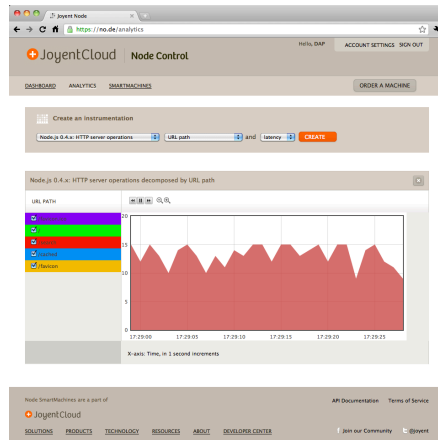
- node.js is a confluence of three ideas:
  - JavaScript's rich support for asynchrony (i.e. closures)
  - High-performance JavaScript VMs (e.g. V8)
  - Solid system abstractions (i.e. UNIX)
- Because everything is asynchronous, node.js is ideal for delivering scale in the presence of long-latency events

- **configuration service:** manages which metrics are gathered
- **instrumenter:** uses DTrace to gather metric data
  - one per compute node, not per OS instance
  - reports data at 1Hz, summarized in-kernel
- **aggregators:** combine metric data from instrumenters
- **client:** presents metric data retrieved from aggregators

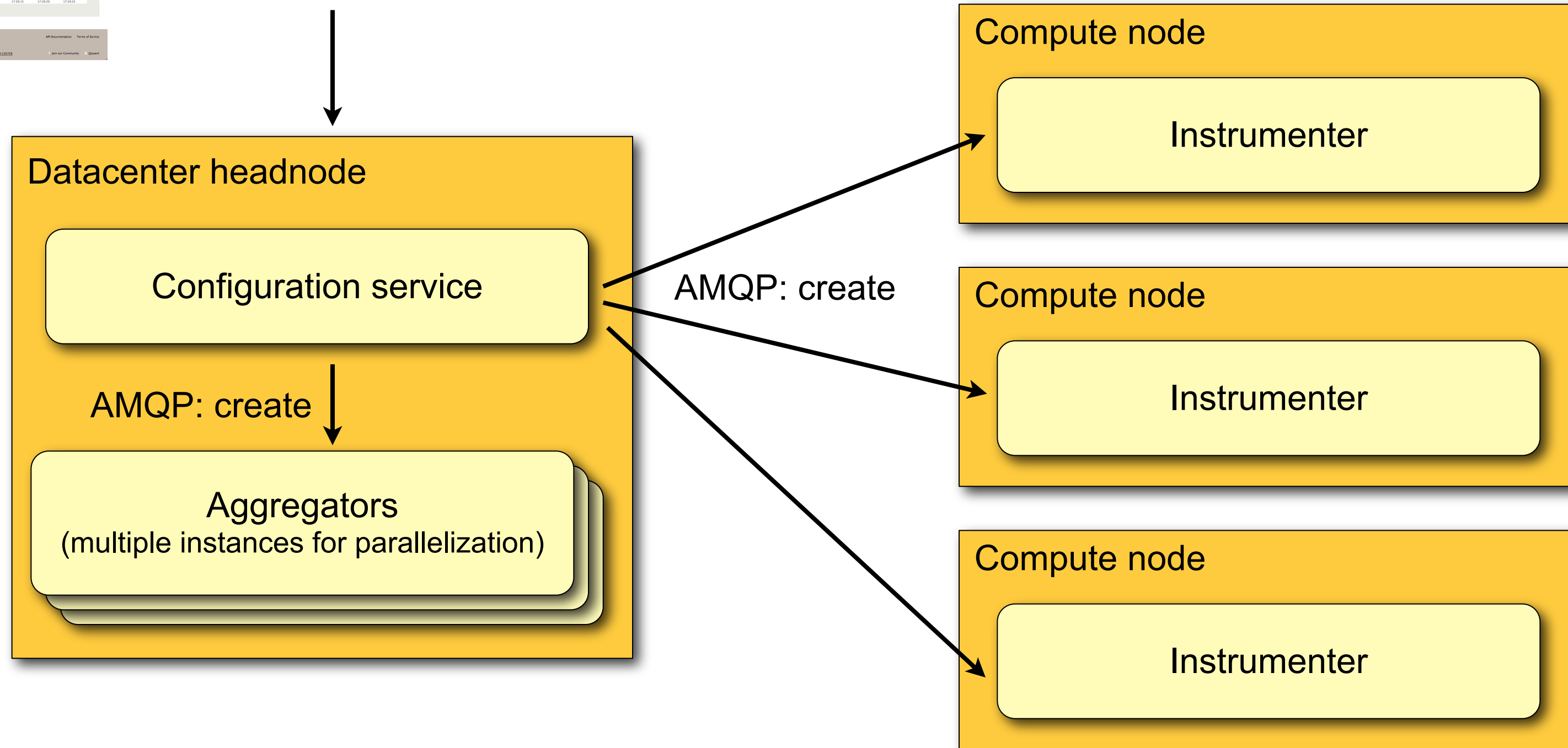
# Distributed service



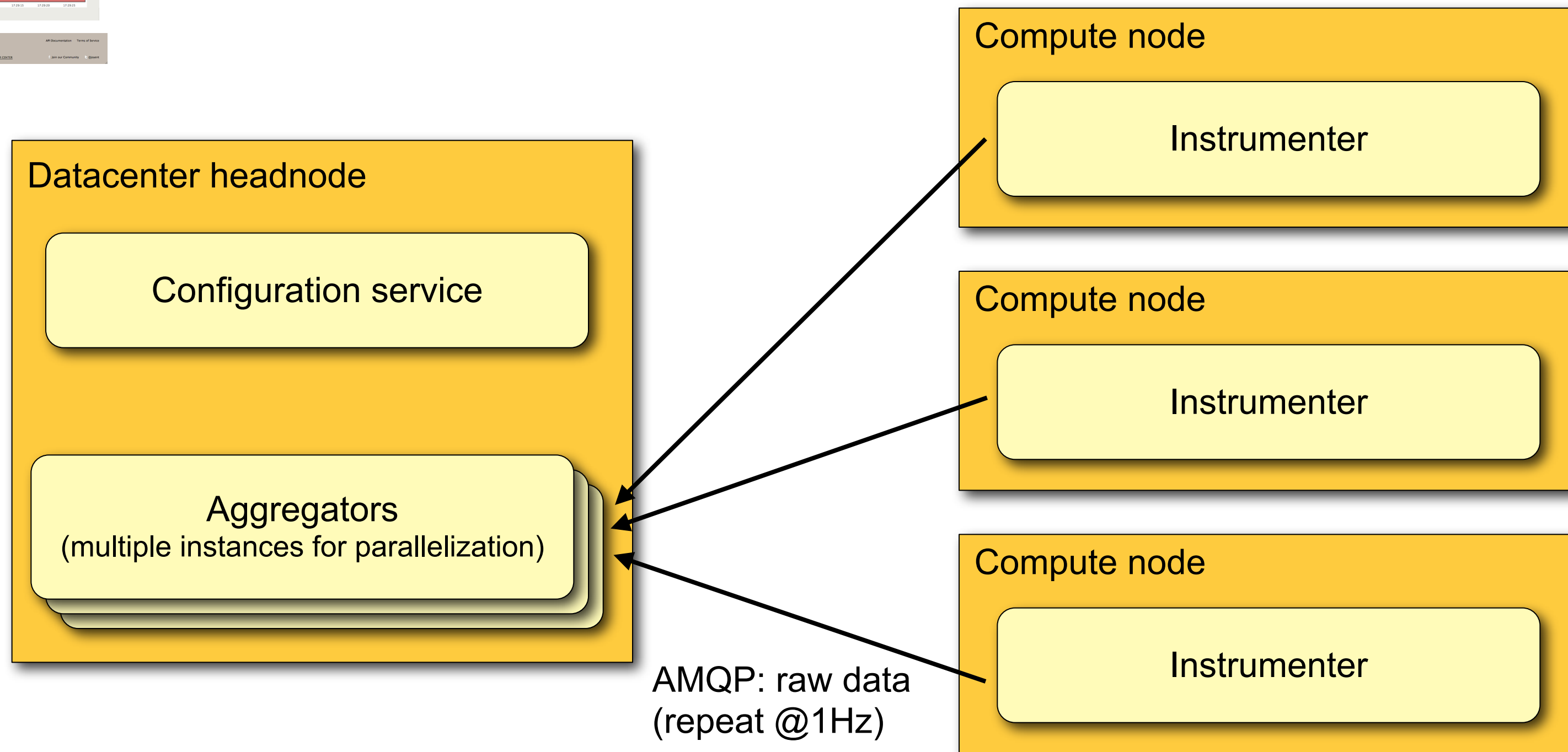
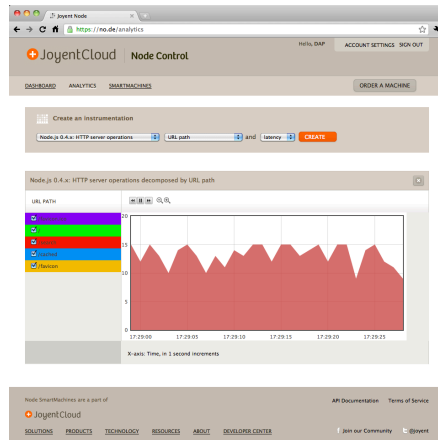
# Step 1: User creates an instrumentation



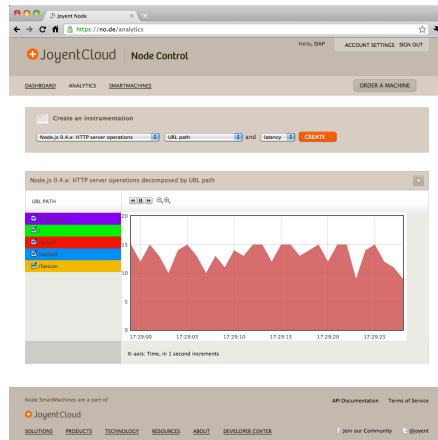
HTTP user/API request: create instrumentation



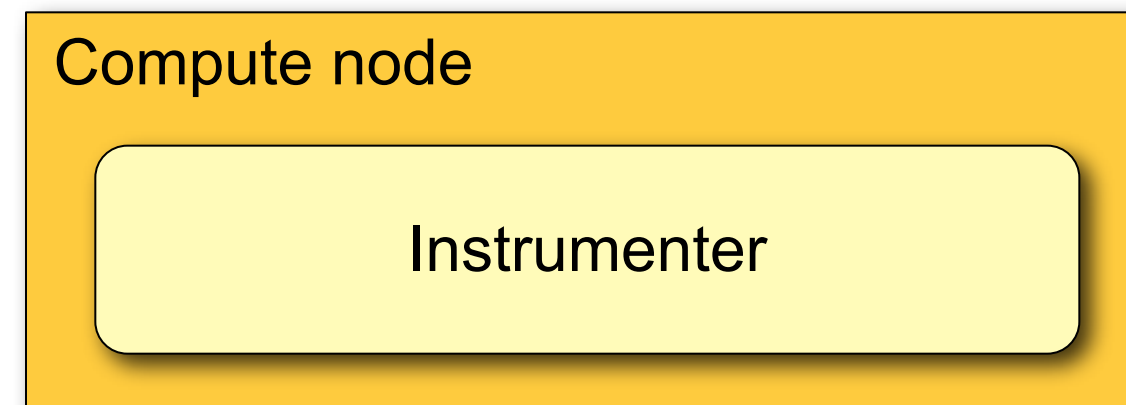
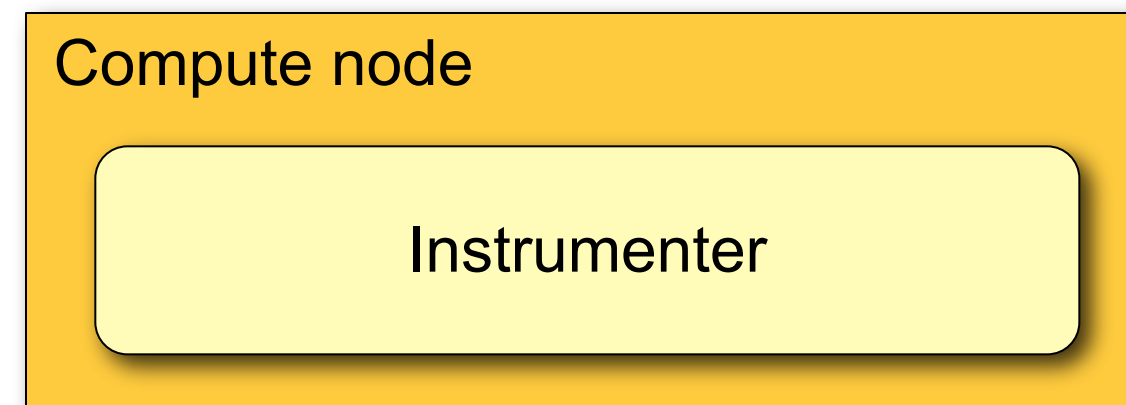
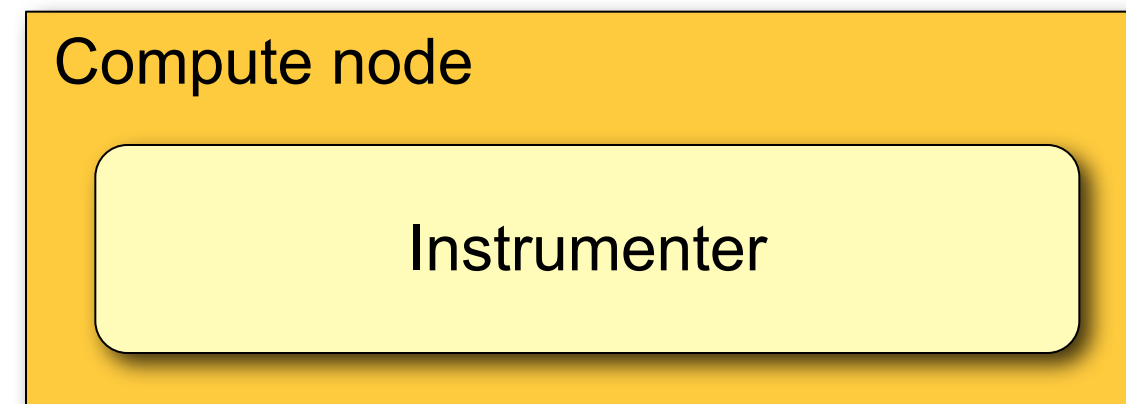
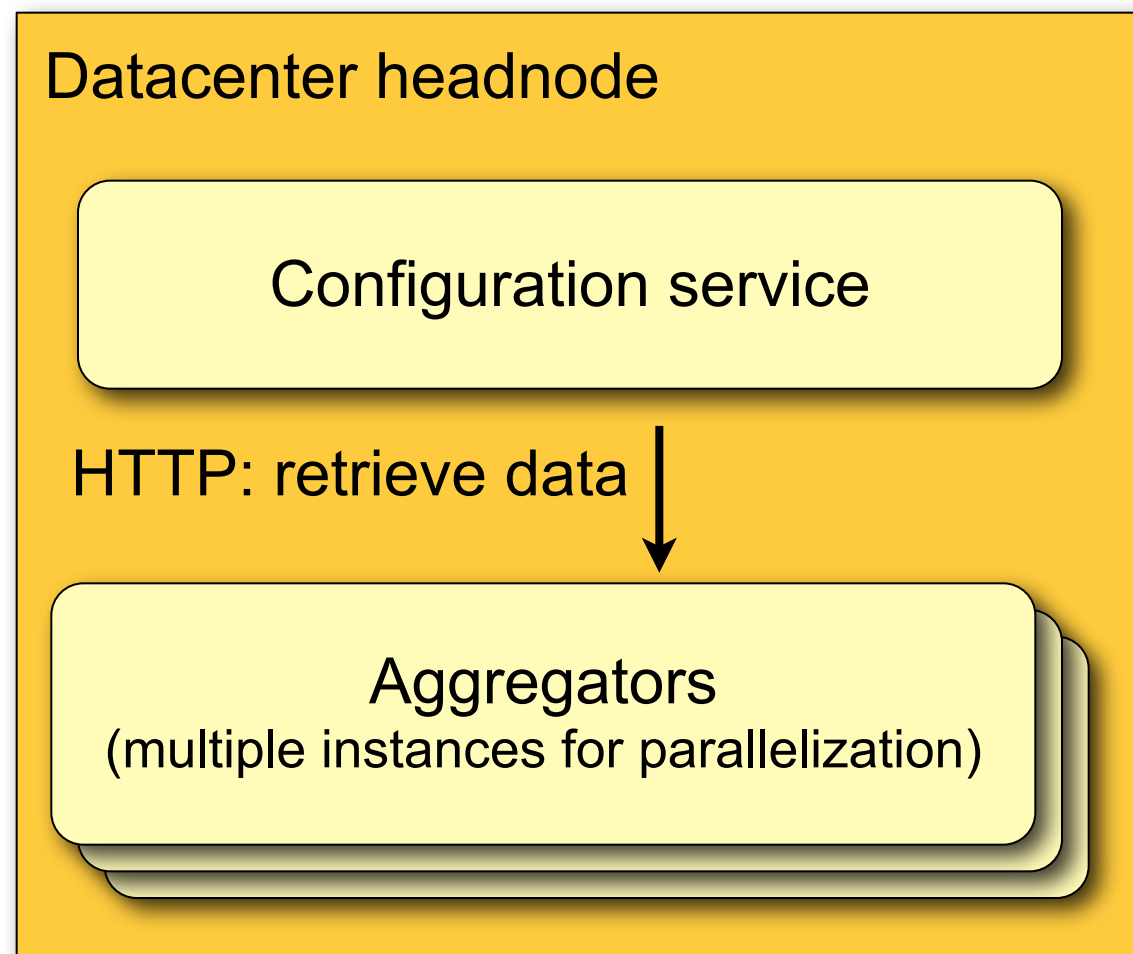
# Step 2: Instrumenters report data



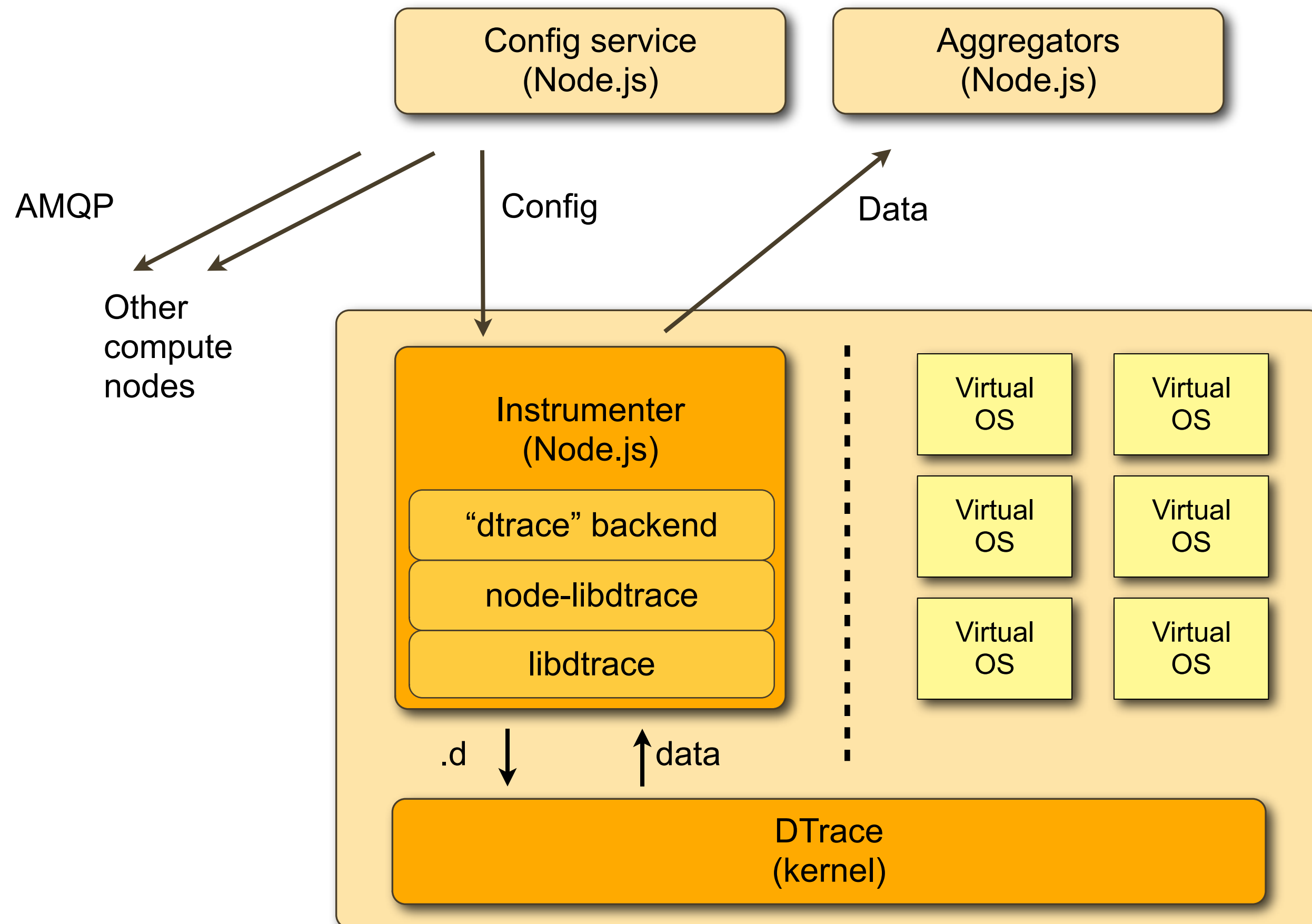
# Step 3: Users retrieve data



HTTP user/API request: retrieve data



# Inside the instrumenter



- AMQP daemon, pluggable backends (DTrace, kstat, ZFS, ...)
- Predefined metrics; each plugin registers implementations for each metric
- Backend interface:

`registerMetric(metric, constructor)` (Invoked by plugin)

`constructor(metric_info)` Initialize object based on metric, decomposition, predicate, etc.

`obj.instrument(callback)` Start collecting data (e.g., start DTrace).

`obj.deinstrument(callback)` Stop collecting data (e.g., stop DTrace).

`obj.value(callback)` Retrieve current data point (**invoked @ 1Hz**)

- Assemble a D script, compile it, and enable DTrace:

```
this.cad_prog = mdGenerateDScript(metric, ...)  
this.cad_dtr = new mod_dtrace.Consumer();  
this.cad_dtr.strcompile(this.cad_prog);  
this.cad_dtr.go();
```

- But how do you dynamically generate a D script to support predicates and decompositions?

- System calls

```
syscall:::return
{
    @ = count();
}
```

- System calls decomposed by application name and latency

```
syscall:::entry
{
    self->latency0 = timestamp;
}

syscall:::return
/self->latency0 != NULL/
{
    @[execname] =
        llquantize(timestamp - self->latency0, 10, 3, 11, 100);
}

syscall:::return
{
    self->latency0 = 0;
}
```

- Number of possible D scripts: exponential with number of possible decompositions
- Need way to automatically generate them

- Meta-D uses JSON to describe a family of D scripts differing in predicate and decomposition

```
{
  module: 'syscall', stat: 'syscalls', fields: [ 'hostname', 'zonename', 'execname', 'latency' ... ],
  metad: {
    probedesc: [ {
      probes: [ 'syscall::entry' ],
      gather: { latency: { gather: 'timestamp', store: 'thread' } }
    }, {
      probes: [ 'syscall::return' ],
      aggregate: {
        default: 'count()',
        zonename: 'count()',
        hostname: 'count()',
        execname: 'count()',
        latency: 'llquantize($0, 10, 3, 11, 100)'
      },
      transforms: {
        hostname: '"' + caHostname + '"',
        zonename: 'zonename',
        execname: 'execname',
        latency: 'timestamp - $0',
      }
    }
  ]
}
```

- Enable: hot-patches system call table entries (redirect into DTrace)
- Disable: revert system call table entries
- Advantages of dynamic tracing:
  - Instruments syscalls in all processes on the system at once
  - **The thread is never stopped**
  - Zero disabled probe effect

- Want to instrument MySQL commands by command name and latency:

```
mysql*:::command-start
```

```
{  
    self->command0 = lltostr(arg1);  
    self->latency0 = timestamp;  
}
```

```
mysql*:::command-done
```

```
/((((self->command0 != NULL))) && (((self->latency0 != NULL))))/  
{  
    @[self->command0] =  
        llquantize(timestamp - self->latency0, 10, 3, 11, 100);  
}
```

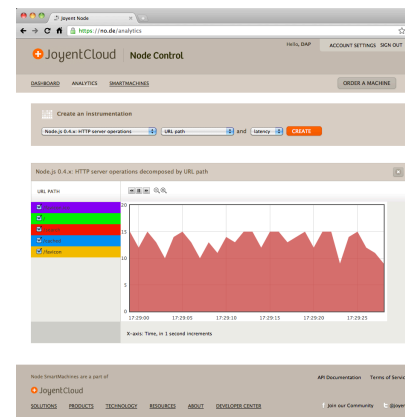
```
mysql*:::command-done
```

```
{  
    self->command0 = 0;  
    self->latency0 = 0;  
}
```

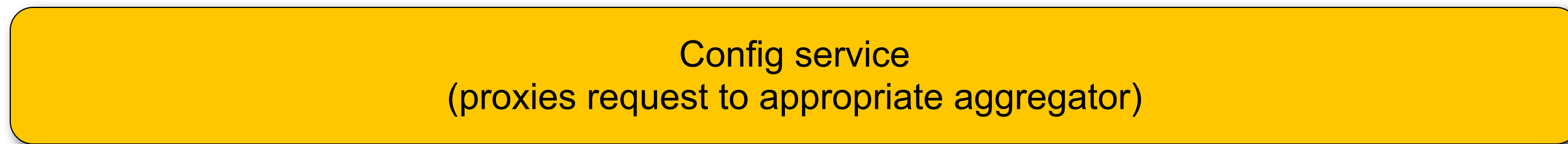
- Userland Statically-Defined Tracing (USDT): developer-defined static probes
  - e.g., `mysql*:::command-start`
  - maintained as functions and arguments evolve
- How it works:
  - In source, macro expands to DTrace function call
  - During link phase: function calls are replaced with nops and their locations recorded
  - On enable, DTrace replaces nops with trap
  - On disable, revert trap back to nop
- Thread is never actually stopped, but does take a round-trip to the kernel.
- **Zero disabled probe effect.**
- See also: pid provider
  - extremely powerful, but interface is unstable and requires instrumenting each process

- Combine userland and kernel tracing:
  - heatmap of total time spent in CPU dispatcher queue *per HTTP request*
  - heatmap of total time spent waiting for filesystem I/O *per MySQL query*
- Examine activity in all applications at once:
  - heatmap of filesystem latency *for all applications on a system, by application name*
  - *...and for all systems in a data center*
- Zero performance impact when not enabled, small impact when enabled
- No need to restart applications
- Can answer arbitrary performance questions **safely in production.**

- Bar charts: easy
  - Clients request raw data, render a chart
  - e.g., Total number of MySQL queries
- Stacked bar charts: easy
  - Clients request raw data for multiple separate data series, render a stacked chart.
  - e.g., Total number of MySQL queries *decomposed by zone name* (each virtual OS instance gets its own set of bars)
- Heatmaps: hard(er)
  - Heatmap contains a **lot** of raw data -- transferring it doesn't scale.
  - Render the heatmaps server-side
  - Rendering is compute-bound, but generally <40ms per heatmap (often more like 10ms).
    - We use multiple aggregators to parallelize the work.



Client request (HTTP)



...  
(other aggregators)

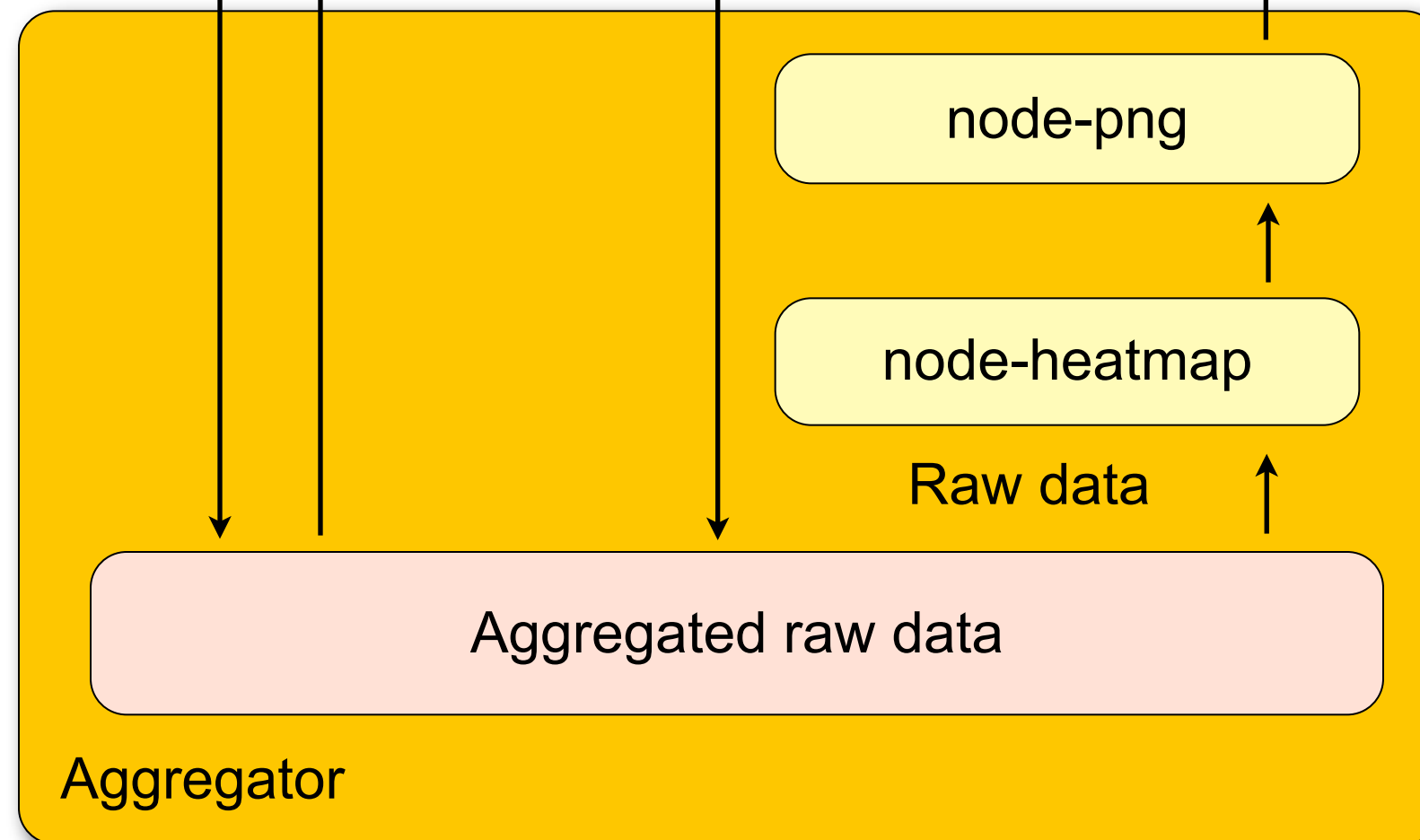
Raw data request

Raw data

Heatmap request

Heatmap

Raw data via AMQP  
(from all instrumenters)



The Problem

Cloud Analytics

**Demo**

Experiences

- <http://rm.no.de:8001/>

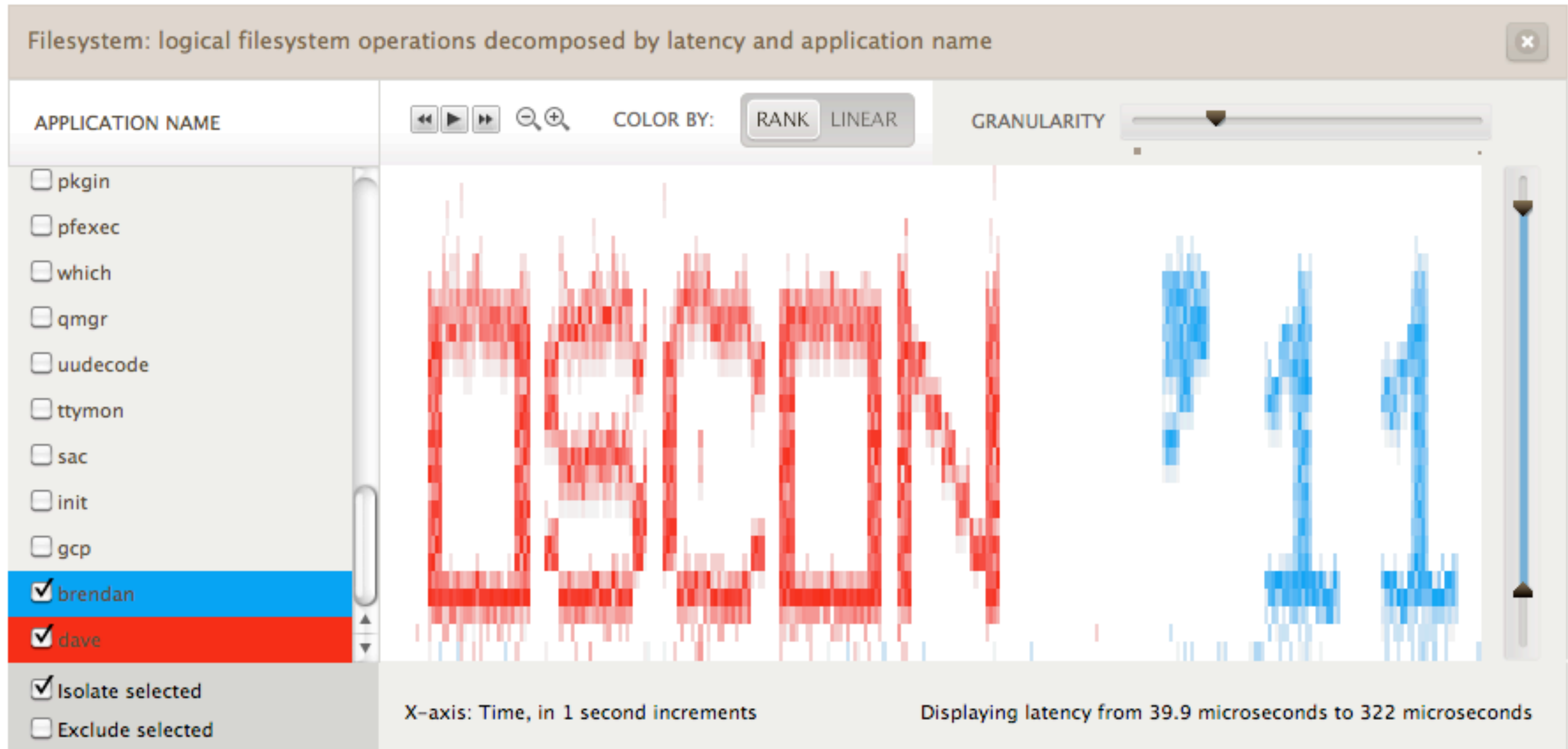
The Problem

Cloud Analytics

Demo

**Experiences**

- Node is solid:
  - All CA components are 100% Node.js (about 85% JavaScript, 15% C++)
  - Although aggregator is compute-bound, scaling it with multiple Node processes was easy
- Weak points:
  - C++ add-ons (no stable ABI and the failure modes are *not* crisp)
  - Diagnosing failures from the field (no post-mortem debugging)
- Building robust AMQP services is non-trivial (exclusive queue problem)



- Thanks!
  - Cloud Analytics: @dapsays, @brendangregg, @bcantrill, @rmustacc
  - Portal and API: @rob\_ellis, @notmatt, @kevinykchan, @mcavage
  - OS, Node teams at Joyent
- Check out our blogs at <http://dtrace.org/>

- “Instrumenting the real-time web: Node.js, DTrace, and the Robinson Projection”  
(Bryan Cantrill, <http://velocityconf.com/velocity2011/public/schedule/detail/18293>)
- “Breaking Down MySQL/Percona Query Latency with DTrace”  
(Brendan Gregg, <http://www.percona.com/live/nyc-2011/schedule/sessions/>)
- “Visualizing System Latency”  
(Brendan Gregg, <http://queue.acm.org/detail.cfm?id=1809426>)
- “Visualizations for Performance Analysis”  
(Brendan Gregg, <http://www.usenix.org/event/lisa10/tech/tech.html#gregg>)
- DTrace Book: <http://www.dtracebook.com/>
- Our blogs: <http://dtrace.org/>