Building a Real-Time Cloud Analytics Service with Node.js

Surge 2011

David Pacheco (@dapsays)

Bryan Cantrill (@bcantrill)

Last year at #surgecon...

Joyent

- Last year, we described the emergence of real-time data semantics in web-facing applications — a trend that we dubbed *data-intensive real-time* (DIRT)
- We discussed some of the ramifications of DIRT — among them the need to observe the stack in production in terms of *latency*
- After Surge 2010, we got to work on a facility to do this...
- The facility — cloud analytics — was first stood up as a production service at Joyent in March and shipped as a product in April
- Over the year, we have continued to deploy and improve it
- Cloud analytics is *itself* a DIRTy application; our implementation and our production experiences may inform decisions for other DIRTy apps

Agenda

Design objective

Architecture overview

Design choices

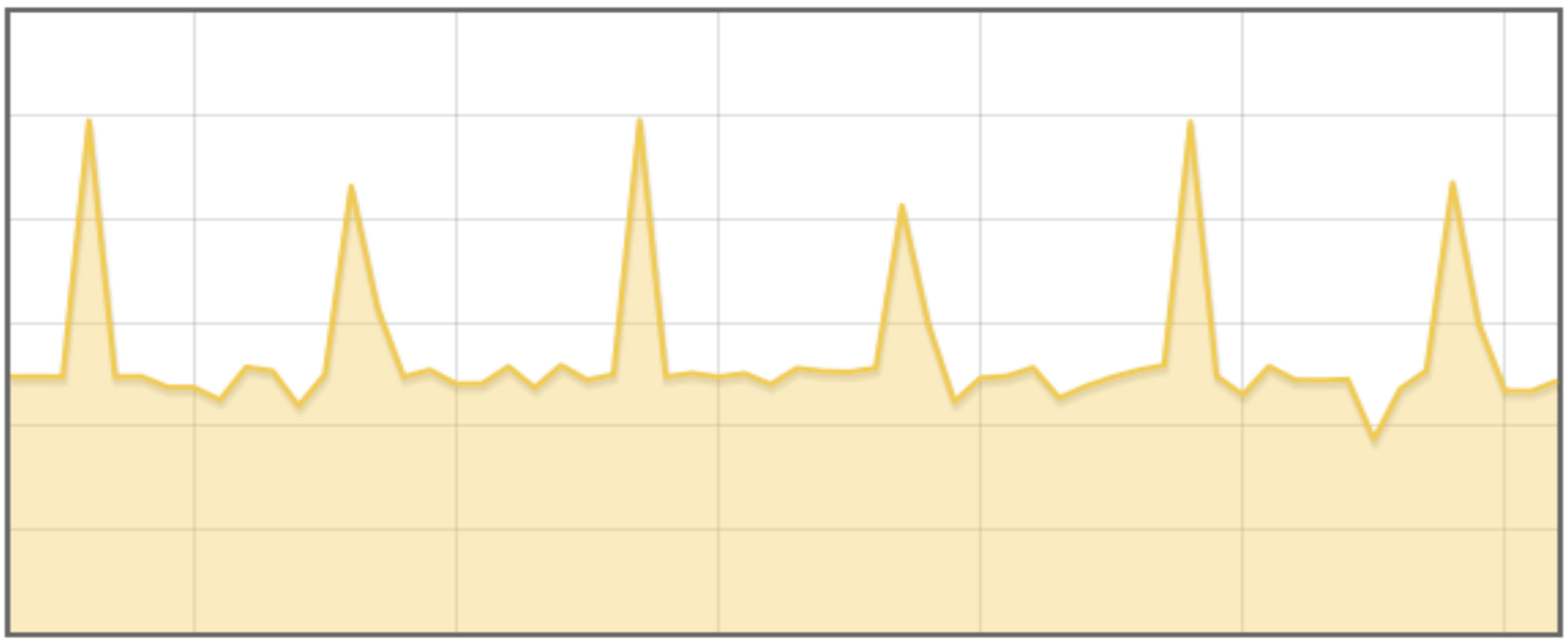
Production experiences

Design objective

- Need to focus on the 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
- Need to allow for *ad hoc* instrumentation
- Need to **summarize** the latency of thousands of operations — without losing critical detail
- Need to summarize that **across a distributed system**
- Need to do this **in real time**

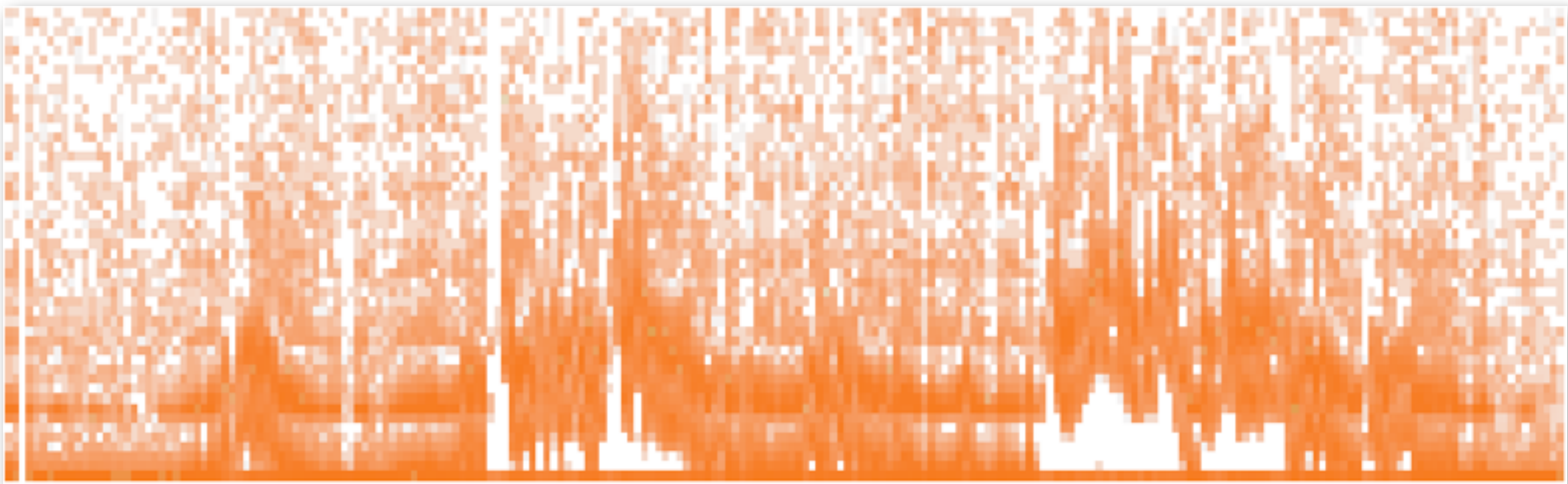
Visualizing latency as a scalar?

Joyent



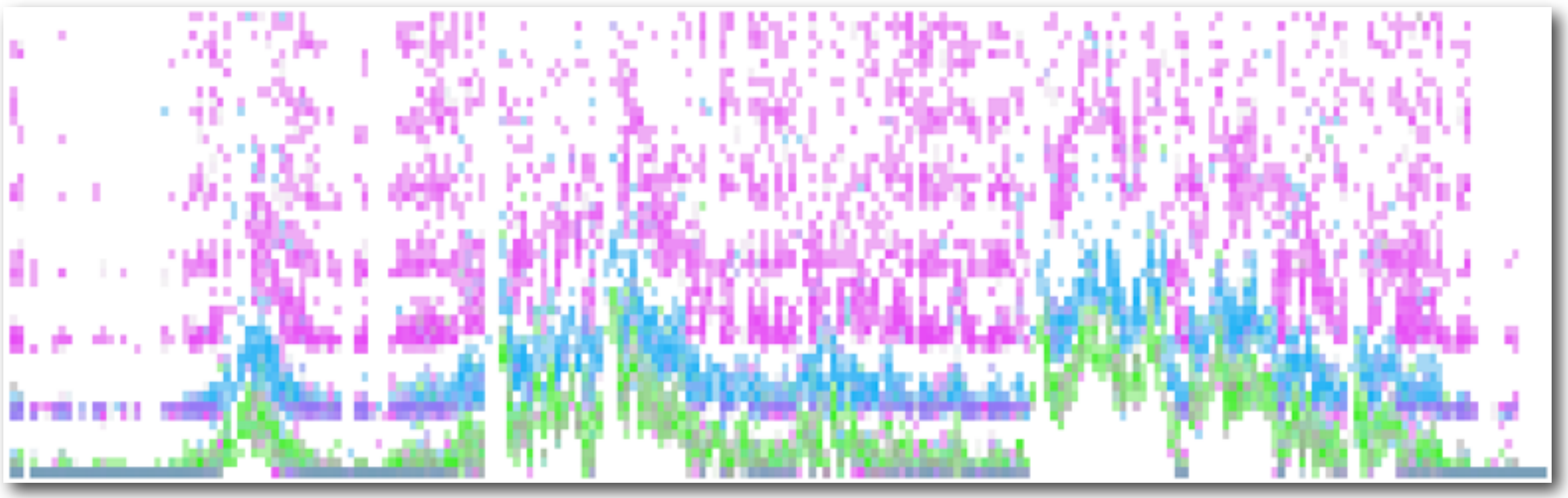
- Visualizing latency as a scalar (e.g., average) hides outliers — but in a real-time system, it is the outliers that you care about!
- Using percentiles is better, but still hides crucial detail

Visualizing latency as a heatmap?



- x-axis = time, y-axis = latency, **z-axis (color saturation) = count**
- Many patterns are now visible (as in this example of MySQL query latency), but critical detail is still missing

Visualizing latency as a 4D heatmap



- *Hue* can be used to express higher dimensionality
- x-axis = time, y-axis = latency, color saturation = count, **color hue = additional dimension** (database table in this example)

Agenda

Design objective

Architecture overview

Design choices

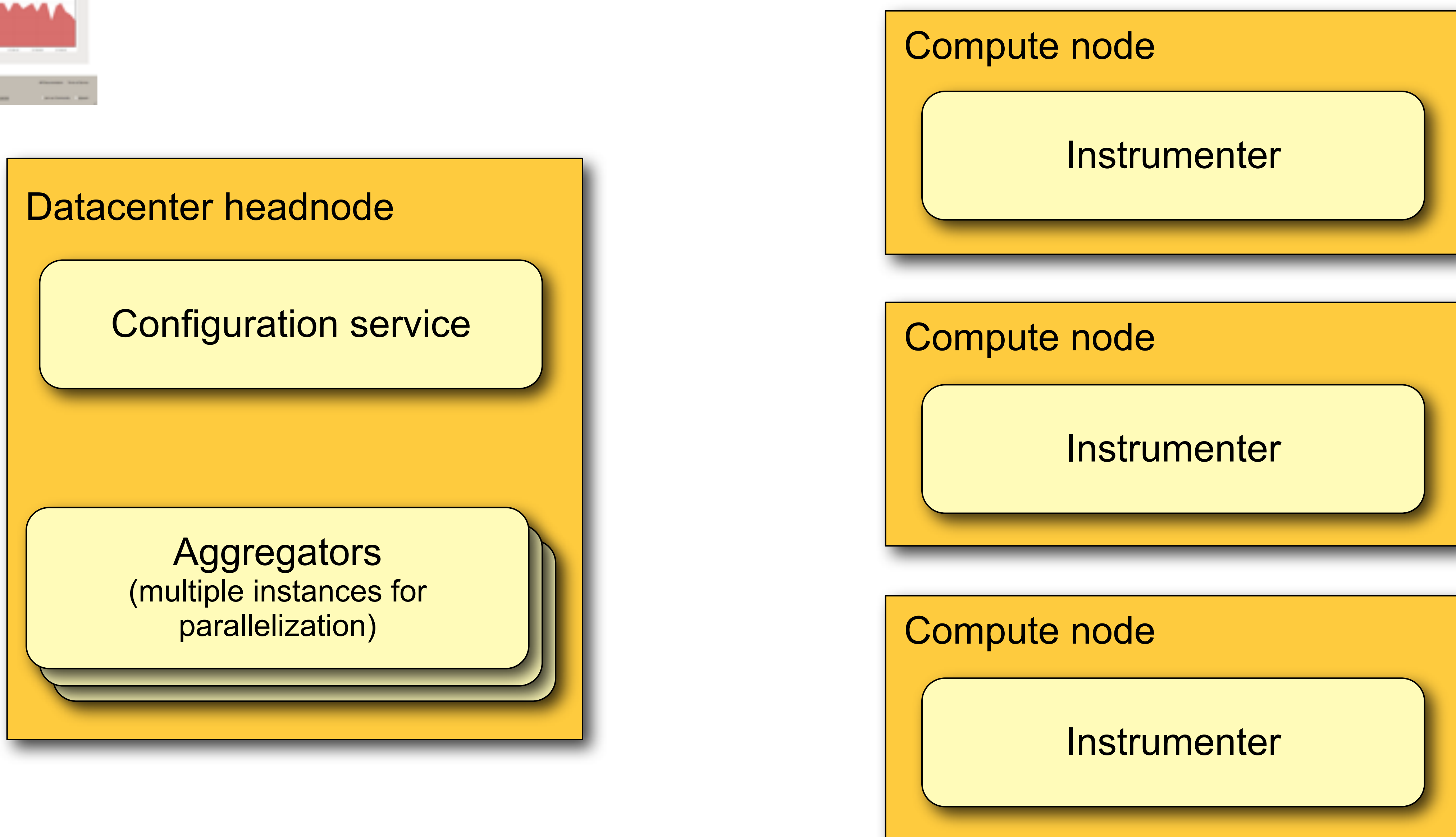
Production experiences

Architectural components

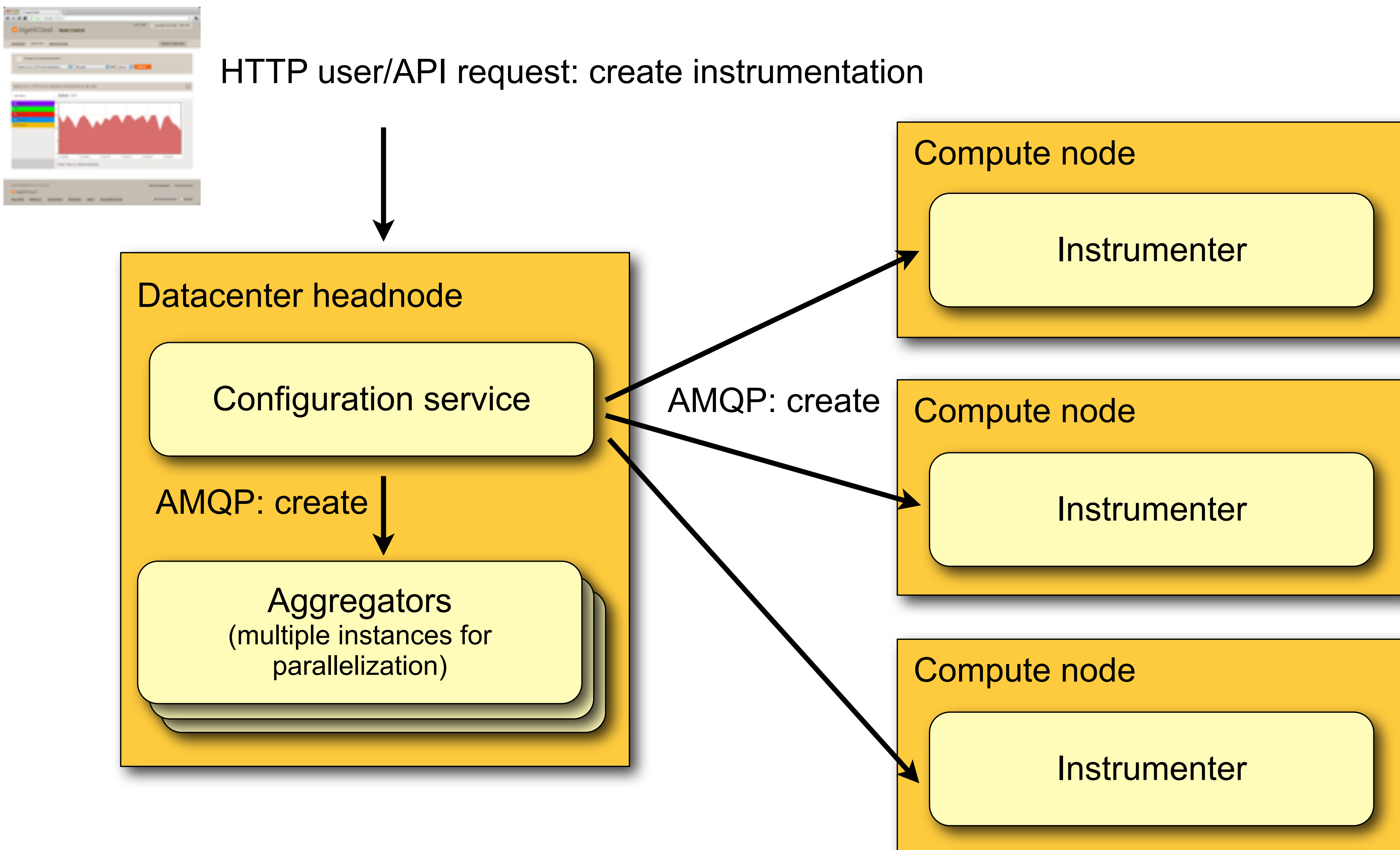
Joyent

- **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

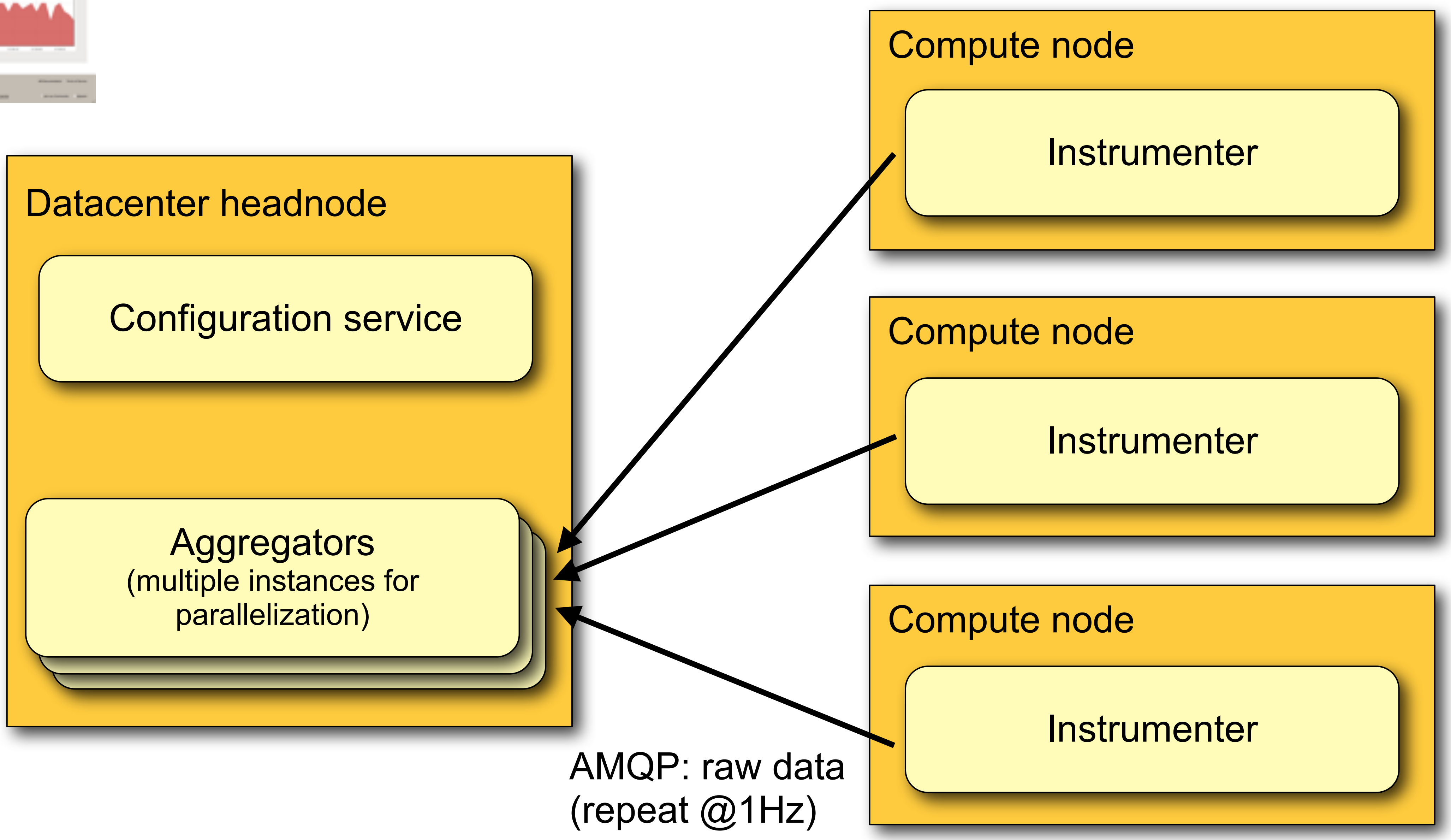
Architectural overview



Step 1: User creates an instrumentation

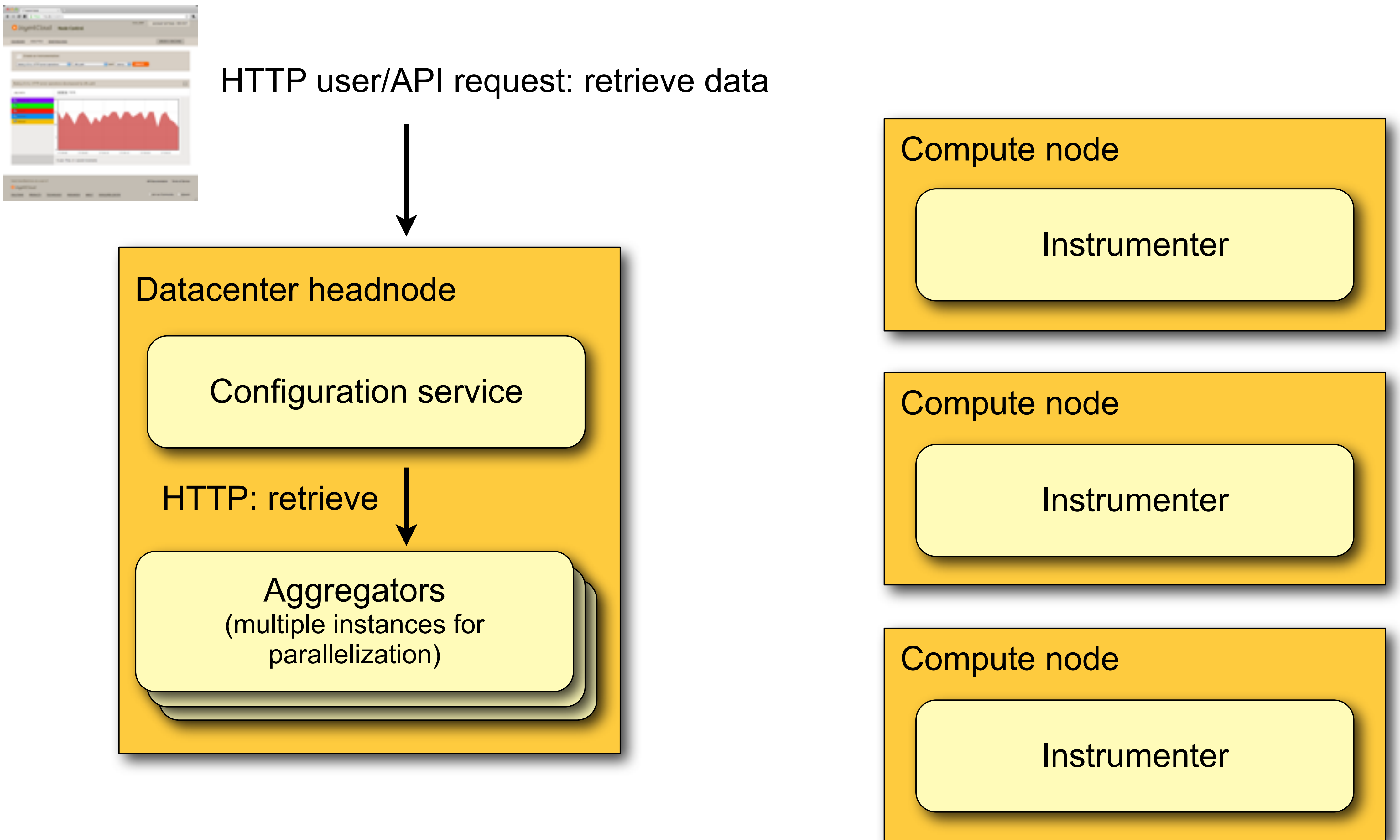


Step 2: Instrumenters report data

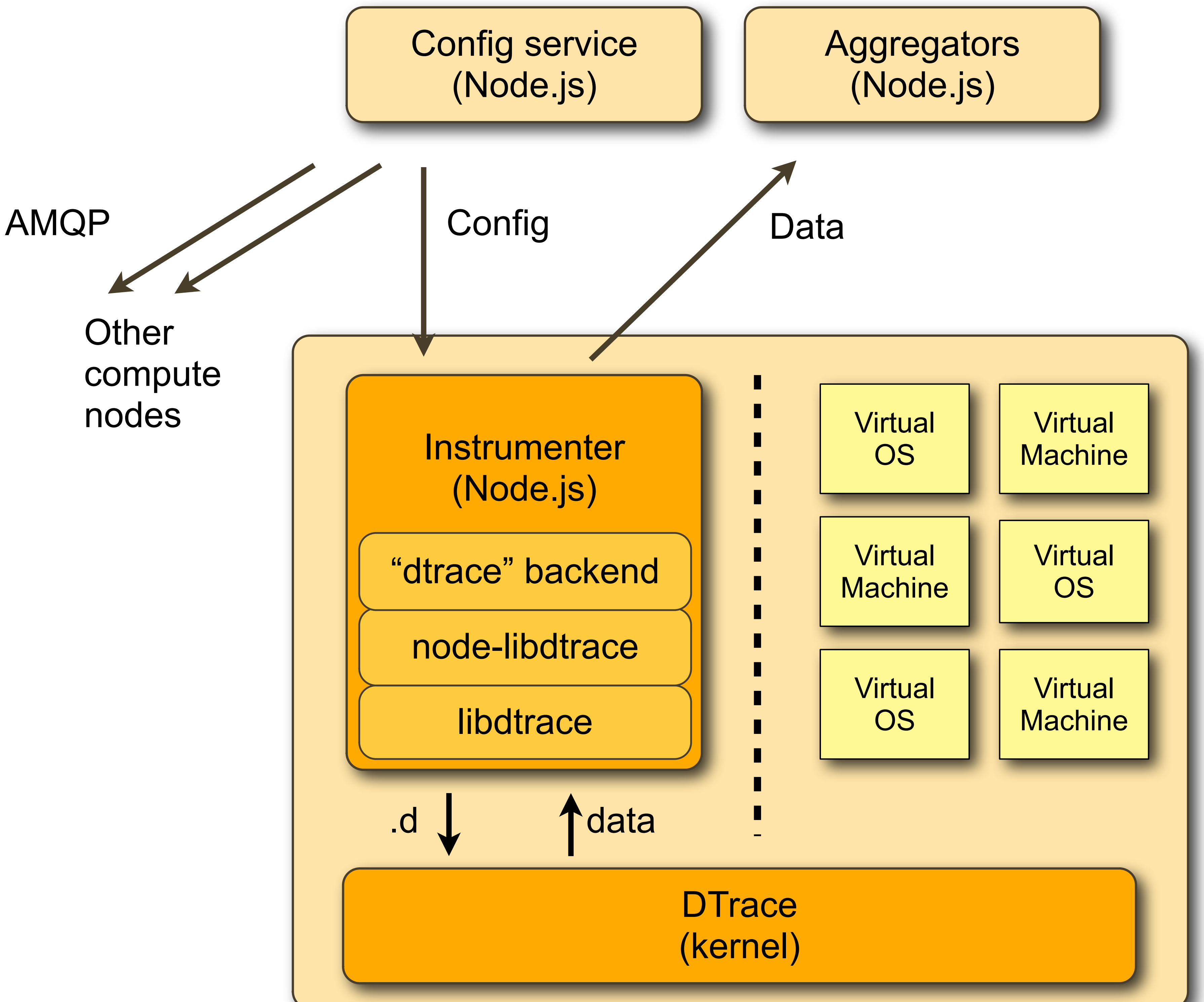


Step 3: Users retrieve data

Joyent



Inside the instrumenter



Agenda

Introduction

Architecture overview

Design choices

Production experiences

Node.js

- 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!');
```

The energy behind Node.js

- 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

Why Node.js

Joyent

- Our previous experience: building complex multi-threaded systems in C
  - Event-oriented model sounds pretty appealing
  - Event-oriented is possible in C, easier in Node.js
- Why Node.js:
  - **minimize latency** between gathering data and serving it to clients (especially in the face of service failure)
  - fast development
- Why not:
  - Poor observability (no pstack, dtrace, mdb, debugger)
  - Limited static analysis tools (compared to C compiler and lint)
  - No postmortem debugging
- At the very least, good choice for prototype.
- If it didn't work out, we wanted to know why.

Why AMQP

Joyent

- Why messaging?
  - Decouples system components
- Why AMQP?
  - Standard protocol with existing libraries, servers, and tools
- Why rabbitmq?
  - We were already using it elsewhere
  - Reputation of reliability and performance
- Why not?
  - Single broker = performance bottleneck
  - Wanted to **quantify** that before choosing a more complex architecture

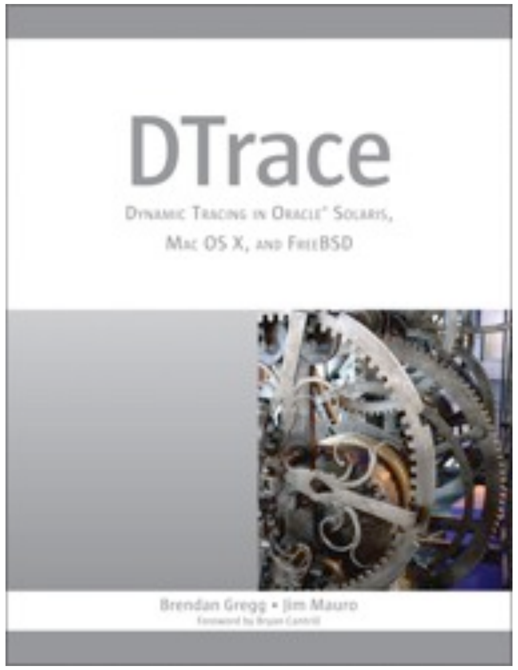
AMQP

Why HTTP/JSON

- Obviously: universal language for web APIs
  - Both browsers and Node.js have (mostly) first-class support for both HTTP and JSON
- But why not WebSockets?
  - Actually, why WebSockets? Usual answer: polling is inefficient
    - TCP connection overhead (obviated by HTTP keep-alive)
    - HTTP header processing (hard to imagine being a performance problem)
    - Extra request processing (not applicable to us)
  - Since our data is essentially continuous, buffered at 1-second intervals...
    - ... there's no "extra request" overhead. Polling is actually what we want.
  - Cons of WebSockets
    - Complexity
    - Observability (how do you measure server-side latency?)
    - Awkward model for historical (non real-time) data
  - We'd want to **quantify** the performance problem before introducing this complexity

Why DTrace

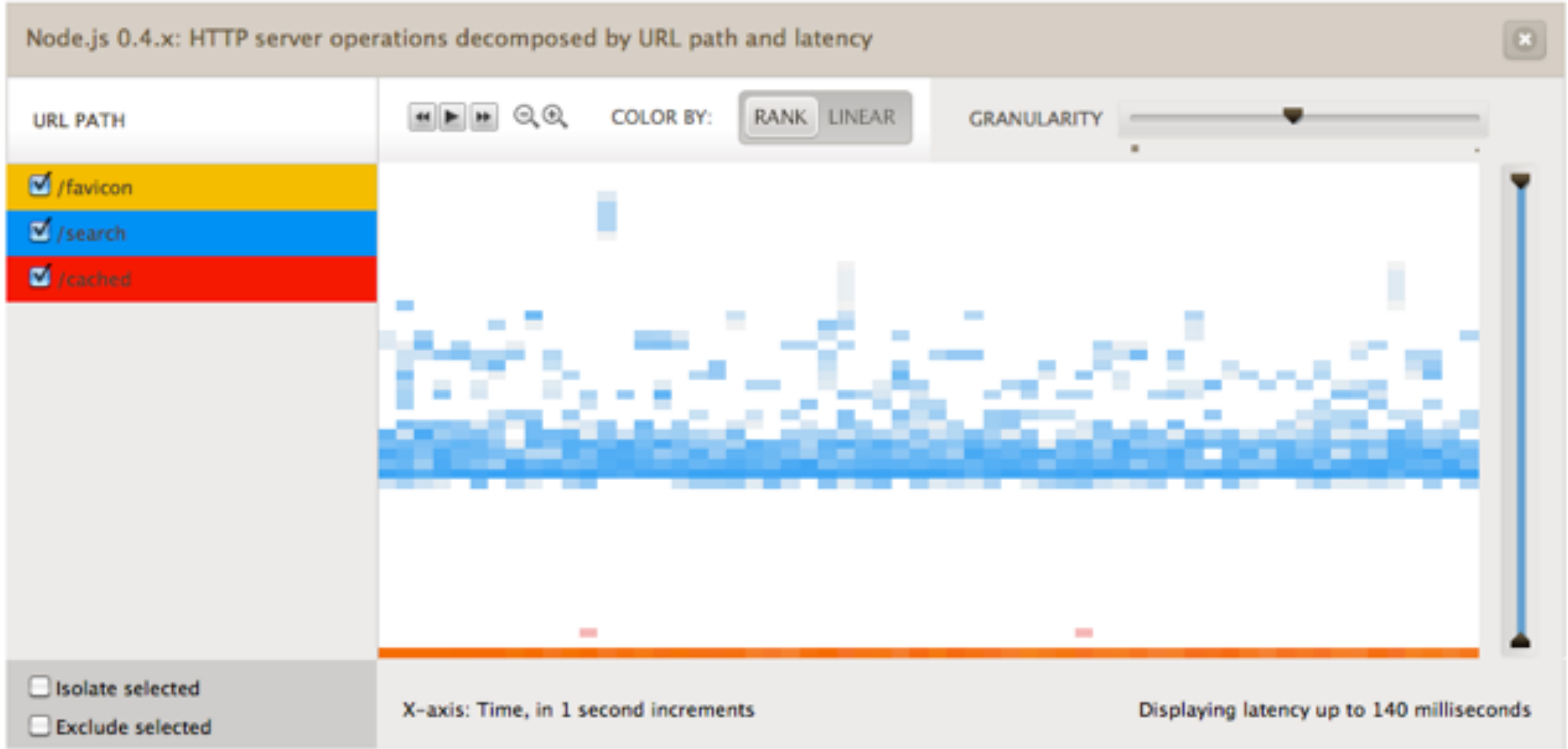
- Comprehensive tracing of both kernel and application-level events in **real-time**
- Scales arbitrarily with:
  - number of events (*in situ* aggregation)
  - number of customer instances (global visibility, OS-level virtualization)
- Suitable for production systems
  - Safe
  - Minimal overhead
  - Zero disabled probe effect
- Extensible via SDT, USDT
- (It's also the only game in town.)



Client-side vs. server-side rendering

Joyent

- Line graphs: client retrieves raw data, renders graphs using flot, d3, etc.
- Heatmaps: client retrieves heatmap image generated on-the-fly by the server
  - Con: lots of compute (requires parallelizing aggregators, but that's actually easy)
  - Con: makes rich interaction somewhat more difficult
  - Pro: heatmap is itself the most compact representation of the data



Agenda

Design objective

Architecture overview

Design choices

Production experiences

Problem: Node.js C++ add-ons

- We need Node.js add-ons (native extensions) for DTrace, kstat, libpng, ...
- Add-ons are written in C++, which has no stable binary interface
  - node and its add-ons must be built with the same compiler and version (or suffer nasty consequences!)
  - *Solution*: CA delivers a bundle with “node” plus binary add-ons
- WAF-based build process is easy to get wrong
  - e.g., build process looking in wrong place for header files
  - e.g., binaries built without links to dependent libraries (fail at runtime)
  - All we can do is fix these problems when we run into them, but it can be painful.

Problem: Node.js limits

- Each aggregator's load could be limited by size of the Node heap
- Each aggregator's load could be limited by 1 CPU (heatmap generation)
- *Solution*: parallelize workload at instrumentation level
  - Spin up "ncpus" aggregators
  - Each new instrumentation gets assigned randomly to one aggregator, which stores the data and services all requests for raw data and heatmap
  - Config service proxies HTTP requests to the appropriate aggregator

Problem: Node.js observability

Joyent

- Hard to figure out what a program is doing (or did do)
- *Solutions*: we built several tools to help with this:
  - cactl: uses AMQP to ping, status-check, or summarize the state of all CA services
  - amqpsnoop: watch all AMQP messages, or filter by arbitrary criteria (works only for messages on topic exchanges)
  - node-panic: primitive postmortem debugging for Node.js
    - When a server crashes or does the wrong thing, it **must** be possible to dump all state immediately so you can restart the service and debug later
    - "cactl" can also send the command to panic via AMQP
- We also use snoop and Wireshark to understand network traffic

Problem: observing spinning programs

- Shortly after first production deployment, we found one of the aggregators spinning
  - Not responding to AMQP or HTTP, not invoking system calls
  - pstack showed it was running JavaScript, but we had no way of seeing what it was running
  - No event loop => couldn't trigger panic via AMQP
  - No event loop => couldn't use SIGUSR1 to start the debugger agent
- Several ways to improve this:
  - *Mitigation*: Randomize aggregator selection to mitigate failure mode
  - *Solution*: Change Node.js SIGUSR1 to open debugger port immediately
  - *Solution*: Created "ncore" tool as part of node-panic to use SIGUSR1 to generate dump (including stacktrace!) of program stuck in infinite loop
  - *Solution* (future): jstack() DTrace action
- Scary part: we haven't ever seen this problem since.

Problem: synchronous DTrace enabling

- DTrace can take several seconds to enable probes on a system
- Currently, this operation is synchronous in node-libdtrace, so instrumenters report no data while this is going on
- Challenging to make this async because libdtrace only supports one concurrent compile at a time due to yacc limitation (!)
- *Solution*: eio_custom() and asynchronous interface

Node: The Good Parts

- Development *was* fast:
  - Time to functional CA prototype: 2 weeks
  - Time to production for CA: 4 months
  - The prototype evolved significantly, but was never thrown out
- CPU, memory usage have **not** been a problem for aggregators or configsvc.
- Events (e.g., HTTP request) typically shown on screen within 2-3 seconds
  - Raw value requests served within a few milliseconds
  - Heatmap requests served around 50-75ms
  - Component failures do not result in latency bubbles for everyone else
- Tools have given us adequate visibility into service status (and where they haven't, we've built more tools)

Problem: AMQP exclusive queues

Joyent

- AMQP allows queues to have an exclusive consumer, enforced by the broker
- What happens when that consumer crashes?
- What happens when that consumer's system crashes?
  - Broker has no way of knowing.
  - On restart, the consumer is rejected from its own queue.
- *Possible solution*: AMQP heartbeating (requires client support)
- *Solution*: when consumer sees RESOURCE_LOCKED error, it pings itself, waits a while, and tries again.
- Note: without AMQP, we'd instead have problems managing connections to multiple components claiming to be the same service.

Problem: AMQP connectivity

- Components can get disconnected from the broker
  - network failure, broker failure, server failure, or even **configuration change**
- Components must handle this while in the middle of sending data
  - *Solution*: arbitrary "write" operations can fail with "socket disconnected" errors
  - node-panic was crucial for understanding Node.js Socket state in these cases
- Components must detect this while idle
  - *Possible solution*: AMQP heartbeating (requires client support)
  - *Solution*: each component periodically pings itself
- Components must keep trying to reconnect
  - and what do we do with messages sent in the meantime?
- Note: these problems exist with direct connections, too.

Problem: RabbitMQ performance with many bindings

Joyent

- During first (largest) major production deployment, rabbitmq lost its mind
  - 90+% CPU utilization (on a 16-way box)
  - Forever-increasing memory utilization (upwards of 400MB) **while queue lengths all zero**
    - No visibility into “dark queue” of internal work
- Spent over a week trying to reproduce in development
  - Eventually reproduced by creating 1500+ bindings on a topic exchange and sending about 100 messages per second.
- *Mitigation*: use rabbit’s management API to build monitoring tools
- *Possible solution*: upgrade rabbitmq to 2.4.0 or later for “fast topic routing”
- *Solution*: use “direct” exchange rather than “topic” exchange
  - (breaks amqpsnoop)

AMQP: The Good Parts

- Per-component configuration is trivial: just needs the broker IP
- Routing key abstraction simplifies failure modes around component crashes
- With the topic routing issue worked around, rabbitmq has easily handled as much traffic as we've thrown at it **with low (enough) latency** (~100ms)
- With the glaring exception of internally queued work, rabbit provides good observability into the state of the distributed system
  - e.g., message traffic on queues and channels
  - e.g., bindings and channels associated with each queue

Summary

- On the most important early decisions (Node.js, AMQP/RabbitMQ, HTTP/JSON, DTrace), we haven't regretted any of these choices.
- Many of the problems were not specific to these technologies
  - Observability: a problem with just about everything but C
  - Network failure: a problem whether using AMQP or direct connections
  - Such limitations can be overcome (by building new tools and fixing the software)
- Some of these were inherent limitations ...
  - Node.js scaling past 1 thread (but that was very easy to work around in our case)
- Still believe it has been and will be much easier to address these problems than to make the alternatives work
- Overall goal is met: visualizing performance data in real-time
- Demo on production system or GTFO!

Appendix

References

- Tools
  - node-panic: https://github.com/joyent/node-panic
  - amqpsnoop: https://github.com/davepacheco/node-amqpsnoop
  - javascriptlint: https://github.com/davepacheco/javascriptlint
  - jsstyle: https://github.com/davepacheco/jsstyle
- Cloud Analytics
  - http://dtrace.org/blogs/dap/2011/03/01/welcome-to-cloud-analytics/
  - http://dtrace.org/blogs/bmc
  - http://dtrace.org/blogs/brendan
  - http://dtrace.org/blogs/rm