CHICAGO
INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE 2016

goto; conference

From Monolith to Microservices

Tony Maher

Dose Media

- www.dose.com
- www.omgfacts.com

- Tony Maher, Team Lead
    - tony@dose.com, @tonymaher5

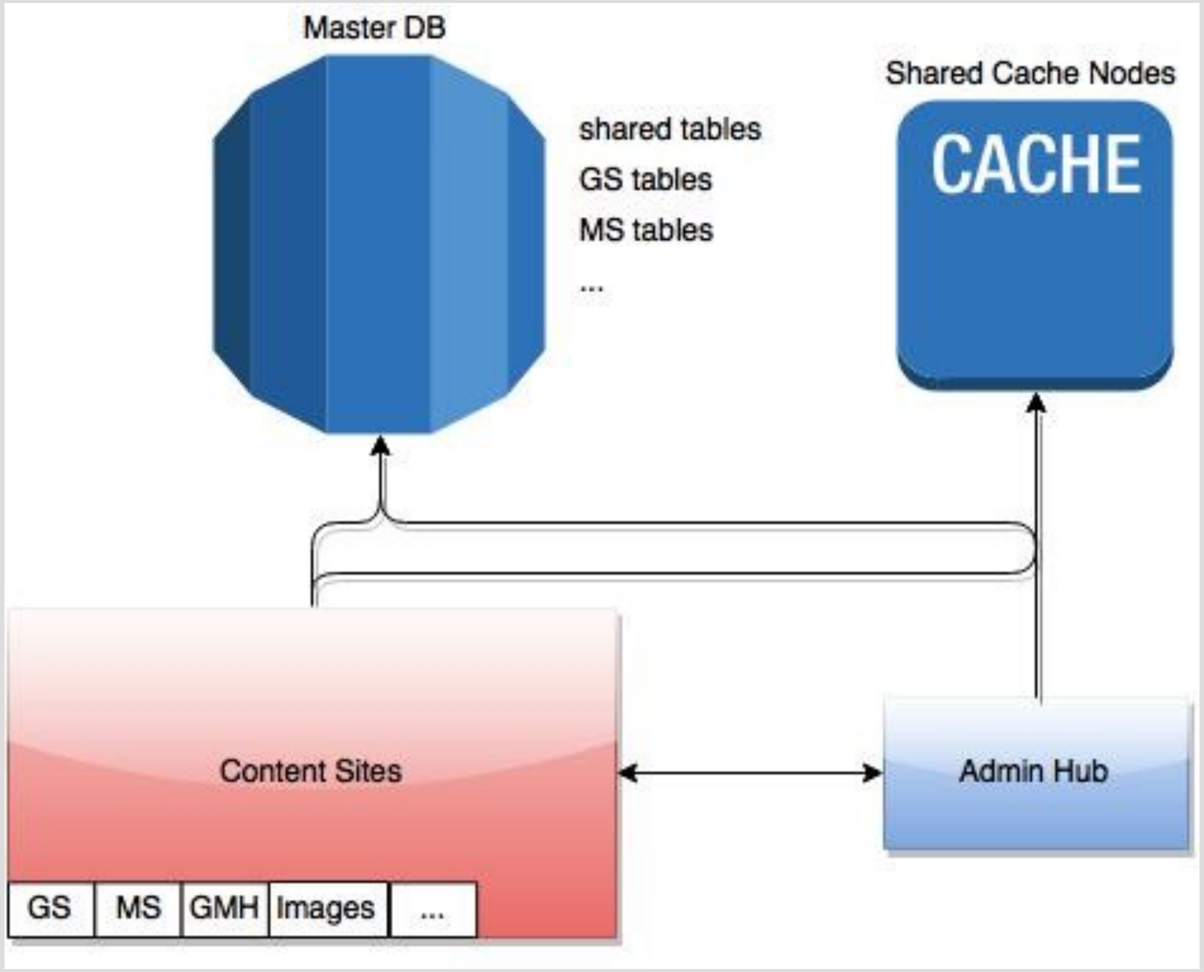
Dose Media

- Large scale websites - 55 million monthly uniques.
  - Uptime is paramount.
- Small, autonomous, and flexible teams.

Outline

- Dose Architecture, 2012-2014
- Results of this architecture
- The start of Microservices and steps we started taking
- Initial Results
- Introduction of Docker
- Current Status
- Next Steps

Dose Legacy Architecture



Dose Legacy Architecture

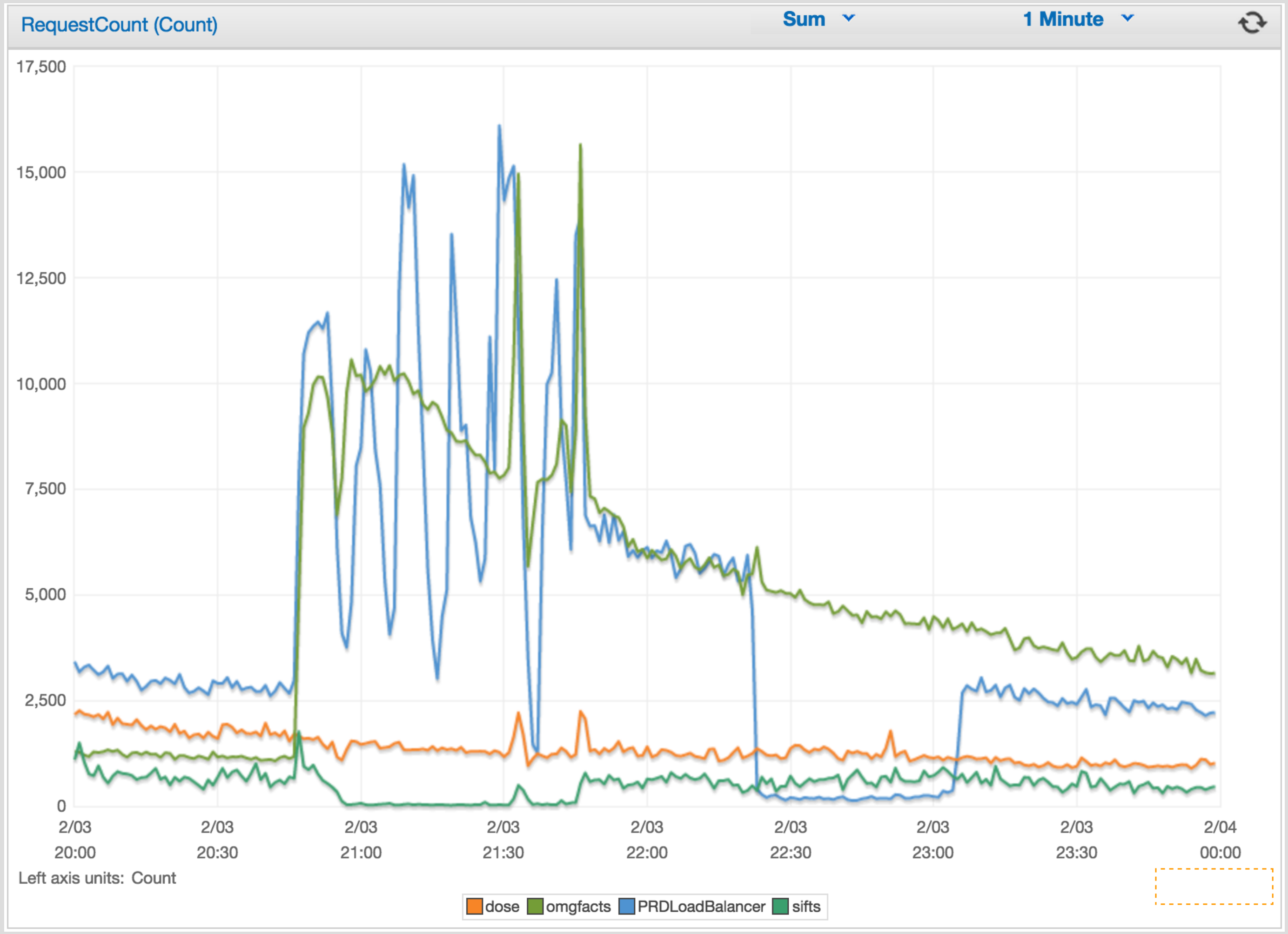
- ~3 million lines of code
- 415 commits to production in Q1 2013, most of which were bug fixes or red alert bandaids.

Dose Legacy Architecture

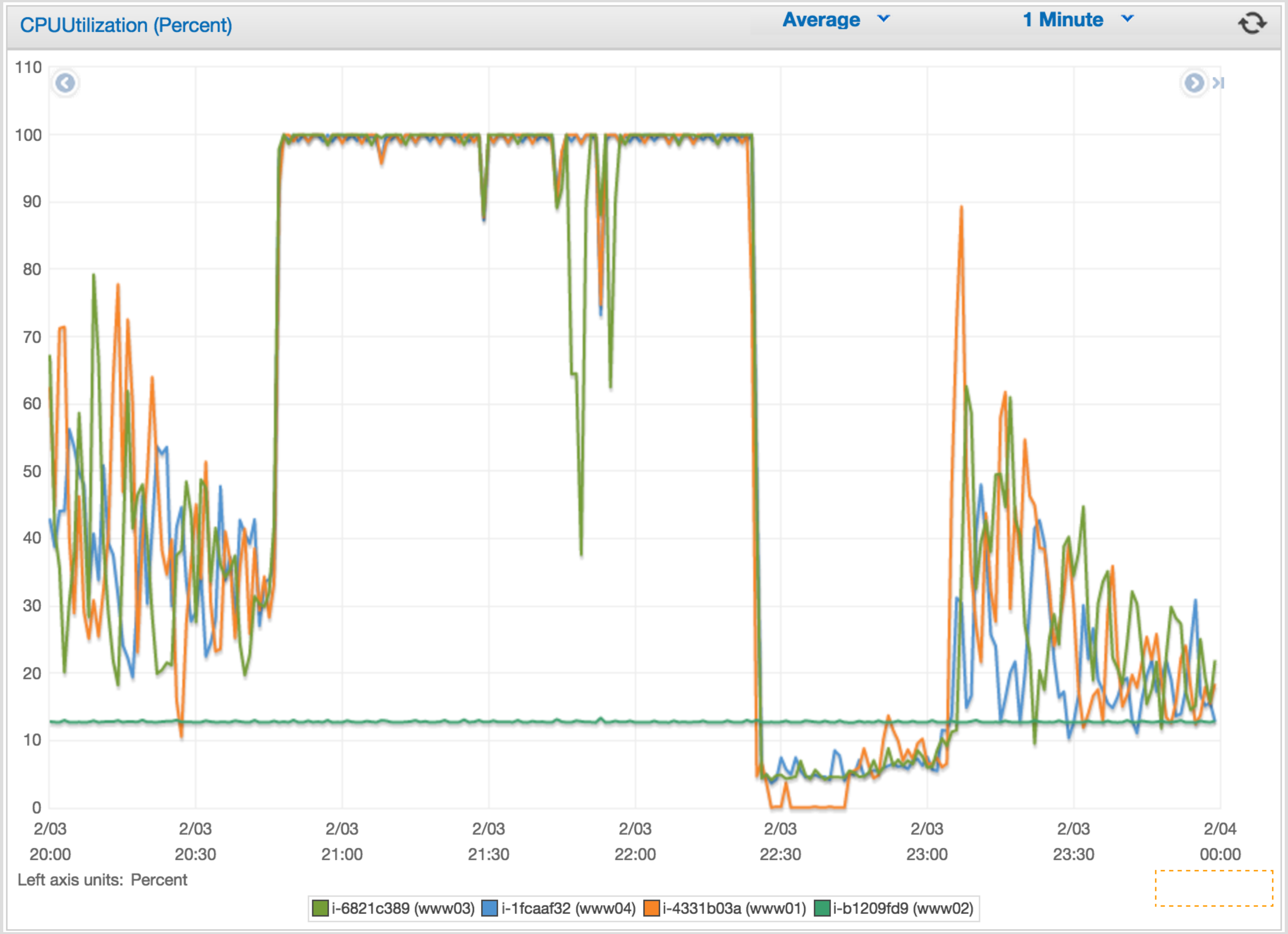
- Response Time: ?
- Downtime: ?

- Horrible/missing monitoring, logging, documentation

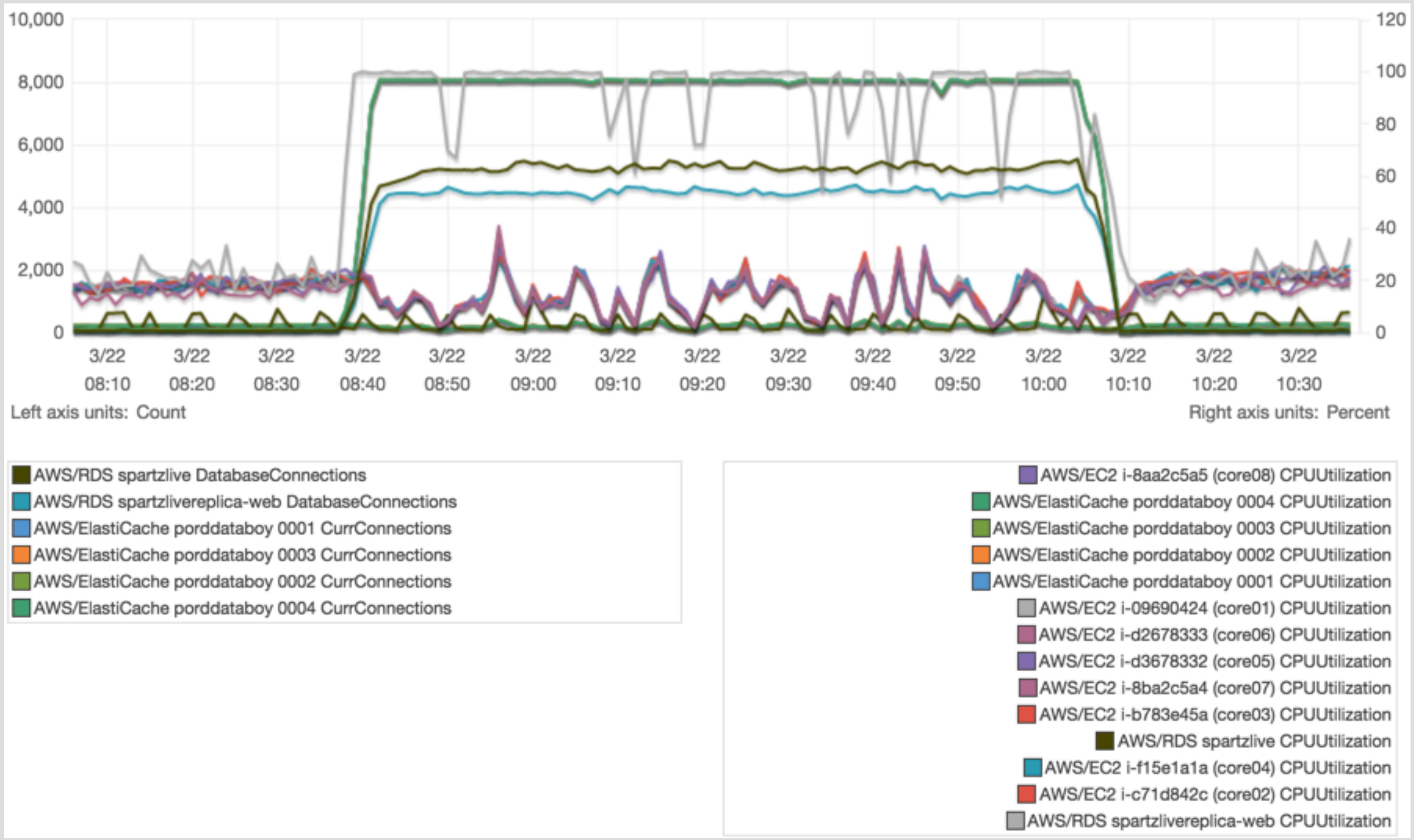
Symptoms


RequestCount (Count)
Sum
1 Minute
17,500
15,000
12,500
10,000
7,500
5,000
2,500
0
2/03 20:00
2/03 20:30
2/03 21:00
2/03 21:30
2/03 22:00
2/03 22:30
2/03 23:00
2/03 23:30
2/04 00:00
Left axis units: Count
dose
omgfacts
PRDLoadBalancer
sifts

Symptoms


CPUUtilization (Percent)
Average
1 Minute
110
100
90
80
70
60
50
40
30
20
10
0
2/03 20:00
2/03 20:30
2/03 21:00
2/03 21:30
2/03 22:00
2/03 22:30
2/03 23:00
2/03 23:30
2/04 00:00
Left axis units: Percent
i-6821c389 (www03)
i-1fcaaf32 (www04)
i-4331b03a (www01)
i-b1209fd9 (www02)

Symptoms



10,000
8,000
6,000
4,000
2,000
0
120
100
80
60
40
20
0
3/22 08:10
3/22 08:20
3/22 08:30
3/22 08:40
3/22 08:50
3/22 09:00
3/22 09:10
3/22 09:20
3/22 09:30
3/22 09:40
3/22 09:50
3/22 10:00
3/22 10:10
3/22 10:20
3/22 10:30
Left axis units: Count
Right axis units: Percent
AWS/RDS spartzlive DatabaseConnections
AWS/RDS spartzlivereplica-web DatabaseConnections
AWS/ElastiCache porddataboy 0001 CurrConnections
AWS/ElastiCache porddataboy 0003 CurrConnections
AWS/ElastiCache porddataboy 0002 CurrConnections
AWS/ElastiCache porddataboy 0004 CurrConnections
AWS/EC2 i-8aa2c5a5 (core08) CPUUtilization
AWS/ElastiCache porddataboy 0004 CPUUtilization
AWS/ElastiCache porddataboy 0003 CPUUtilization
AWS/ElastiCache porddataboy 0002 CPUUtilization
AWS/ElastiCache porddataboy 0001 CPUUtilization
AWS/EC2 i-09690424 (core01) CPUUtilization
AWS/EC2 i-d2678333 (core06) CPUUtilization
AWS/EC2 i-d3678332 (core05) CPUUtilization
AWS/EC2 i-8ba2c5a4 (core07) CPUUtilization
AWS/EC2 i-b783e45a (core03) CPUUtilization
AWS/RDS spartzlive CPUUtilization
AWS/EC2 i-f15e1a1a (core04) CPUUtilization
AWS/EC2 i-c71d842c (core02) CPUUtilization
AWS/RDS spartzlivereplica-web CPUUtilization

Symptoms

- Frequent crashes.
- Problem in one area ballooned to problems across entire network.
- HUGE resource costs for relatively simple functionality.
- Extremely long debug times.

Dose Legacy Architecture

- Response Time: ?
- Downtime: ?

- We know they were reeeeaaaally bad.

- Best guess would be around 1-2 seconds server response time, <99.0% uptime.

The Start of Microservices

- Full team buy-in… No edicts from the mountaintop.
- Multiple microservice related book clubs.
- Regular architecture planning meetings.

The Start of Microservices

- The general concept: **It’s simply good OO class design, abstracted to services and applications.**
- SOLID Design Principles… SRP

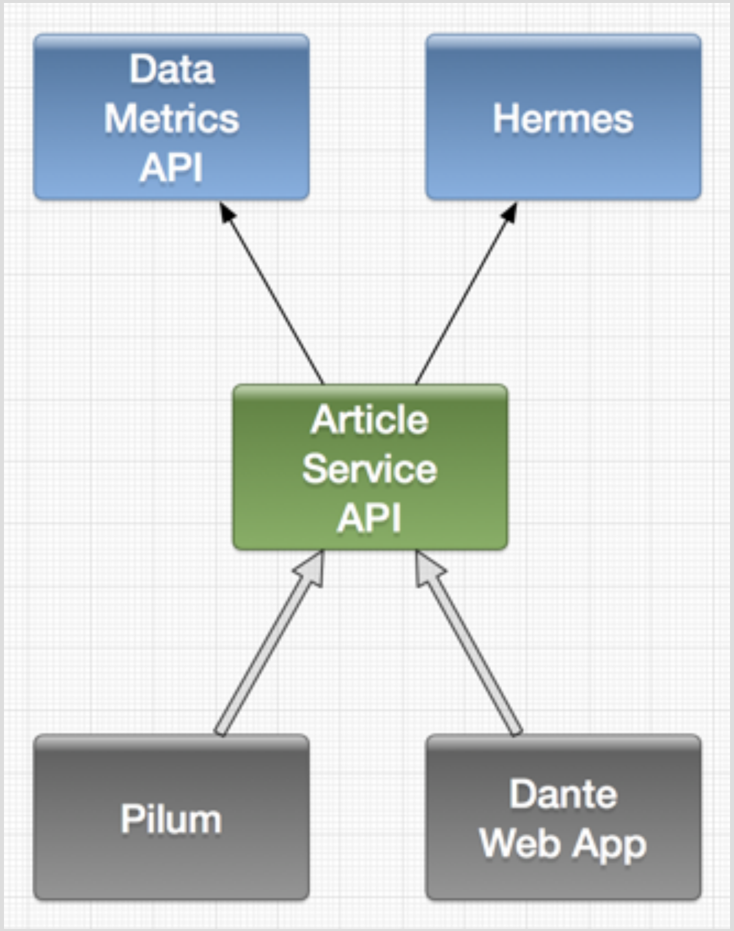
The Start of Microservices

- Just little things at first…
  - Keep modules small and isolated.
  - No God Classes.
  - USE INTERFACES.
- Split distinct functionality into separate codebase, with its own web cluster, database, cache, everything.

The Start of Microservices

- Resource APIs - Data based microservices which just expose related resources/entities.
- Service APIs - Behavior based microservices which communicate with Resource APIs.
- Client Applications - Web and mobile apps which only communicate with Service APIs.

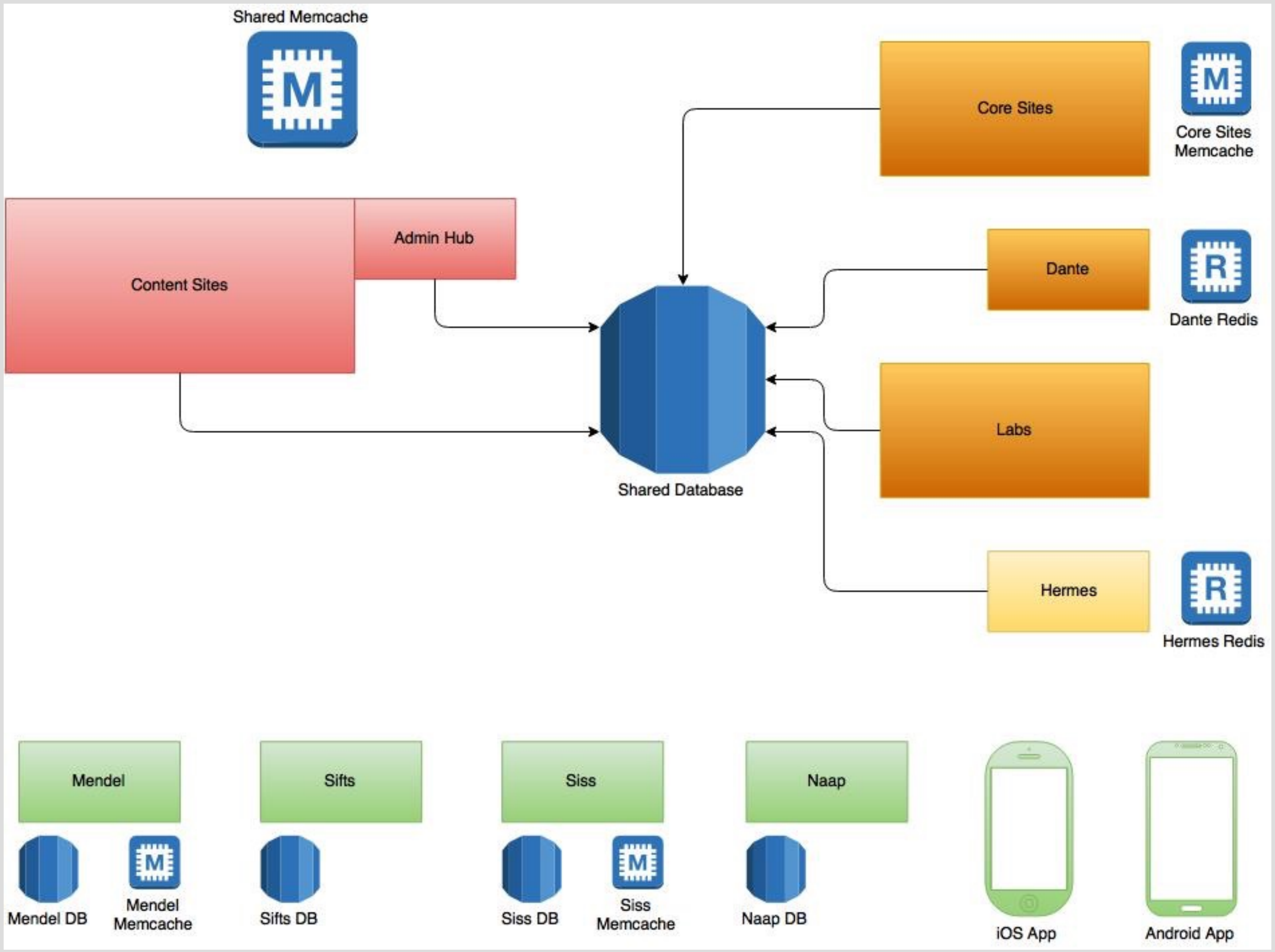
The Start of Microservices



The Start of Microservices

- When we wanted to split a service, but too much of its functionality was still tied directly to the functioning monolith…
    - We created what we call a “hydra” - shared resources (usually the monolithic database) used by separate services.

The Start of Microservices



The Start of Microservices

- At this point (late 2014 - early 2015), we’ve got a couple Microservices, a couple Hydras, and the legacy monolith.

- Already we see vast improvements though.

Initial Results

- Average server response time: 800ms
- Uptime: 99.96%

- Still not great, but at least we’re heading in the right direction.

State of Devops

- Local development was done in a vagrant box, which may or may not have had the same versions as qa or prod, build or batch nodes.

- Updates were run on a long running “build” node with a potentially very different architecture than other environments.

State of Devops

- Toss it over the wall attitude toward devops and deployments…

“It works locally so must be a devops problem”

State of Devops

- So we identified our needs:
  - **Consistency** between environments.
  - **Transparency** on project requirements and dependencies.
  - **Flexibility** to change as we experiment and learn more about the process and how it works for us.

Introduction of Docker

- We started using **docker-compose**, as well as semi-regularly updated docker base images, to ensure consistent architectures across environments.
- Moved configuration and deployment closer to development.
- Put everything you need to run an environment in the codebase.

Introduction of Docker

- Every service has its own docker compose file in its repository.
- Versions are imaged by git hash, so docker helps us ensure we’re testing exactly what’s going into production.
- Developers are intimately familiar with dependencies and deployment processes.

Current Status

- At this point, we’ve…
  - Created a few independent microservices.
  - Shunted some of our bulkier legacy functionality into hydras.
  - Killed a lot of old code.
  - Started using a docker based deployment pipeline to decrease the differences between environments.

Current Status

- From Jan 1, 2016 to now:
  - Our largest microservice has only a couple hundred lines of custom code.
  - Websites’ Uptime: 100%
  - Websites’ Average Server Response Time: 119ms

Next Steps

- All new and distinct functionality goes into its own microservice, deployed to its own cluster.
  - Only communicate with other services over API requests.
  - **NO BACKDOORS.**
- Use 3rd party resources whenever possible.
  - Don’t reinvent the wheel.

Next Steps

- Continue to decouple the hydras/monoliths.
  - Convert shared resources into Resource APIs.
  - **Kill legacy code.**
- Create microservices (or hopefully use 3rd party resources) to help orchestration between microservices.

Key Takeaways

- Don’t fall prey to analysis paralysis… take little steps.
- Use shunts to ease transitions (but put deadlines on their lifetime!).
- Delete code wherever possible.
- Treat internal services just like you would 3rd party services.

Key Takeaways

- Limit (hopefully to 1 service) the number of integration points to shared systems, especially databases.
- Version APIs to further decouple services and prevent changes in one service from affecting another.
- Keep micro services so small that it’s easier to rewrite than to refactor.

Questions and Contact

- Feel free to reach out…
  - tony@dose.com, @tonymaher5

goto; conference

Please

Remember to rate this session

Thank you!

follow us @gotochgo