Scaling the Unix Philosophy to Big Data

Surge 2013

Mark Cavage (@mcavage)
David Pacheco (@dapsays)
Joyent

What is the Unix Philosophy?

- 1986: Jon Bentley to Don Knuth: write a program that demonstrates Literate Programming
- Bentley asked Doug McIlroy to review it
- The challenge is still relevant today:

“Given a text file and an integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency.”

Knuth’s solution

- 10 pages of a custom algorithm in WEB, a Pascal derivative of his own invention

McIlroy’s solution

- One-liner:

```
tr -cs A-Za-z '\n' | tr A-Z a-z | \
   sort | uniq -c | sort -rn | sed ${1}q
```

The Unix philosophy

- Small programs that do one thing and do it well
- Facilitated by several conventions:
  - standardized input/output, stream processing, newline-separated records, often with fields separated by whitespace (or some other character) conventions
- **Not just the tools**, but an **approach** to building programs

Big Data: 1986 all over again?

- Google’s MapReduce paper sets up the same problem: “*Count of URL Access Frequency: The map function processes logs of web page requests and outputs (URL; 1). The reduce function adds together all values for the same URL and emits a {URL; total count} pair.*”
- 10 years later, this is still the canonical example in most M/R systems
- A natural fit for MapReduce, too:

Map

Reduce

```
tr -cs A-Za-z '\n' | tr A-Z a-z | \
    sort | uniq -c | sort -rn | sed ${1}q
```

Challenges bringing Unix to Big Data

- "Big Data" => need ability to store an arbitrary amount of data
- Arbitrary programs => compute abstraction must be the **OS itself**
- Parallel execution => still need orchestration abstractions (MR)
- Cloud deployment => must support multi-tenancy

Scaling Storage

- Everybody at Surge probably knows this, but you’ve got 3 choices: block/file/object
- Block: So very many wrongs, but at minimum it’s opaque, so out of the gate it’s a terrible abstraction
- File: NAS is what we really want, but H/A NAS is a lie. It’s trying to be both **C** and **A** in **CAP**
- Object: “similar to” a file abstraction, with liberating semantics...

Object Storage

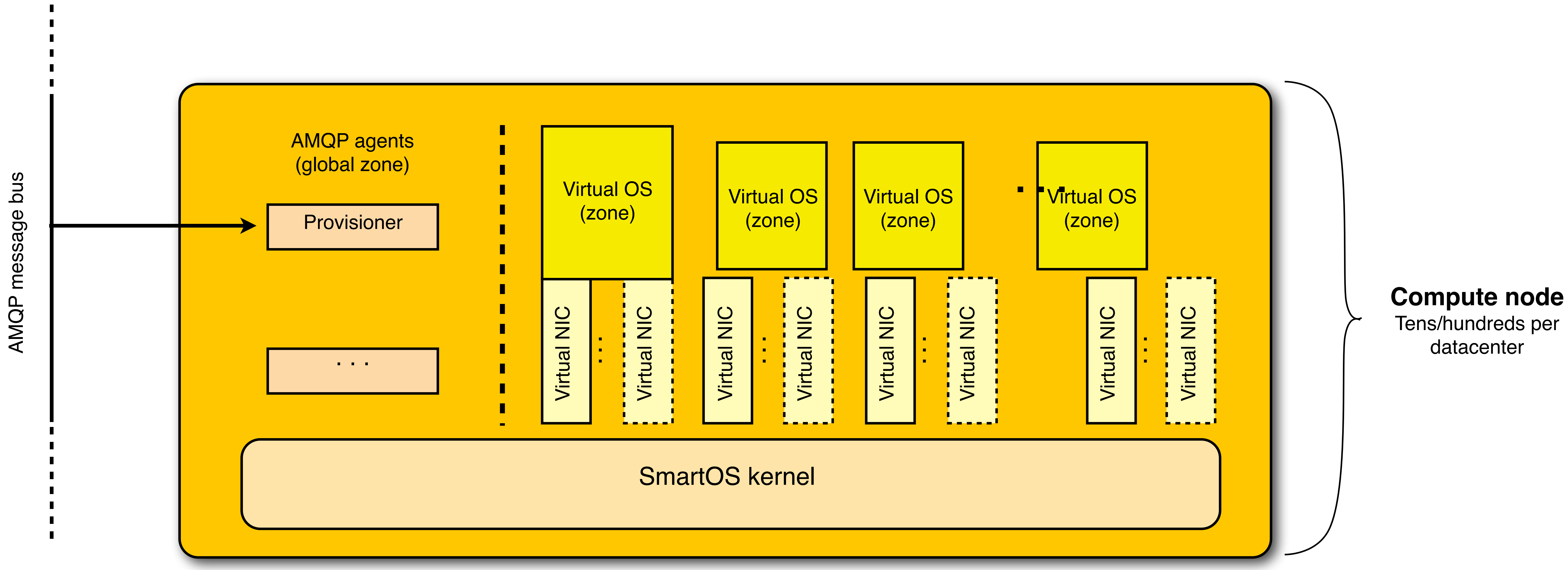
- Object stores (typically) look like a file system, but aren’t quite
- No partial updates
- No exposing volumes, or need to interface with existing clients
- Universal protocol (HTTP)
- The challenge is how to make UNIX work with an Object Store efficiently...

Virtualizing the OS

- One kernel on bare metal, many virtual OS containers (“zones”), each with its own root filesystem
- Much more efficient than hardware-based virtualization
- “root” in the zone does not compromise the rest of the system
- Rich interface between “global zone” and individual tenants’ zones

Virtualizing the OS

Joyent



Compute node
Tens/hundreds per
datacenter

Hyperlofs

- What if we had an object store, that left files as objects?
- Could we **bring back** the semantics of the FS when running compute?
- Hyperlofs!

Hyperlofs

```
 /*
* Hyperlofs is a hybrid file system combining features of the tmpfs(7FS) and
* lofs(7FS) file systems.  It is modeled on code from both of these file
* systems.
*
* The purpose is to create a high performance name space for files on which
* applications will compute.  Given a large number of data files with various
* owners, we want to construct a view onto those files such that only a subset
* is visible to the applications and such that the view can be changed very
* quickly as compute progresses.  Entries in the name space are not mounts and
* thus do not appear in the mnttab.  Entries in the name space are allowed to
* refer to files on different backing file systems.  Intermediate directories
* in the name space exist only in-memory, ala tmpfs.  There are no leaf nodes
* in the name space except for entries that refer to backing files ala lofs.
*
* The name space is managed via ioctls issued on the mounted file system and
* is mostly read-only for the compute applications.  That is, applications
* cannot create new files in the name space. If a file is unlinked by an
* application, that only removes the file from the name space, the backing
* file remains in place.  It is possible for applications to write-through to
* the backing files if the file system is mounted read-write.
*/
```

Putting it altogether: Manta

- Scalable, durable HTTP Object Store
- Namespace looks like a POSIX filesystem
- *In situ* compute as a first-class operation

Manta: design parameters

- **CAP: Choose strong consistency**
  - CAP is not a monolithic choice:

    Can build A on top of C, but choosing A prohibits C
- Must be highly-available (multi-AZ, tolerates transient failures)
- Objects must be stored as simple files (so we can run programs on them)
- Compute API should “feel like Unix”

Manta architecture

- Frontend: Node.js REST servers
- Metadata: Postgres (sharded, replicated for availability)
- Storage: ZFS
- Compute
- Asynchronous services (metering, garbage collection, monitoring)
- Group membership: DNS on ZooKeeper

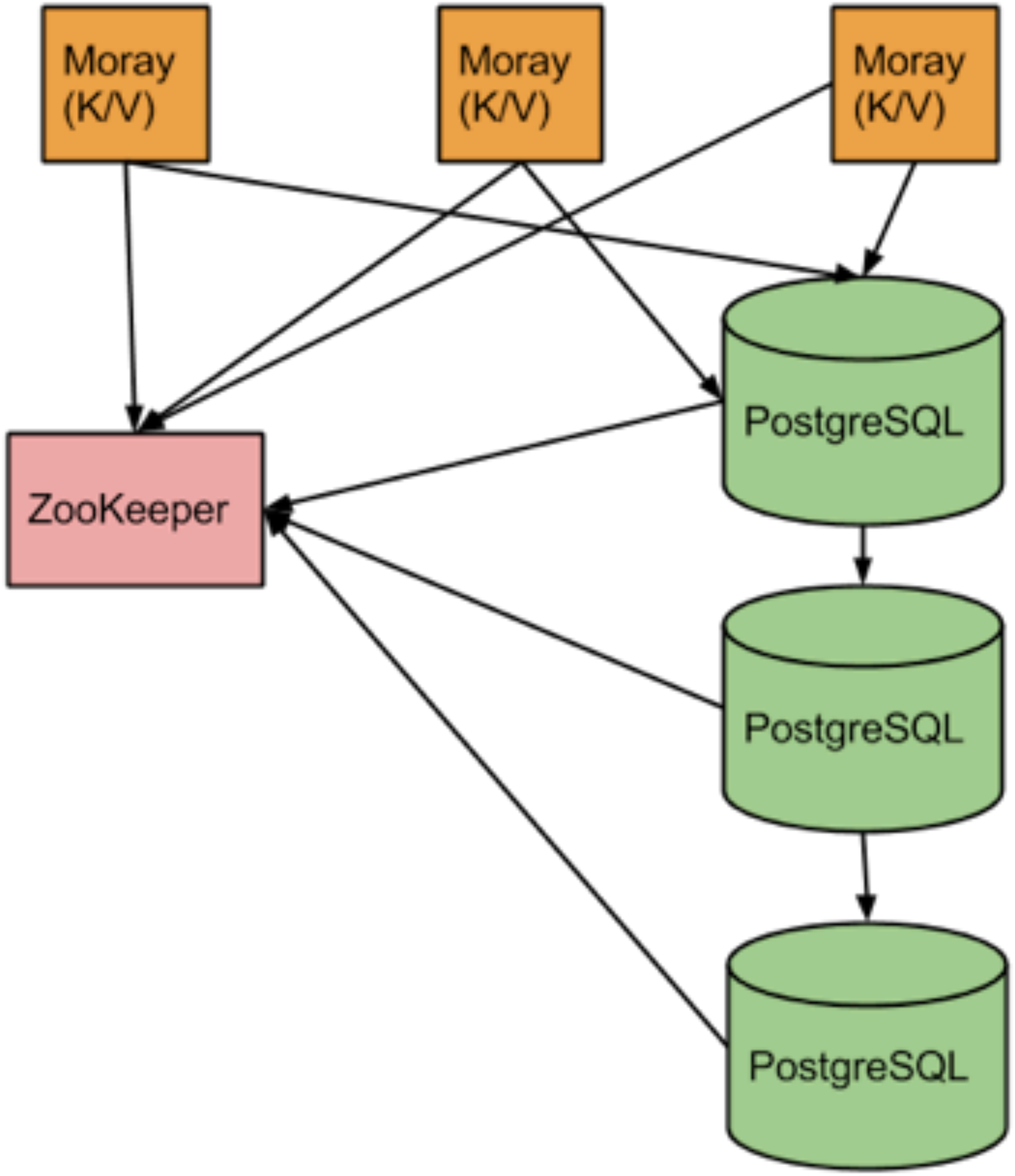
Frontend

- Stud: SSL Terminator
- HAProxy: HTTP Terminator/LB
- WebAPI: restify on Node.js
- Redis: authentication cache

Metadata tier

- Metadata copies on 2+ Postgres DBs
- Consistent hash on dirname ("/mark/stor/foo")
- Replication Topology Managed with Zookeeper
- Moray: custom Node.js key/value interface on top

Metadata tier



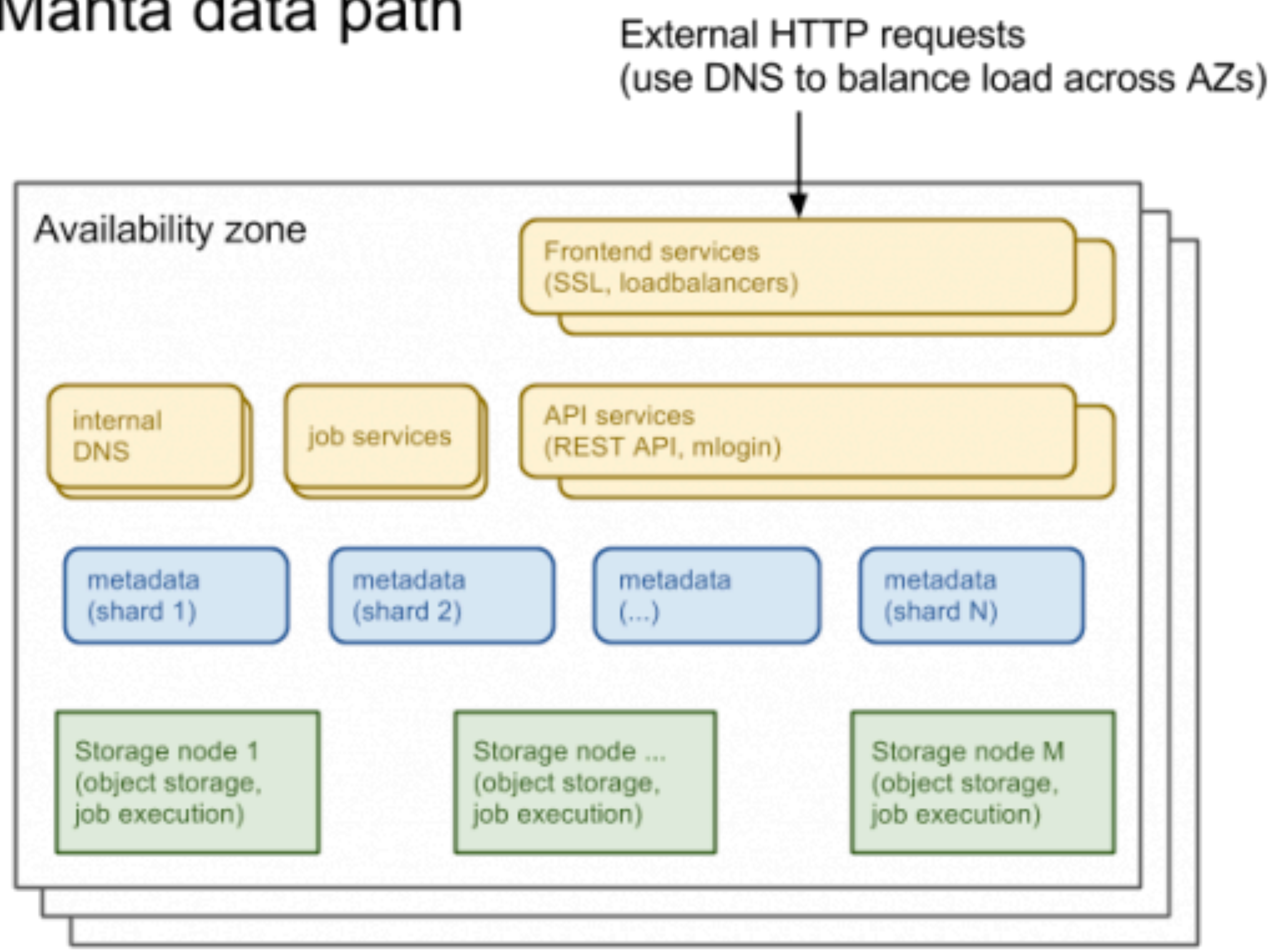
Storage: bare metal

Storage: bare metal

- 73 TiB (soon to be 100 TiB) in 4U, 256GB DRAM, RAIDZ2
- SmartOS (ZFS, Zones)
- Storage interface: Nginx
- Needs to support compute jobs, too (more later)

Storage architecture: each AZ

Manta data path



External HTTP requests
(use DNS to balance load across AZs)

Availability zone

Frontend services
(SSL, loadbalancers)

internal
DNS

job services

API services
(REST API, mlogin)

metadata
(shard 1)

metadata
(shard 2)

metadata
(...)

metadata
(shard N)

Storage node 1
(object storage,
job execution)

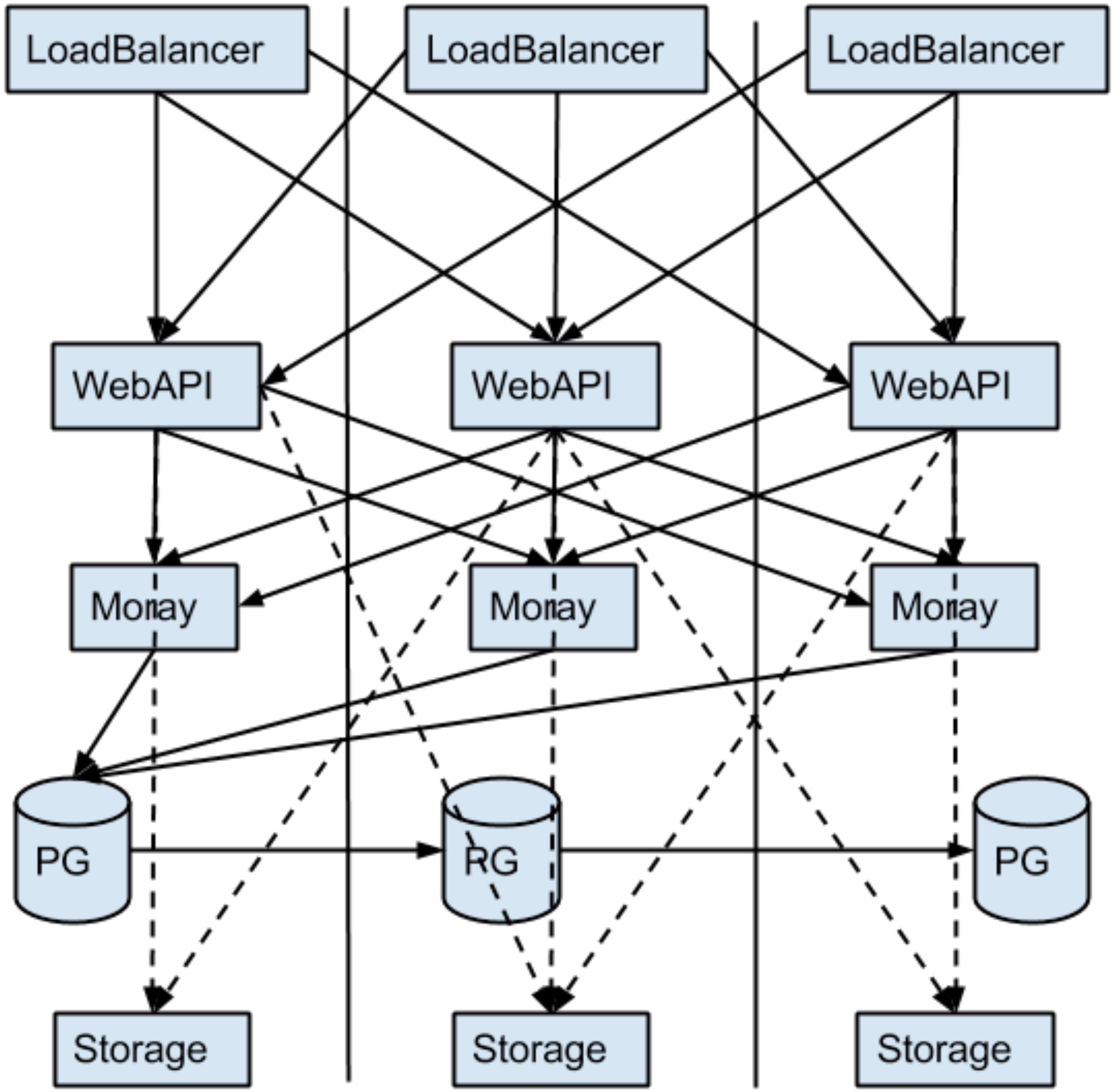
Storage node ...
(object storage,
job execution)

Storage node M
(object storage,
job execution)

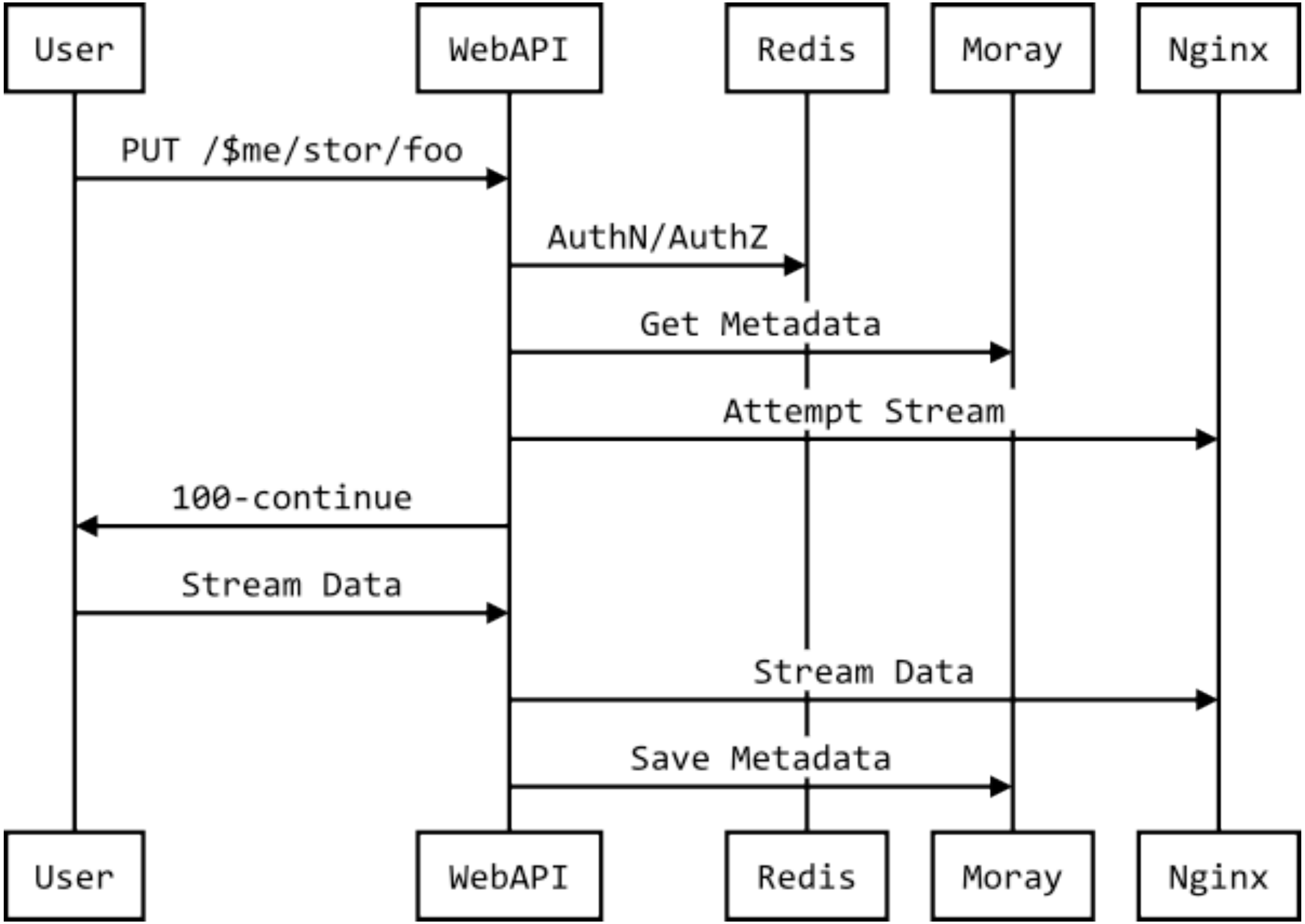
🟨 **Stateless services**
🟦 **Metadata tier**
🟩 **Storage tier**

- Stateless services use internal DNS to balance load across all AZs
- Metadata shards have at least one instance in each AZ
- Copies of objects are stored in separate AZs (up to three)

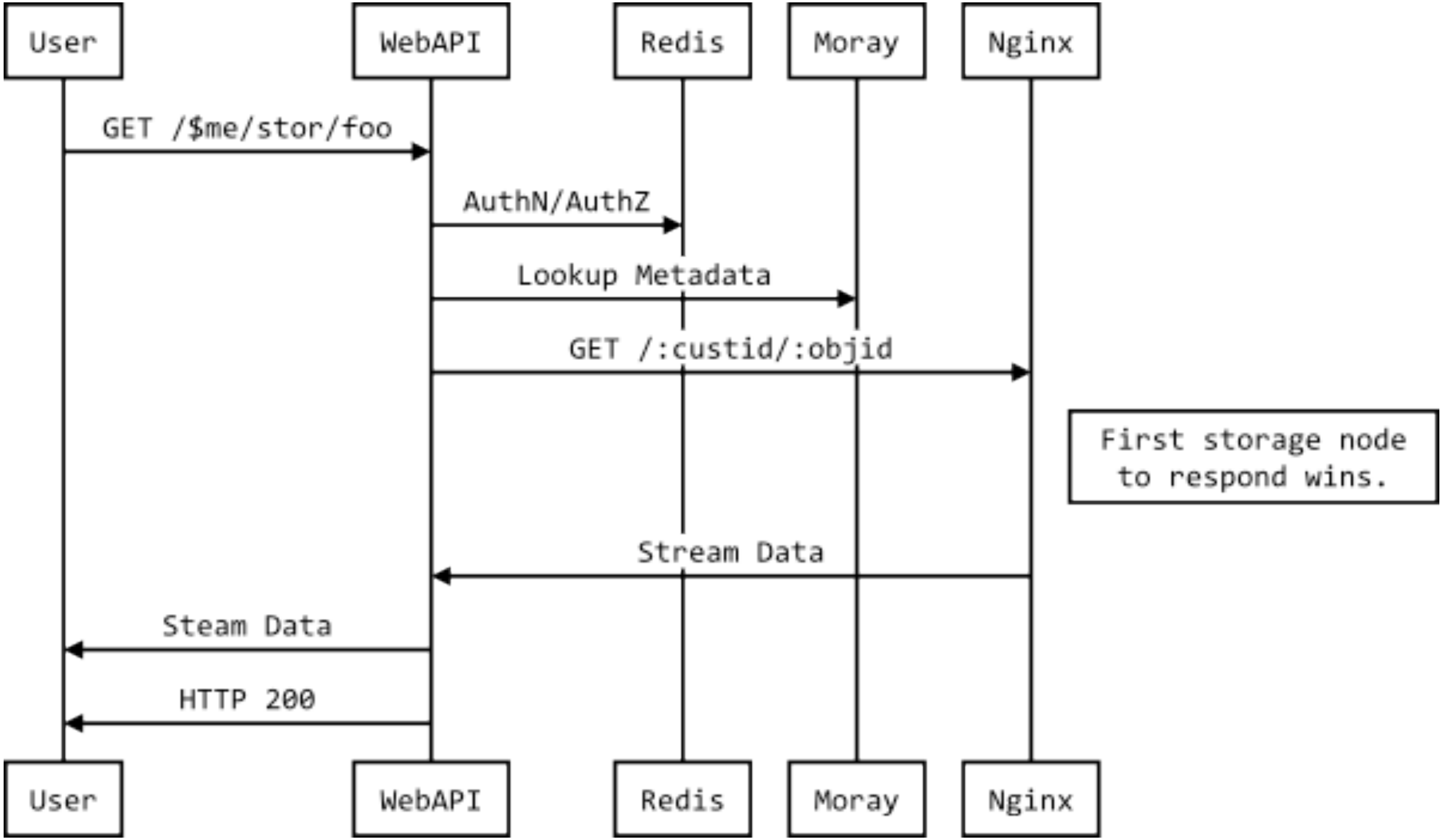
Storage architecture (X-DC deployment)



Storage: PUT request



Storage: GET request



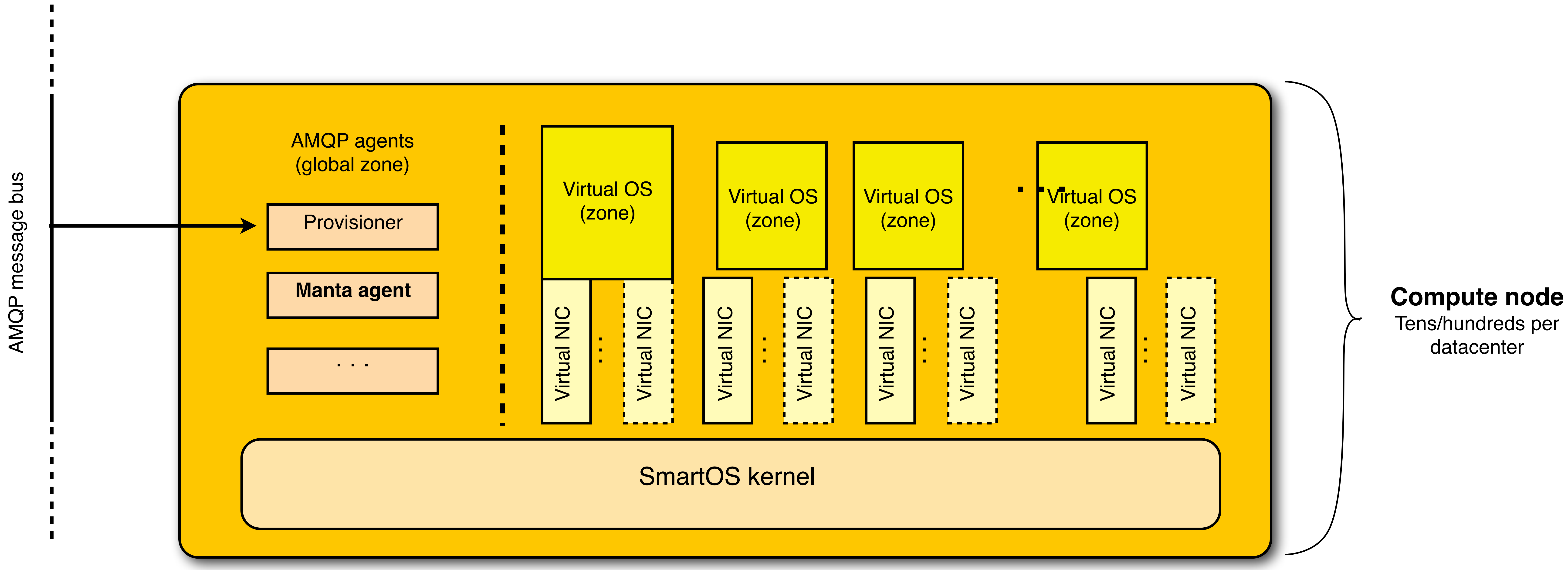
Group Membership: DNS

- Custom (Node.js) DNS server
- Participants write an “ephemeral node” in ZK on startup
- This “mostly” works...NSCD sucks and ZK sucks
- But modulo *removing* capacity, it’s pretty nice

Compute: overview

- Users submit **jobs,** which specify pipelines to run either on each input separately (**map**) or all inputs together (**reduce**).
- Inputs: objects, accessed as regular files
- Outputs: saved as objects
- Orchestration: fleet of *jobsupervisors* (stateless)
- State: stored in one shard of the metadata tier (postgres) (war stories coming up)

Virtualizing the OS



AMQP message bus

AMQP agents
(global zone)

Provisioner

Manta agent

. . .

Virtual OS
(zone)

Virtual OS
(zone)

Virtual OS
(zone)

. . .

Virtual OS
(zone)

Virtual NIC

Virtual NIC

Virtual NIC

Virtual NIC

Virtual NIC

Virtual NIC

Virtual NIC

Virtual NIC

SmartOS kernel

Compute node
Tens/hundreds per
datacenter

Compute: execution

- User programs run inside transient zones **managed by the service**.
- Resource usage: capped but allows bursts
- Input: objects mapped in as RO files (for “map”) and redirected as stdin.
- When done: “zfs rollback” and reboot the zone
- (All of this is behind-the-scenes)
- Demo

Bentley’s challenge, scaled up

- Arbitrarily scalable variant of McIlroy’s solution to Bentley’s challenge:

```
mfind /manta/public/examples/shakespeare | \
     mjob create -o -m "tr -cs A-Za-z '\n' | \
     tr A-Z a-z | sort | uniq -c" -r \
     "awk '{ x[\$2] += \$1 }
     END { for (w in x) {
         print x[w] \" \" w } }' | \
     sort -rn | sed ${1}q"
```

Everything else

- Metering (for billing): compute job run over log files (JSON + bunyan)
- Monitoring: compute job run over log files (JSON + bunyan)
- Garbage collection: compute job run over database dumps of the metadata tier (JSON), plus manifests reported by storage nodes
- ... (consistency audit, storage rebalancing, etc.)

Debugging Node.js

- Heavy use of MDB and DTrace
- bunyan (and live bunyan -p)
- Example: Frontend memory consumption
  - Node v0.10.X “just fixed it” (via rewriting Streams API)

Zookeeper

- Don’t reboot “the leader”
- Don’t do too many reads...or writes...
- Don’t give it too little DRAM
- “No, don’t touch it, don’t even look at it!”

Nginx

- POSIX fsync() trivia

Postgres (job state)

- Lots of churn, 24/7 duty cycle (bad idea?)
- Vacuuming
- Analyzing
- Table fragmentation

Postgres (replication)

- Synchronous replication: master claims to be up-to-date, slave has no idea about replication, no data flowing (!!!)

Key takeaways

- Unix loves Big Data
- Eventual consistency is not the only option
- When the *storage system of record* is globally available and supports arbitrary compute, many use cases become unified:
  - CDN source (e.g., web assets)
  - Log storage, processing, and analysis
  - Image processing and video transcoding
  - Indexing and data warehousing

The most important Big Data problem

References

- “Programming Pearls: a literate program”: http://dl.acm.org/citation.cfm?id=315654
- “MapReduce:
  Simplified Data Processing on Large Clusters”
  http://research.google.com/archive/mapreduce.html
- Manta CAP tradeoffs: http://dtrace.org/blogs/dap/2013/07/03/fault-tolerance-in-manta/
- Manta Docs: http://apidocs.joyent.com/manta/

Scaling the Unix Philosophy to Big Data

Surge 2013
Mark Cavage (@mcavage)
David Pacheco (@dapsays)
Joyent