AWS re:Invent

November 27- 29, 2012 | The Venetian | Las Vegas

Building Web-Scale Applications with AWS

Simon Elisha / Principal Solution Architect / @simon_elisha

James Hamilton / Vice President & Distinguished Engineer

#reinvent

8

Months of Travel

7

Minutes of Terror

100,000

Concurrent Viewers

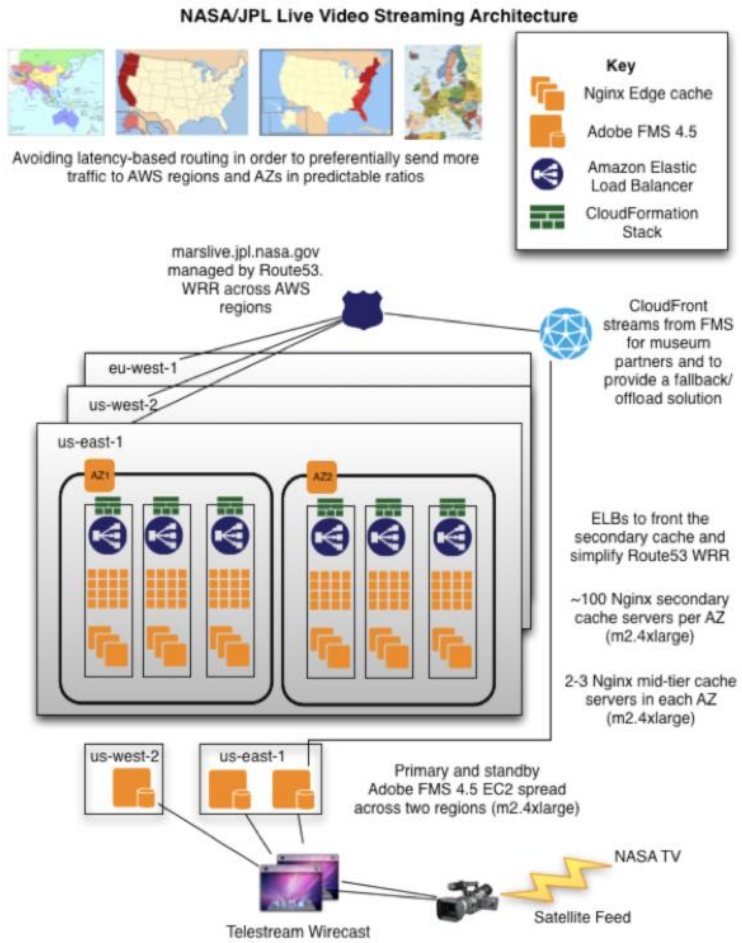
0

Second Chances

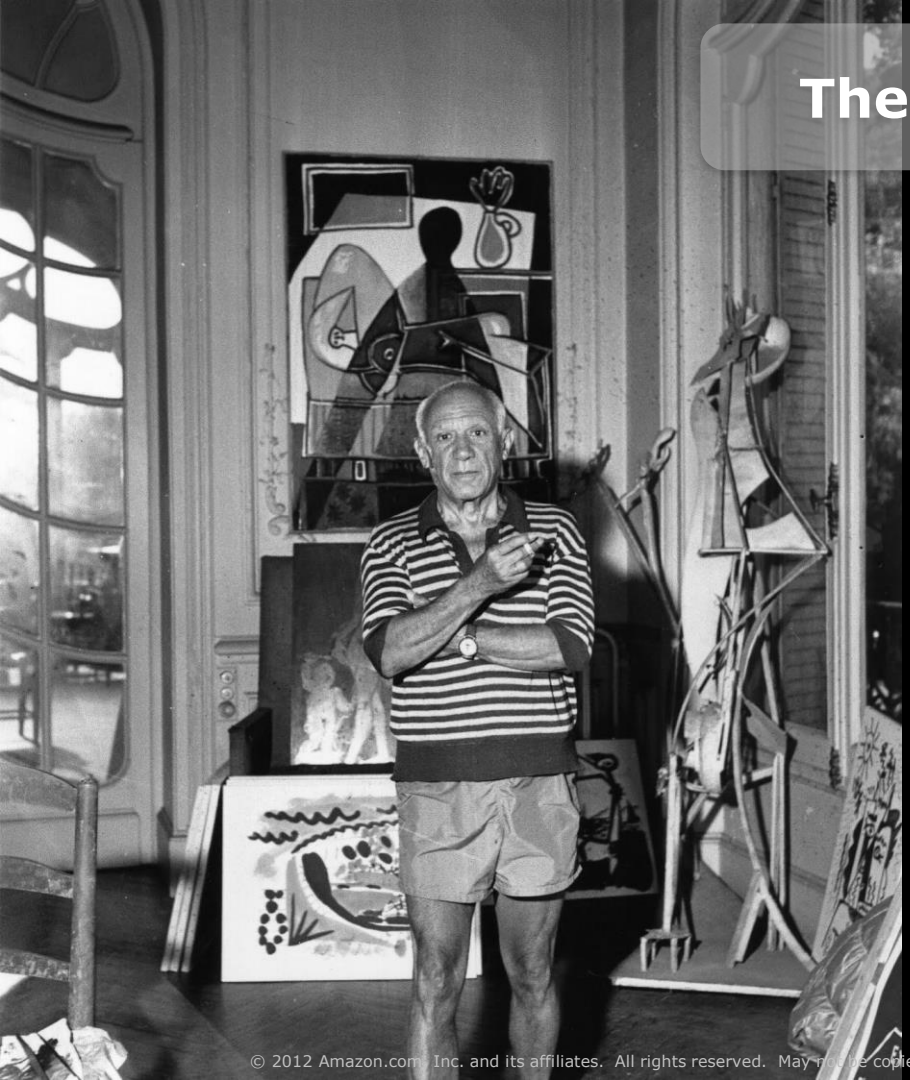
A Tale of Web Scale

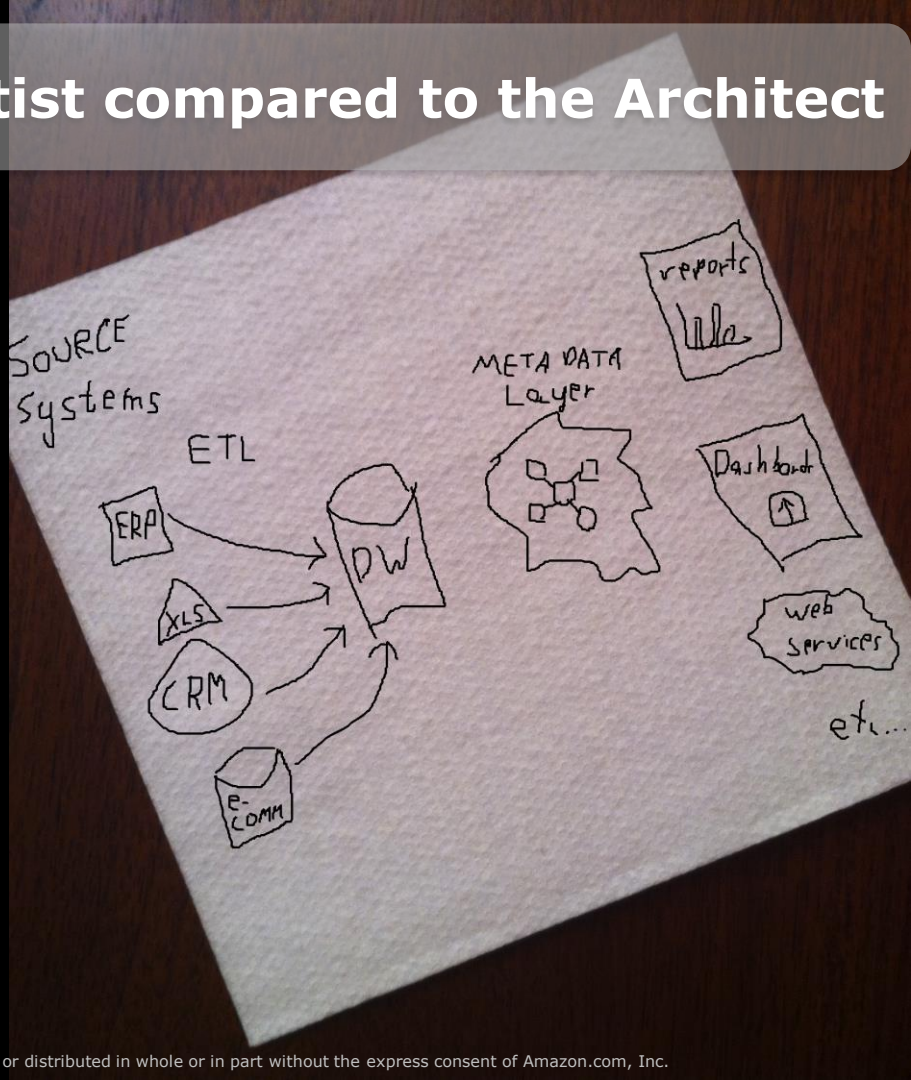
Streaming images and video of Curiosity's landing on Mars

- *Multiple Regions – fully automated*
- *Each stack handles up to 25 Gbps of traffic*
- *Dynamically handles spikes in load*
- *Spreads workload across geographies*
- *Over 100 instances per stack*
- *All gone away now…*



The Artist compared to the Architect





Primary Colors

Fundamental Design Considerations for Web-Scale Applications on AWS

1. Design for Failure

and nothing will really fail

"Everything fails, all the time"
Werner Vogels, CTO Amazon.com

- Avoid single points of failure
- Assume everything fails, and design backwards
- Goal: Applications should continue to function even if the underlying physical hardware fails or is removed or replaced

2. Loose coupling sets you free

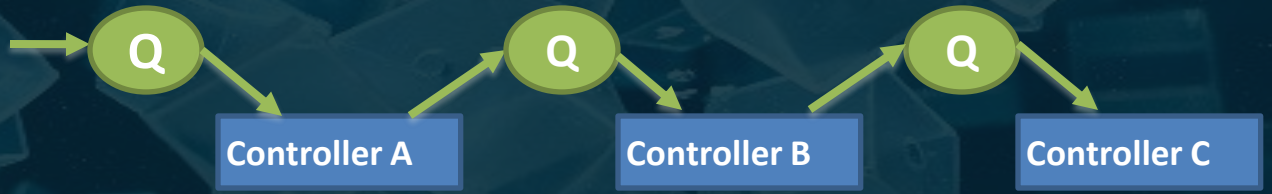
The looser they're coupled, the bigger they scale

- Independent components
- Design everything as a black box
- De-couple interactions
- Load-balance clusters

Use Amazon SQS as Buffers

Tight Coupling

Loose Coupling using Queues



3. Implement Elasticity

Elasticity is a fundamental property of the Cloud

- Don’t assume health or fixed location of components
- Use designs that are resilient to reboot and re-launch
- **Bootstrap** your instances: Instances on boot will ask a question “*Who am I & what is my role?*”
- Enable dynamic configuration

- Use Auto-scaling (Free)
- Use [Elastic] Load Balancing on multiple layers
- Use configurations in SimpleDB/S3/etc to bootstrap instance

4. Build Security in every layer

Design with Security in mind

With the cloud, you lose a little bit of physical control, but not your ownership

- Create distinct Security Groups for each Amazon EC2 tier
- Use security group-based rules to control access between layers
- Use Virtual Private Cloud (VPC) to combine internal and AWS assets
- Encrypt data "at-rest" in Amazon S3
- Encrypt data "in-transit" (SSL)
- Consider encrypted file systems in EC2 for sensitive data
- Use AWS Identity & Access Management (IAM)
- Use MultiFactor Authentication (MFA)

5. Don't fear constraints

Re-think architectural constraints

More RAM? Distribute load across machines
Shared distributed cache

Better IOPS on my database?
Multiple read-only / sharding / DB clustering / Caching / Provisioned IOPs / SSD instances

Hardware Config does not match?
Implement Elasticity

Your hardware failed or messed up config?
Simply throw it away and switch to new hardware with no additional cost

Performance
Caching at different levels (Page, Render, DB)

6. Think Parallel

Serial and Sequential are now history

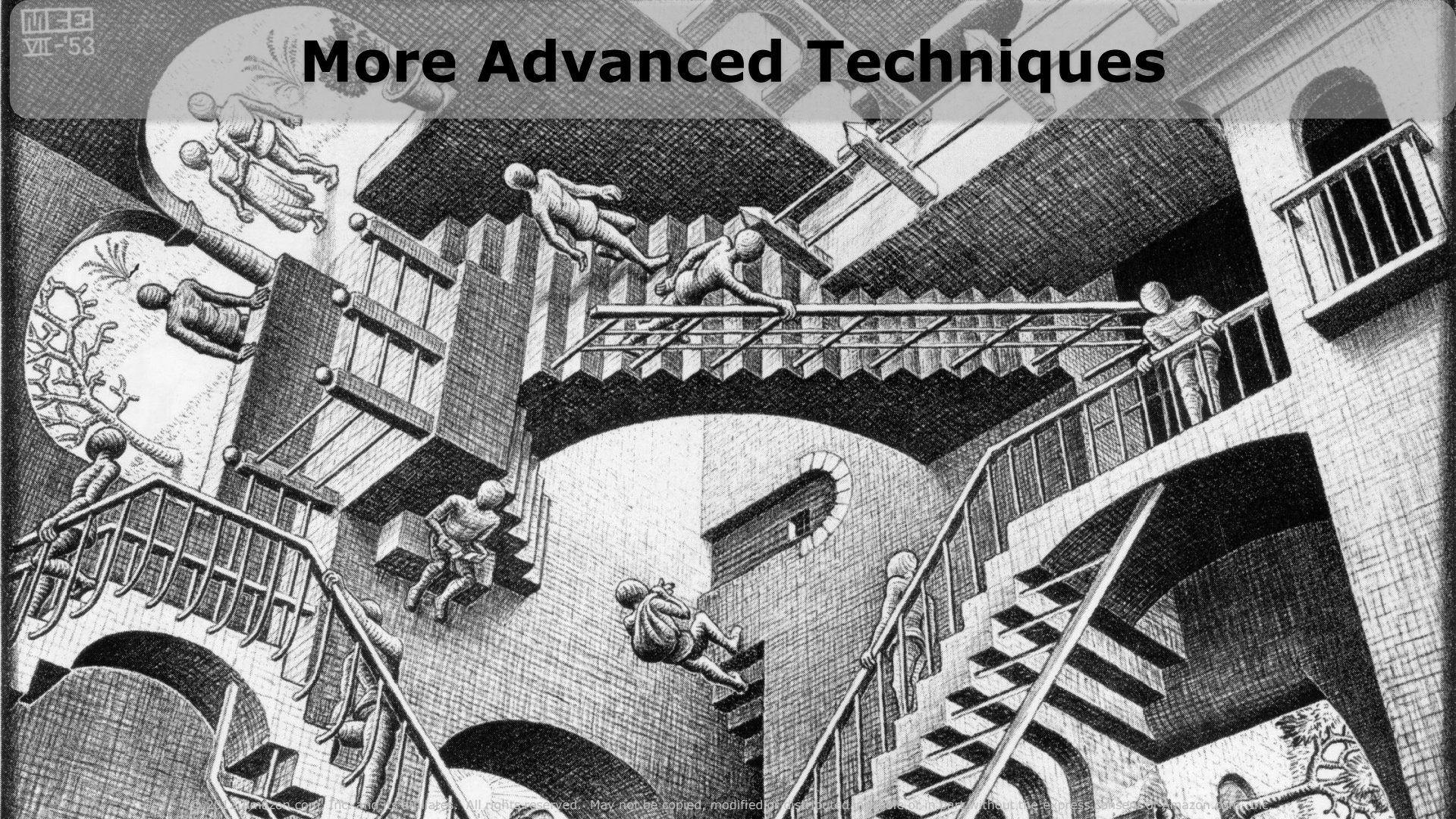
- Multi-threading and concurrent requests to cloud services
- Run parallel MapReduce Jobs
- Use Elastic Load Balancing to distribute load across multiple servers
- Decompose a Job into its simplest form

7. Leverage multiple storage options

One size DOES NOT fit all

Amazon Simple Storage Service (Amazon S3): large static objects
Amazon Glacier: long term archival of objects
Amazon CloudFront: content distribution
Amazon DynamoDB: infinitely scalable NoSQL "Big Tables"
Amazon ElastiCache: in memory caching
Amazon CloudSearch: fast, highly-scalable search functionality
Amazon Elastic Compute Cloud (Amazon EC2) local disk drive : transient data
Amazon Elastic Block Store (Amazon EBS): persistent storage + snapshots on S3
Amazon EBS PIOPs: consistent, persistent storage for any RDBMS + Snapshots on S3
Amazon Relation Database Service (Amazon RDS): RDBMS service –
Automated and Managed MySQL, Oracle & SQL Server
Amazon EC2 High I/O Instances: high performance, local SSD-backed storage

More Advanced Techniques

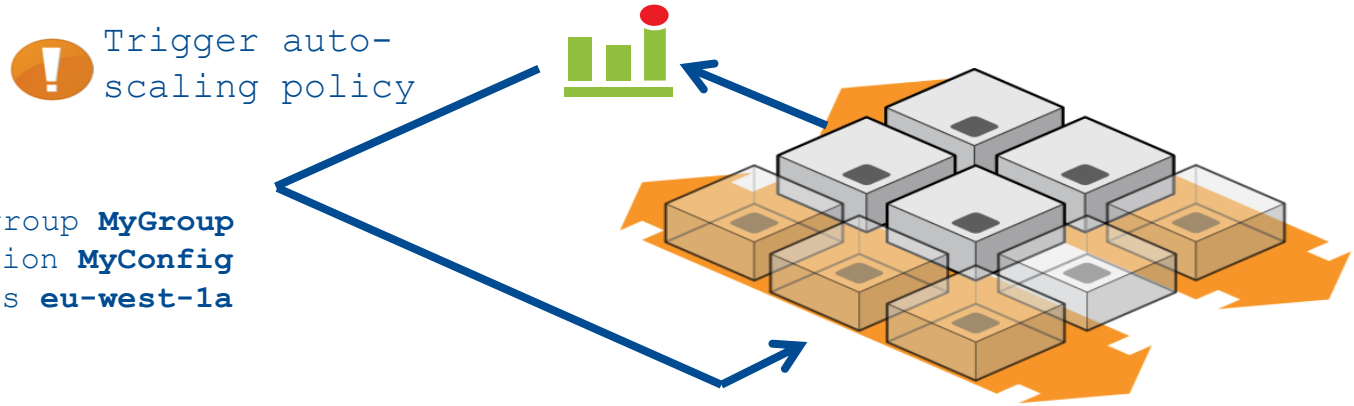


Stateless Software Architecture

Does not retain information about the last session into the next – e.g. user data, parameters, logic outcomes.

You know – like that thing called the HTTP Protocol…

Lets you Auto Scale




```
as-create-auto-scaling-group MyGroup
      --launch-configuration MyConfig
      --availability-zones eu-west-1a
      --min-size 4
      --max-size 200
```

- Deployment & Administration
- App Services
- **Compute** | Storage | Database
- Networking
- AWS Global Infrastructure

Auto Scaling

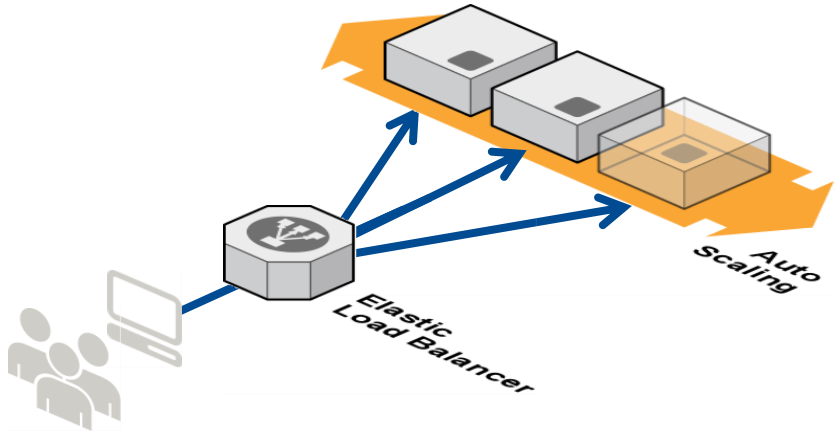
Automatic re-sizing of compute clusters based on demand

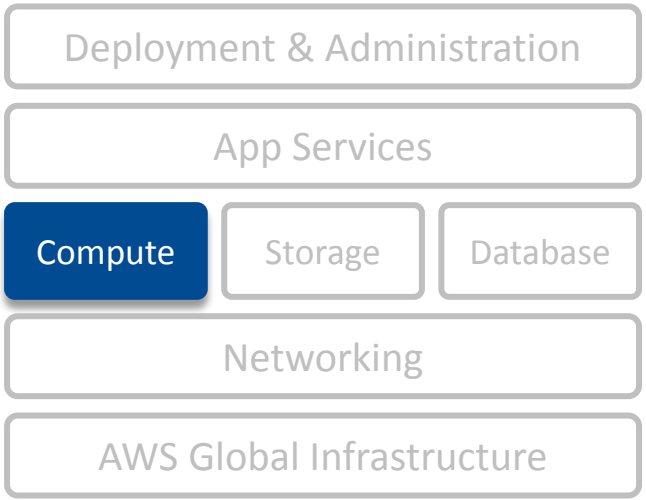
Feature	Details
Control	Define minimum and maximum instance pool sizes and when scaling and cool down occurs.
Integrated to Amazon CloudWatch	Use metrics gathered by CloudWatch to drive scaling.
Instance types	Run Auto Scaling for On-Demand and Spot Instances. Compatible with VPC.

…and Spread the Load

Elastic Load Balancing

- *Create highly scalable applications*
- *Distribute load across EC2 instances in multiple availability zones*





Feature	Details
Available	Load balance across instances in multiple Availability Zones
Health checks	Automatically checks health of instances and takes them in or out of service
Session stickiness	Route requests to the same instance
Secure sockets layer	Supports SSL offload from web and application servers with flexible cipher support
Monitoring	Publishes metrics to CloudWatch

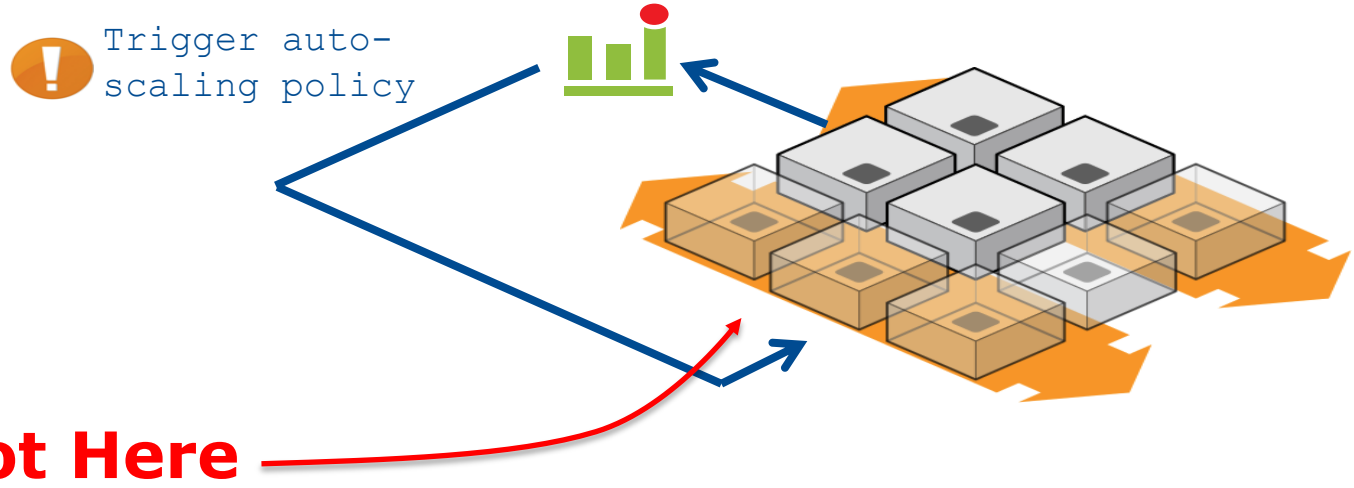
But usually some state has to reside somewhere

- Cookies in browser
- Session database
- Memory-resident session manager
- Framework provided session handler

So this store of state needs to be…

- Performant
- Scalable
- Reliable

Where should session state reside?



Here →

Session State Service

State must reside **OUTSIDE** the scope of the elements you wish to scale

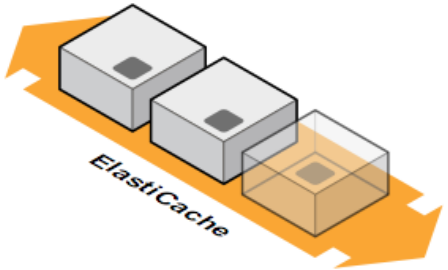
And what do I build it on?

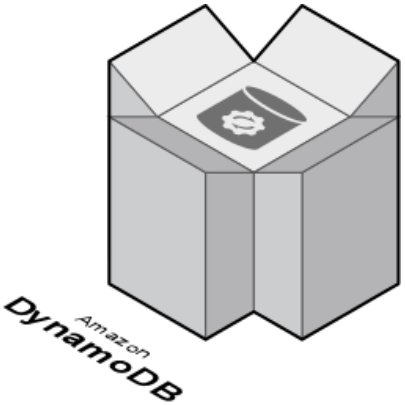
Performant

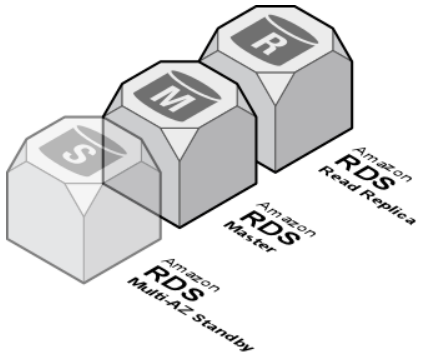
Scalable

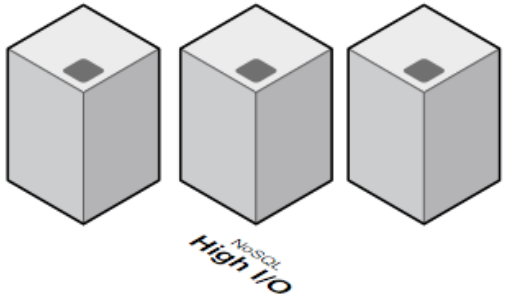
Reliable

The state service **itself** must be well architected









AWS Reference Architectures

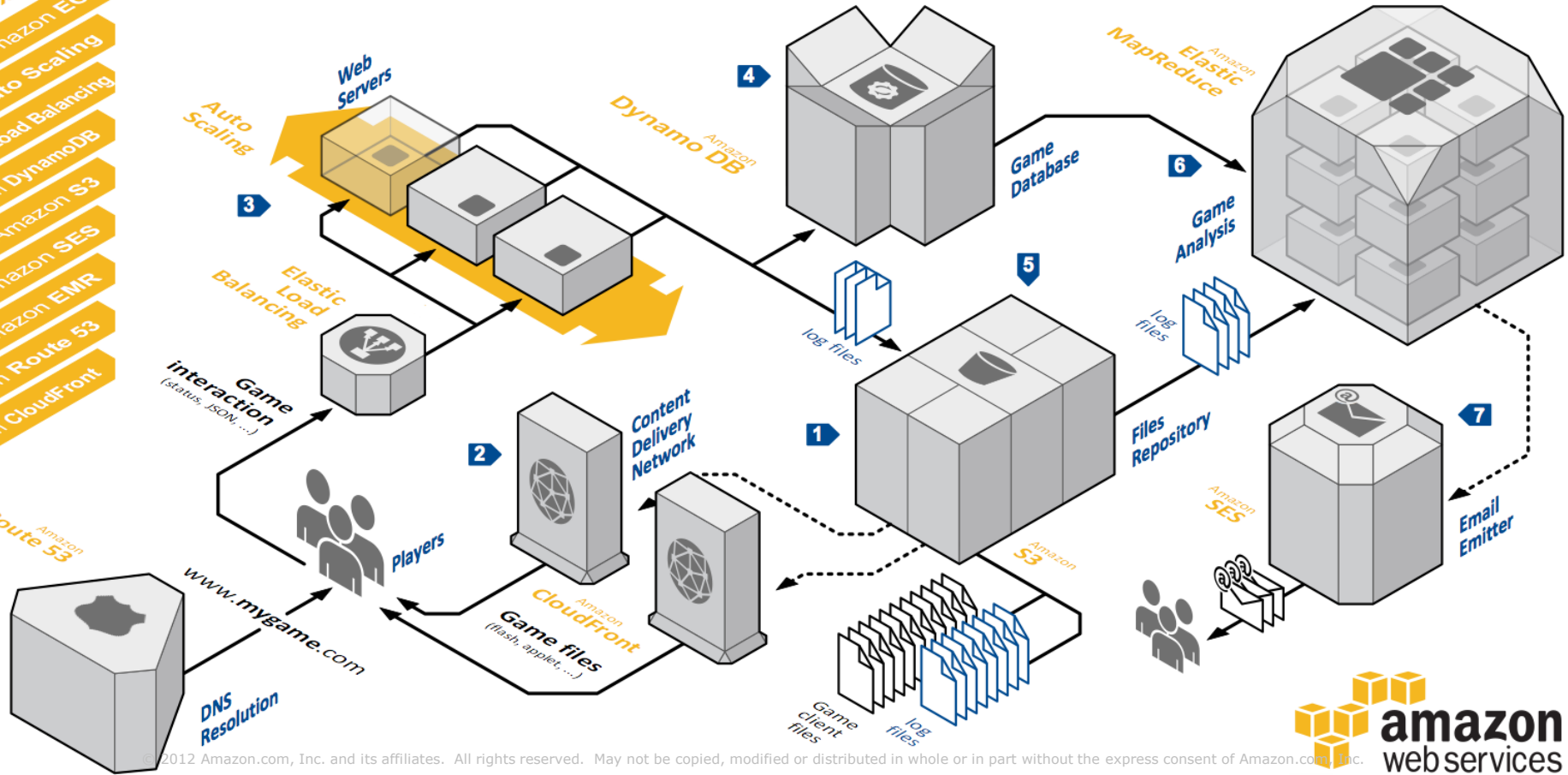
- Amazon EC2
- Auto Scaling
- Elastic Load Balancing
- Amazon DynamoDB
- Amazon S3
- Amazon SES
- Amazon EMR
- Amazon Route 53
- Amazon CloudFront

ONLINE GAMES

Online games back-end infrastructures can be challenging to maintain and operate. Peak usage periods, multiple players, and high volumes of write operations are some of the most common problems that operations teams face.

But the most difficult challenge is ensuring flexibility in the scale of that system. A popular game might suddenly receive millions of users in a matter of hours, yet it must continue to provide a satisfactory player experience. Amazon Web Services provides different tools and services that can be used for building online games that scale under high usage traffic patterns.

This document presents a cost-effective online game architecture featuring automatic capacity adjustment, a highly available and high-speed database, and a data processing cluster for player behavior analysis.



amazon web services

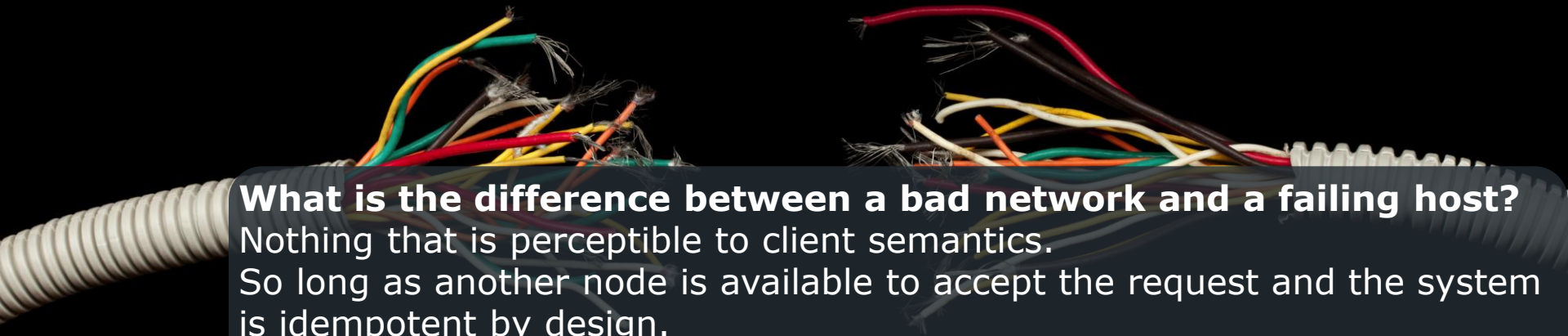
“If at first you don’t succeed…”

Retry logic is **fundamental** to scalable design

But without back-off logic, your service may still “break”

Use Idempotency as “secret sauce”

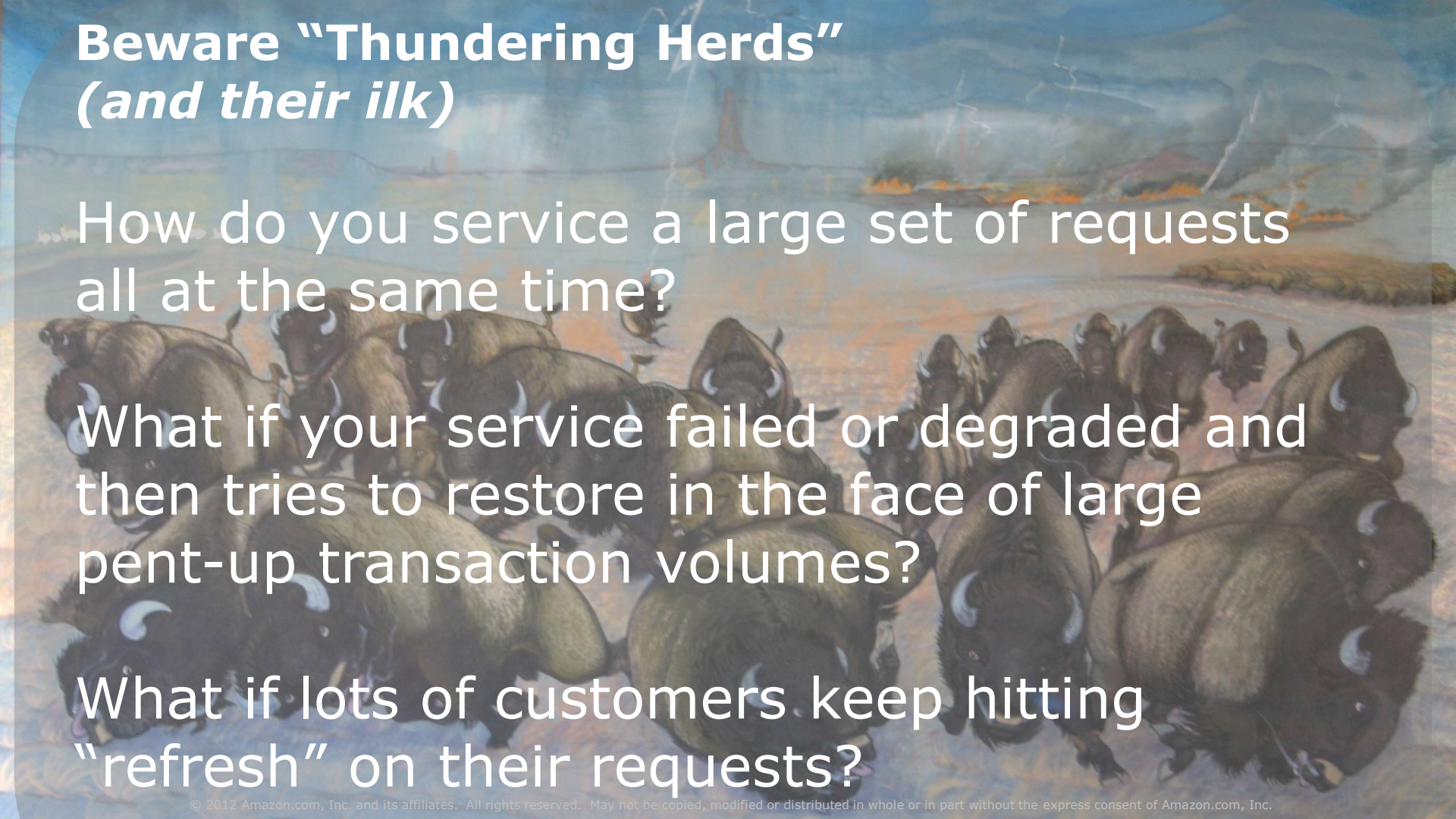
Without retries, the Web would not work.
Assumes things will fail, and we will need to retry.



What is the difference between a bad network and a failing host?
Nothing that is perceptible to client semantics.
So long as another node is available to accept the request and the system is idempotent by design.

Retry is now familiar behavior for users.
E.g. refresh your web page, refresh Twitter, etc.

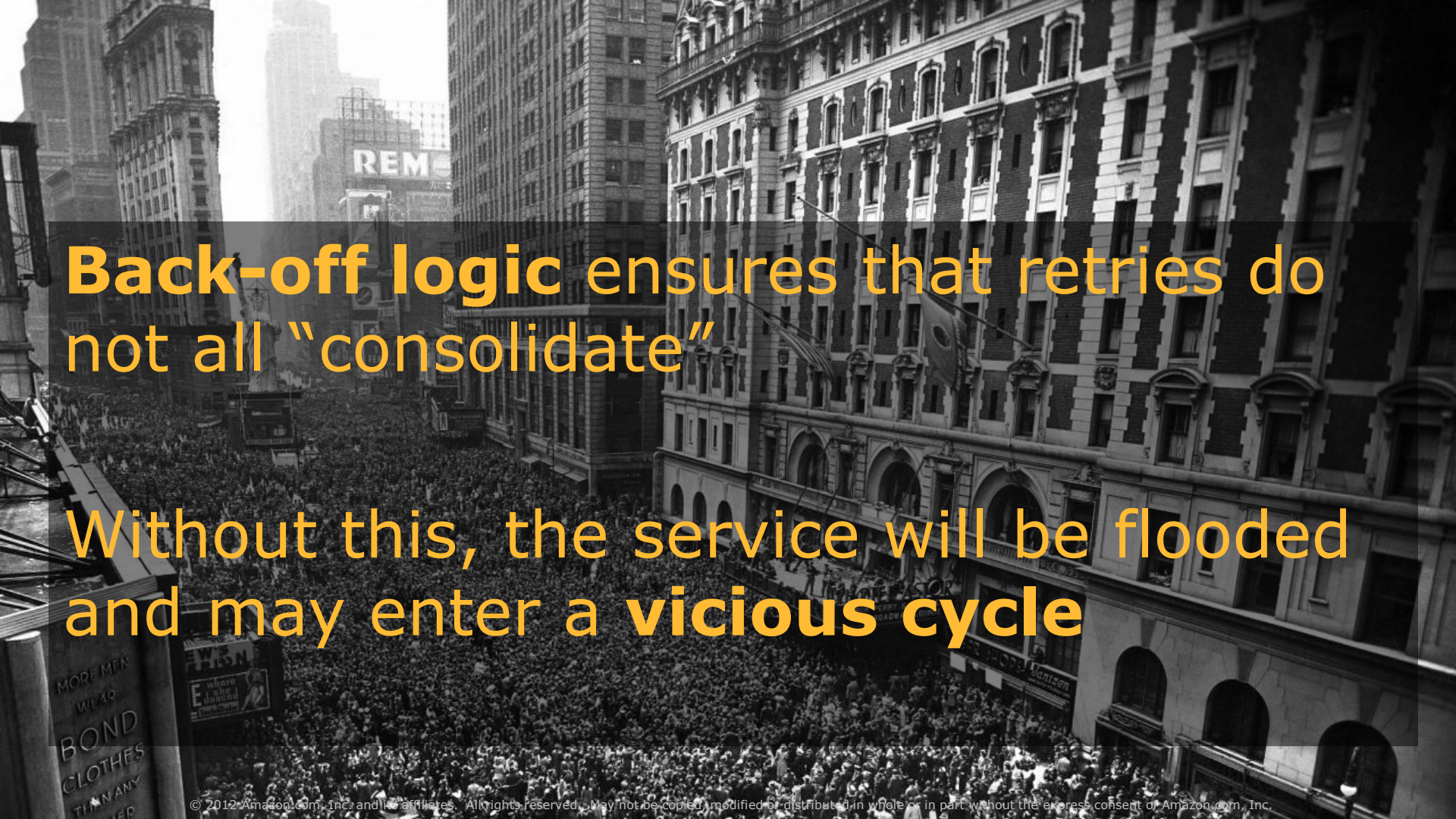
Beware "Thundering Herds" *(and their ilk)*



How do you service a large set of requests all at the same time?

What if your service failed or degraded and then tries to restore in the face of large pent-up transaction volumes?

What if lots of customers keep hitting "refresh" on their requests?



Back-off logic ensures that retries do not all "consolidate"

Without this, the service will be flooded and may enter a **vicious cycle**

Exponential Backoff Algorithm

```
currentRetry = 0
 DO
   status = execute Amazon SimpleDB request
   IF status = success OR status = client error (4xx)
     set retry to false
     process the response or client error as appropriate
   ELSE
     set retry to true
     currentRetry = currentRetry + 1
     wait for a random delay between 0 and (4^currentRetry * 100) milliseconds
   END-IF
 WHILE (retry = true AND currentRetry < MaxNumberOfRetries)
```

Idempotence

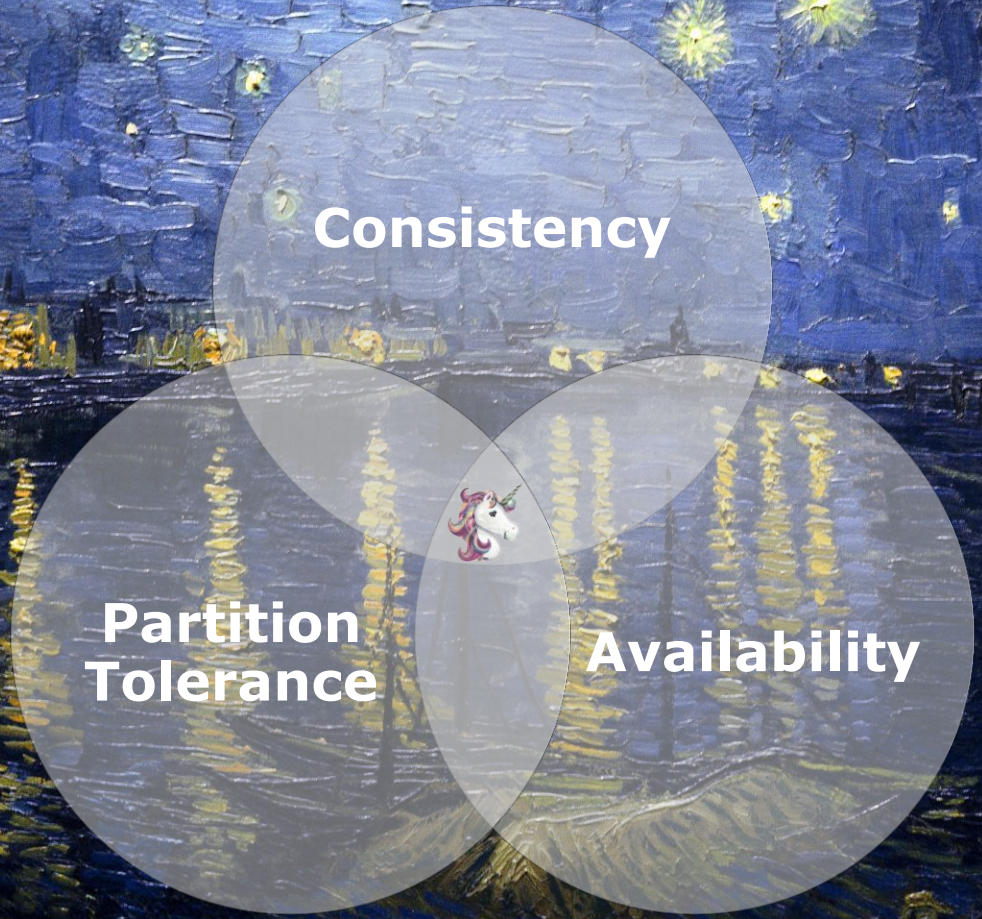
The property of an operation whereby it can be applied **multiple** times without changing the result beyond the initial application.

It keeps you “safe” because executing the same thing twice (or more) has no more affect than doing it once.

EC2 Instance Creation Example

PROMPT> **ec2-run-instances ami-b232d0db -k gsg-keypair --client-token 550e8400-e29b-41d4-a716-446655440000**

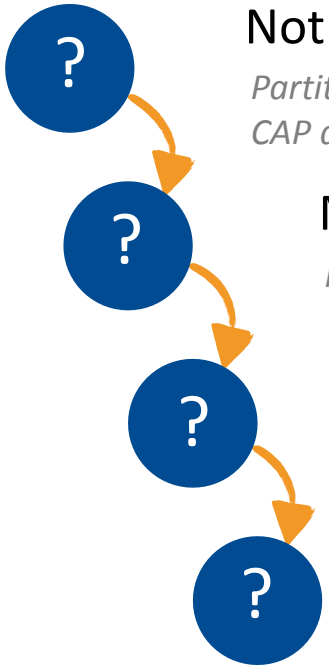
CAP Theorem



CAP Examples…

- CA — Synchronously replicated databases in normal operation
- CP — Read-only storage systems, synchronously replicated databases when partitioned, membase
- AP — Multi-master / asynchronously replicated databases (Active Directory, Outlook and Exchange, DNS)

CAP Theory 12 Years on…



Not as simple as “choose 2 out of 3”

Partitions are rare – so why sacrifice C or A for a rare/managed event?
CAP attributes are continuous rather than binary measures – particularly in complex systems.

Managing partitions enables the “easy” choice of C & A

Mitigate effect of P on C & A for the cases that P can occur.

A function/operation-level decision, not system level.

e.g. an ATM can still accept deposits during Partition.

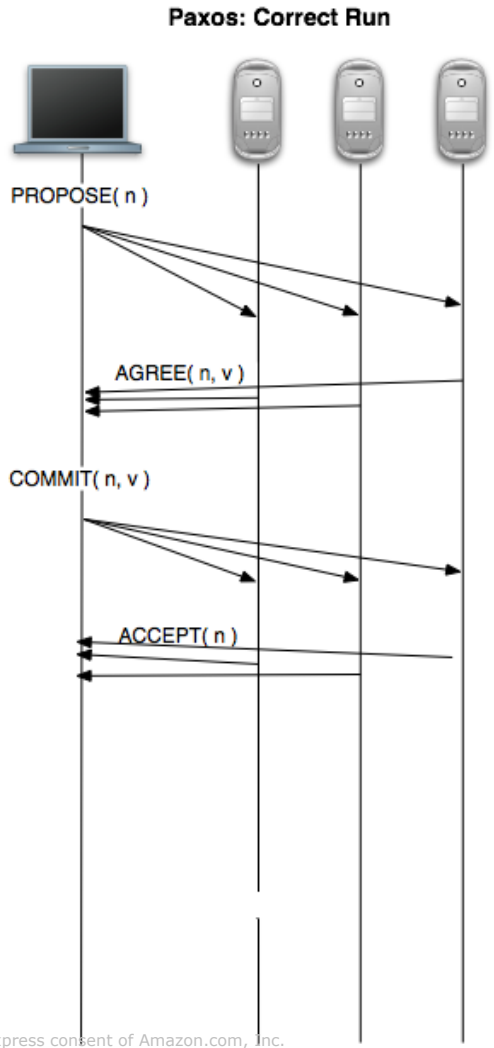
Latency and the Partition decision are closely related

Designers can set time bounds intentionally according to target response times; systems with tighter bounds will likely enter partition mode more often and at times when the network is merely slow and not actually partitioned.

**http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed*

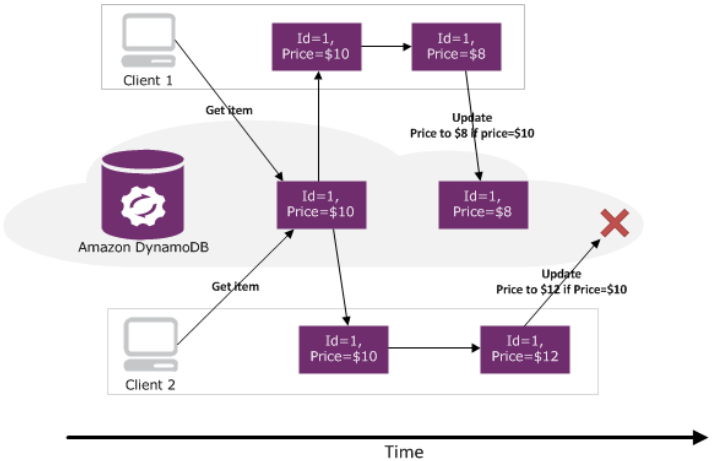
Paxos clusters

- ✓ Paxos algorithm comes as close as possible (in known CompSci) to achieving CAP
- ✓ CA distributed state machine in normal operation
- ✓ Multi-master, quorum based (can define quorum size, but must be at least bare majority) - Consensus
- ✓ Can tolerate any kind of partition so long as quorum is maintained (becomes "unavailable" for clients of non-quorum nodes)
- ✓ Formally proven to provide reliable distributed state transitions (updates, aka "availability")
  - ✓ But not infinite time
  - ✓ In practice, it works out ok

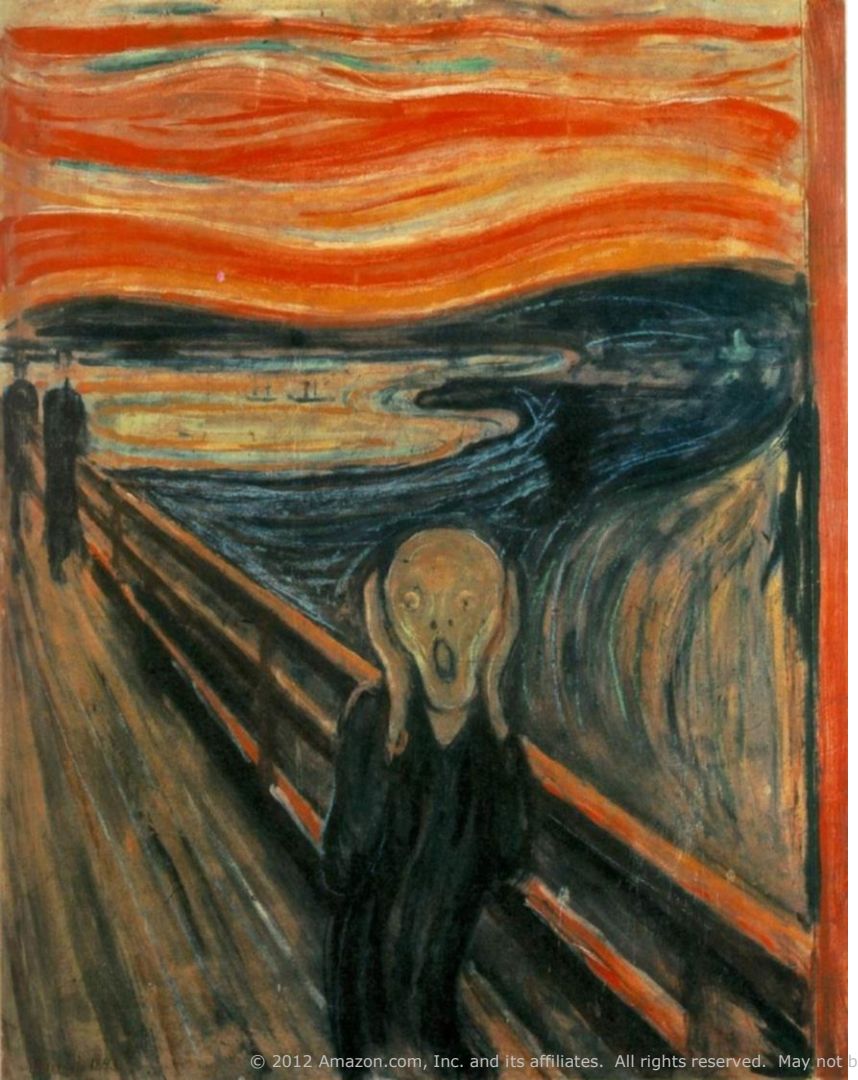


Paxos sounds tricky to implement…

- ✓ Shift that kind of work off to other systems
- ✓ E.g. DynamoDB conditional writes (idempotent too!)
  - ✓ Update only if the specified condition is met




```
// This updates the price only if current price is 10.00.
        expectedValues.put("Price",
                new ExpectedAttributeValue()
                    .withValue(new AttributeValue().withN("10.00")));
```



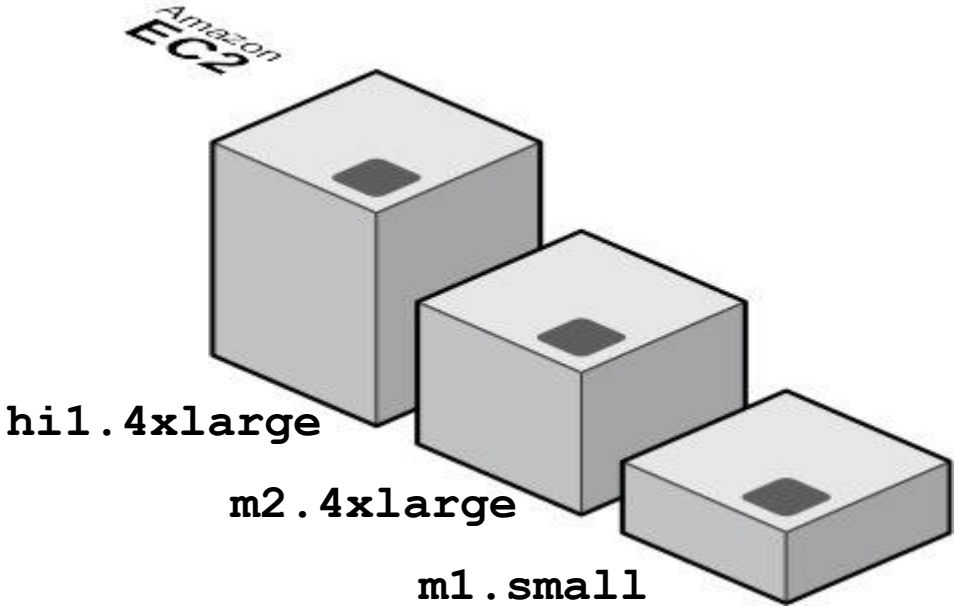
Data Tier Scalability

The bane of the Architect's existence

Vertical Scaling

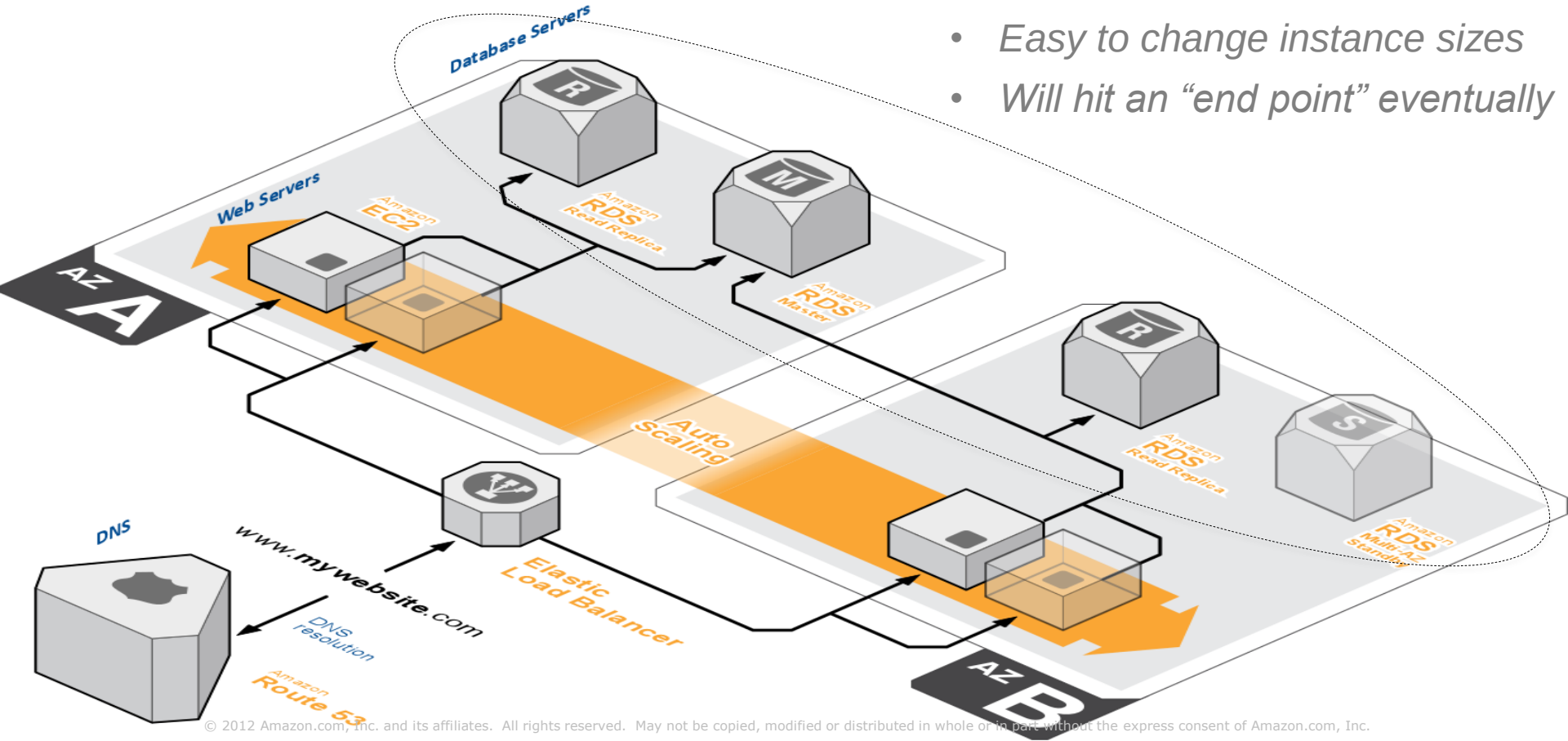
“We need a bigger box”

- *Simplest Approach*
- *Can now leverage PIOPS*
- *High I/O for NoSQL DBs*
- *Easy to change instance sizes*
- *Will hit an “end point” eventually*



Master/Slave Horizontal Scaling

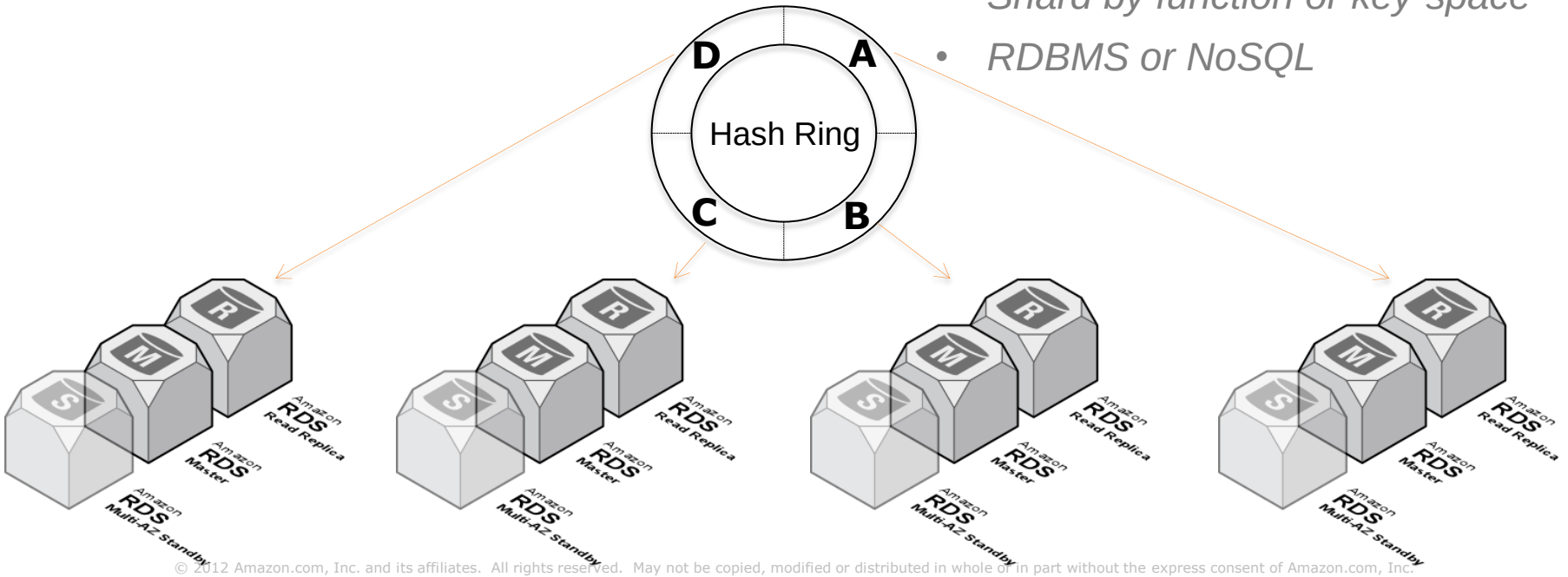
- *Reasonably simple to adapt to*
- *Can now leverage PIOPS*
- *Easy to change instance sizes*
- *Will hit an "end point" eventually*



Sharded Horizontal Scaling

"With great power comes great responsibility"

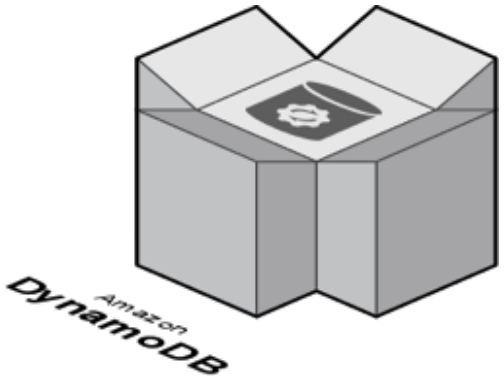
- *More complex at application layer*
- *ORM support can help*
- *No practical limit on scalability*
- *Operational complexity/sophistication*
- *Shard by function or key-space*
- *RDBMS or NoSQL*



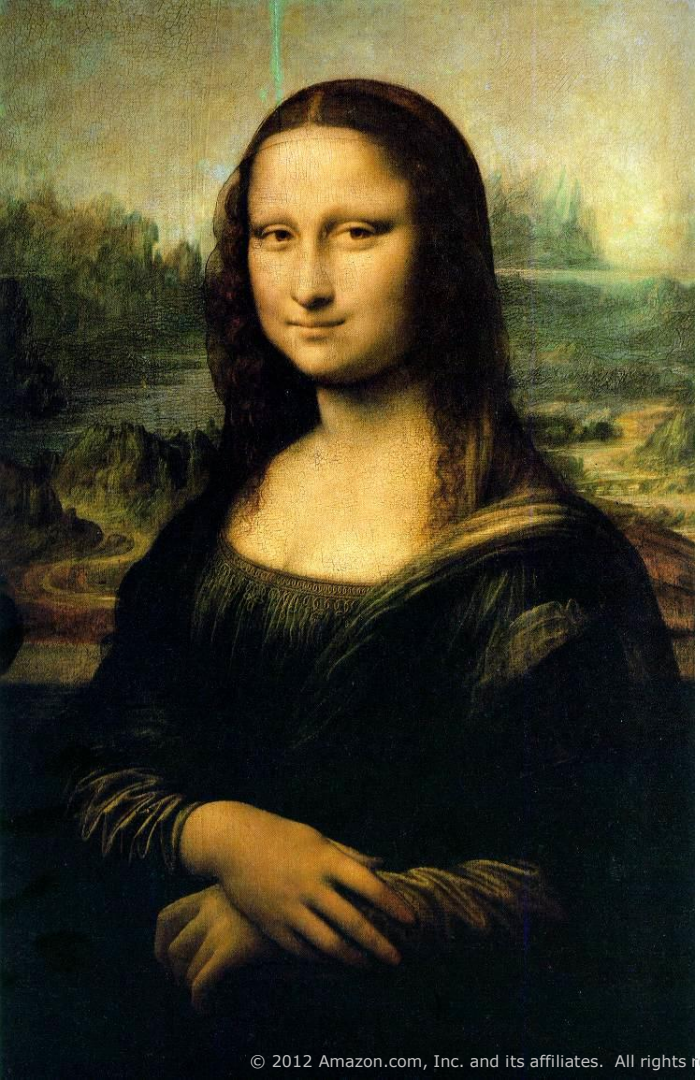
Horizontal Scaling – Fully Managed

DynamoDB

- *Provisioned throughput NoSQL database*
- *Fast, predictable performance*
- *Fully distributed, fault tolerant architecture*
- *Considerations for non-uniform data*



Feature	Details
Provisioned throughput	Dial up or down provisioned read/write capacity
Predictable performance	Average single digit millisecond latencies from SSD backed infrastructure
Strong consistency	Be sure you are reading the most up to date values
Fault tolerant	Data replicated across availability zones
Monitoring	Integrated to CloudWatch
Secure	Integrates with AWS Identity and Access Management (IAM)
Elastic MapReduce	Integrates with Elastic MapReduce for complex analytics on large datasets



Creating a Masterpiece for the Ages

Use these techniques (and many, many others)

SITUATIONALLY

AWARENESS

of the options is the first step to good design

SCALING

is the ability to move bottlenecks around to the least expensive part of the architecture

AWS

makes this easier – so your application is not the victim of its own success

For perspective…

Please welcome

James Hamilton

Top ~~10~~ 5 Ways to Early Post Mortem

5. We'll add monitoring & alerting as we get production experience & a baseline

4. We don't need incremental deployment for V1

3. We'll get into production & add automated testing before second release

2. All our test cases are passing

1. We can partition the database when needed – We have 10x capacity needed for first year

Thank You

Questions?

@simon_elisha

http://aws.amazon.com/podcast

#reinvent

We are sincerely eager to hear your **FEEDBACK** on this presentation and on re:Invent.

Please fill out an evaluation form when you have a chance.

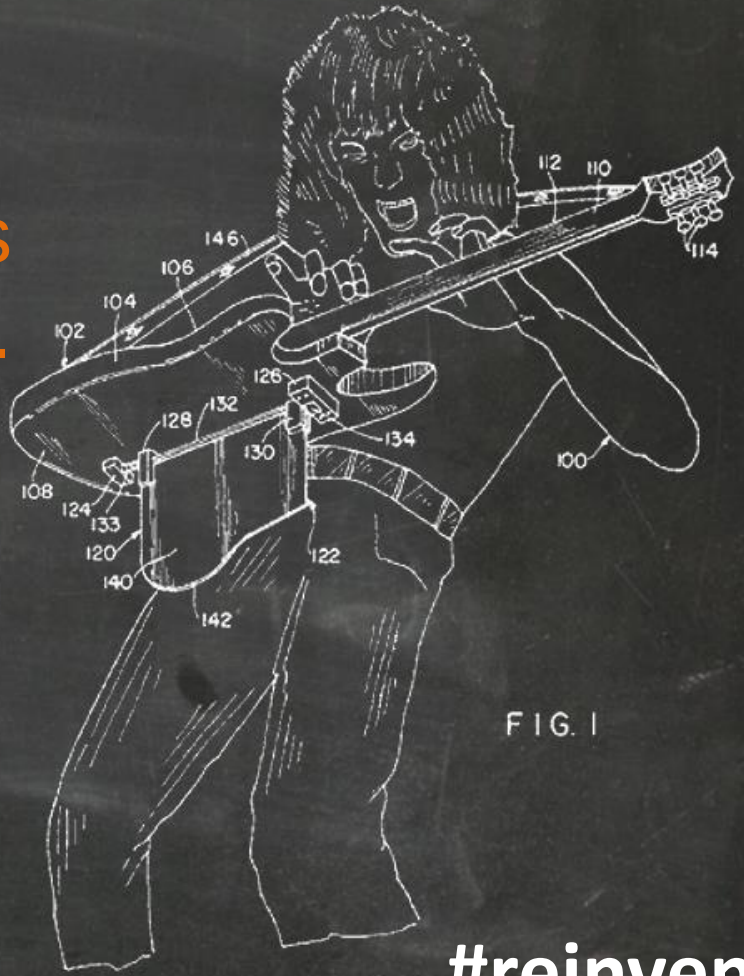


FIG. 1

#reinvent