AWS re:Invent

NOV. 29 – DEC. 3, 2021 | LAS VEGAS, NV

DAT304

Deep dive on Amazon ElastiCache for Redis

Joe Travaglini
Principal Product Manager
AWS

Lindsey Berg
Software Engineer
Groupon

Agenda

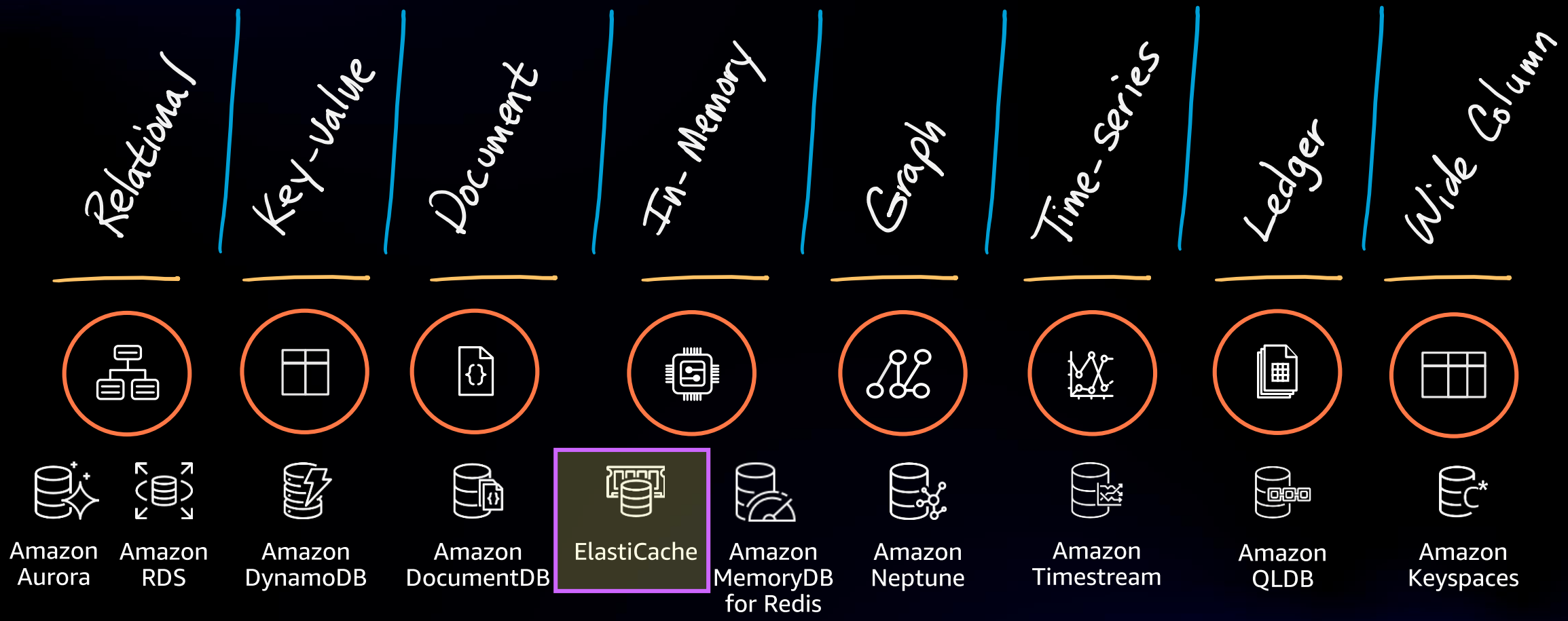
- AWS database services
- Amazon ElastiCache for Redis
- Groupon’s migration to ElastiCache
- Using Redis for deal curation at Groupon

aws

Overview of ElastiCache and AWS purpose-built databases

aws

Purpose-built databases



Amazon ElastiCache for Redis

Redis compatible

Fully compatible with
open-source Redis

Fully managed

AWS manages all hardware
and software setup,
configuration, monitoring

Highly available

Multi AZ with auto failover
Cross-Region replication
Cluster auto scaling

Extreme performance

In-memory data store
and cache for microsecond
response times

Secure and compliant

Network isolation, encryption
at rest/in transit, RBAC
HIPAA, PCI-DSS, FedRAMP

Easily scalable

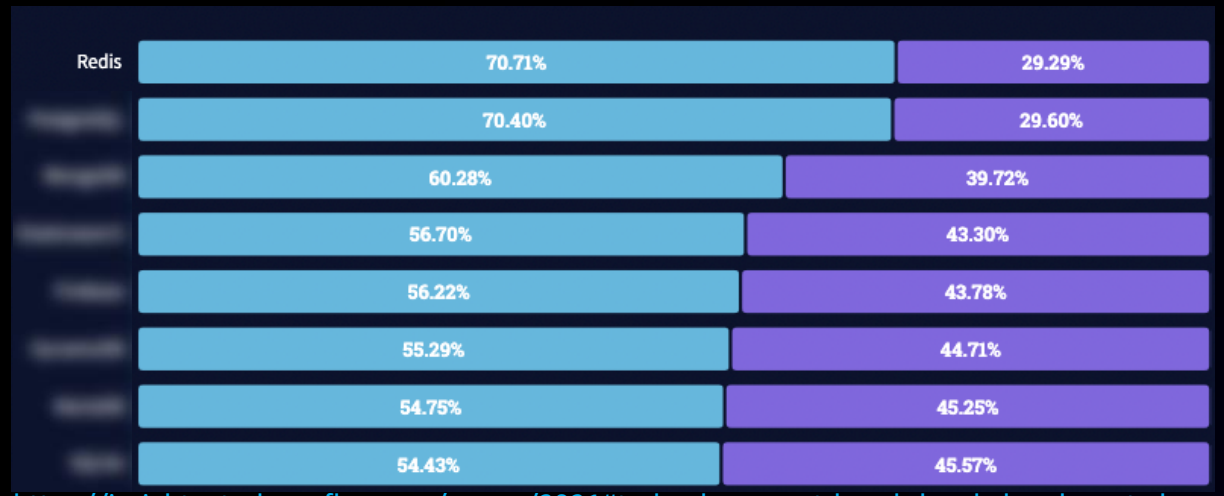
Scale reads with replicas
Scale writes with shards
Up to 500 nodes per cluster

aws

Redis – #1 in-memory data store, most loved

Rank Nov 2021	Rank Oct 2021	Rank Nov 2020	DBMS	Database Model	Score Nov 2021	Score Oct 2021	Score Nov 2020
1.	1.	1.	Redis	Key-value, Multi-model	171.50	+0.15	+16.08
2.	2.	2.	[illegible]	Multi-model	76.99	+0.43	+8.09
3.	3.	3.	[illegible]	Multi-model	40.82	+0.54	+8.32
4.	4.	4.	[illegible]	Key-value	26.37	+0.35	+0.62
5.	5.	↑ 6.	[illegible]	Key-value	10.81	+0.62	+1.99
6.	6.	↓ 5.	[illegible]	Key-value, Multi-model	10.00	+0.34	+0.23

https://db-engines.com/en/ranking/key-value+store



https://insights.stackoverflow.com/survey/2021#technology-most-loved-dreaded-and-wanted

Developers love Redis because its

1. Blazing fast
2. Versatile
3. Feature-rich and easy to use

aws

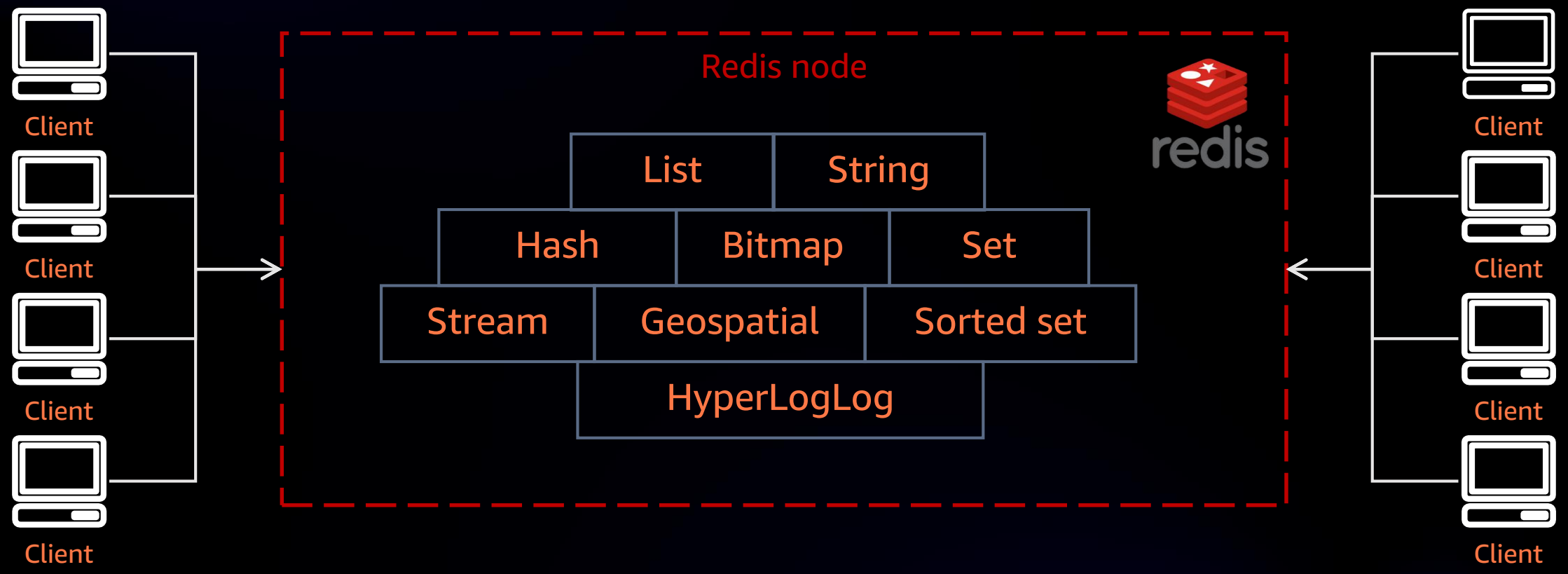
Redis data types

Data type	Description	Example use case	Associated commands
String	Sequence of bytes	Caching	APPEND,GET,SET,INCR,DECR,GETSET...
List*	A list of strings	Many	LSET,LLEN,LPUSH,LPOP,LTRIM,RPOP...
Set*	Non-repeating, unordered collection of strings	Cardinality	SADD,SCARD,SDIFF,SUNION,SINTER,SMEMBERS...
Sorted set*	Non-repeating, ordered collection of strings	Leaderboards	ZADD,ZCARD,ZCOUNT,ZRANK,ZSCORE...
Hash*	Map of key/value pairs	Session store	HGET,HGETALL,HKEYS,HVALS,HMSET,HMGET...
Streams	Log data structure	Message queue	XADD,XRANGE,XREAD,XACK,XCLAIM,XLEN...
Geospatial	Longitude/latitude-based entries	Maps, "nearby"	GEOADD,GEODIST,GEOPOS,GEORADIUS...
Bitmaps	Special usage of string type		GETBIT,BITCOUNT,SETBIT,SETRANGE,GETRANGE...
HyperLogLogs	Special usage of string type		PFADD,PFCOUNT,PFMERGE

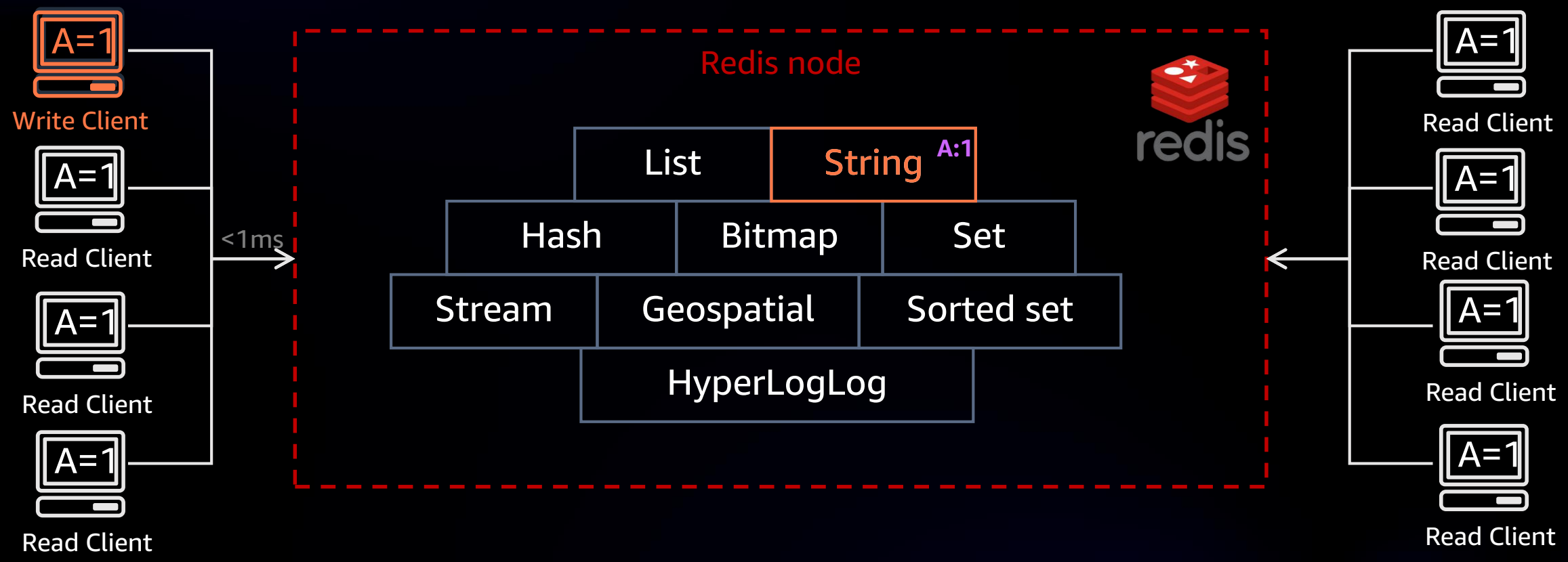
** These data types are capable of storing over 4 billion subelements*

aws

ElastiCache for Redis: Distributed in-memory store

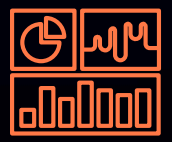


ElastiCache for Redis: Distributed in-memory store



ElastiCache for Redis use cases

Caching



Real-time analytics

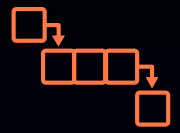
Gaming leaderboards

Geospatial

Media streaming

Session store

Chat apps



Message queues

Machine learning

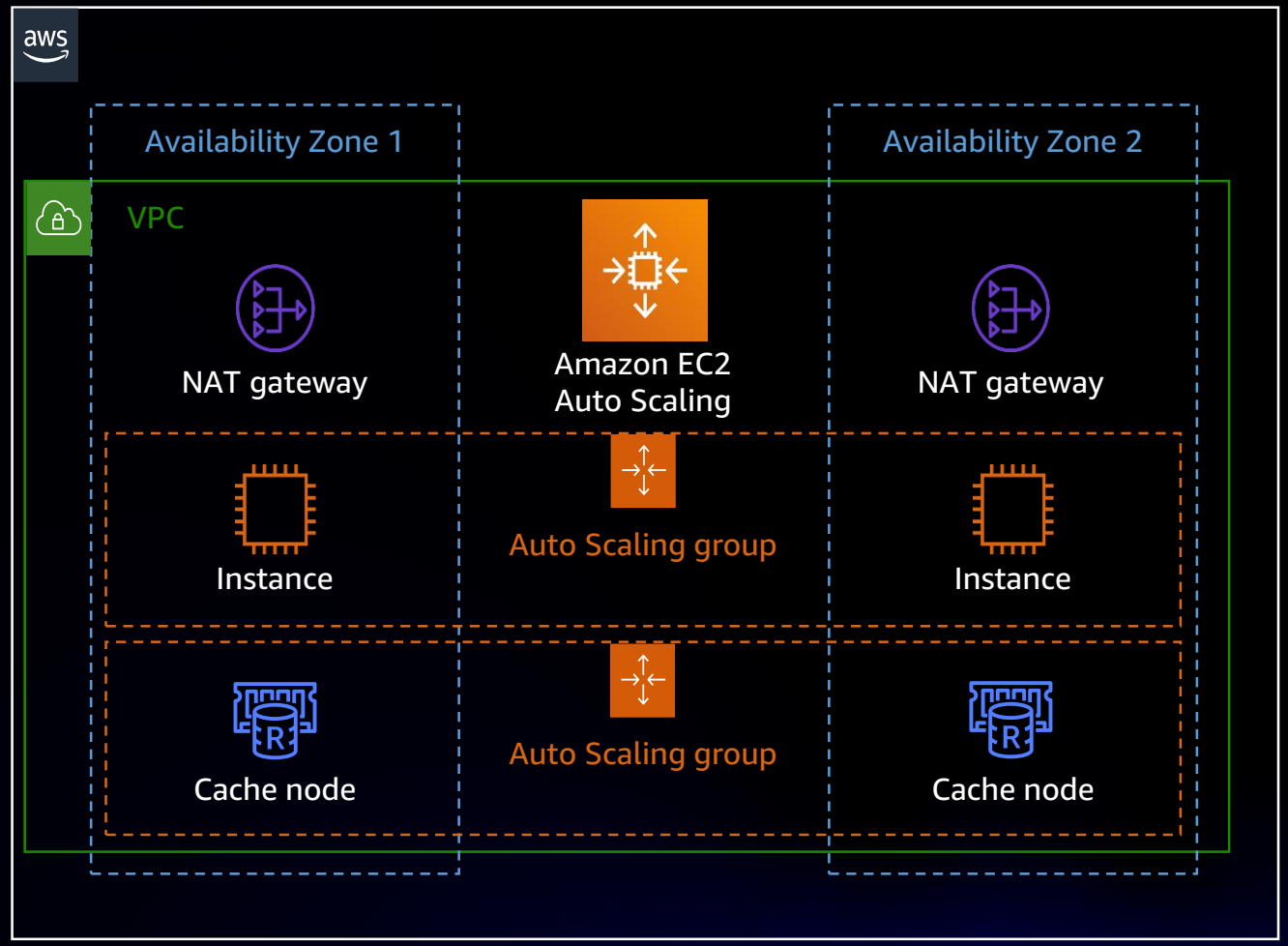
aws

What’s new with Amazon ElastiCache for Redis

aws

ElastiCache for Redis auto scaling

- Automatically add shards or replicas to your cluster
- Use predefined rules or Amazon CloudWatch metrics to horizontally scale in and out
- Schedule scaling activities for predictable workload and capacity changes



Example: Add read replicas under load

config.json

```
{
  "TargetValue": 50,
  "CustomizedMetricSpecification":
  {
          "MetricName": ”EngineCPUUtilization",
          "Namespace": "AWS/ElastiCache",
          "Dimensions": [
            {
              "Name": ”ReplicationGroup",
              "Value": ”autoscaling-demo"
            },
            {
              "Name": ”Role",
              "Value": ”REPLICA"
            }
           ],
          "Statistic": ”Average",
          "Unit": ”Percent”
   }
 }
```

This auto scaling policy triggers when the CloudWatch metric for CPU utilization crosses 50% on average for all replicas in the cluster named autoscaling-demo

Example: Add read replicas under load

Create a scalable target for your ElastiCache cluster with application auto scaling

```
aws application-autoscaling register-scalable-target \
   --service-namespace elasticache \
   --scalable-dimension elasticache:replication-group:Replicas \
   --resource-id replication-group/autoscaling-demo \
   --min-capacity 1 --max-capacity 5
```

Apply the policy you just created to this auto scaling resource

```
aws application-autoscaling put-scaling-policy \
   --policy-name myscalablepolicy \
   --policy-type TargetTrackingScaling \
   --resource-id replication-group/autoscaling-demo \
   --service-namespace elasticache \
   --scalable-dimension elasticache:replication-group:Replicas \
   --target-tracking-scaling-policy-configuration file://config.json
```

aws

Auto scaling best practices

Keep it simple

- Recommend using a single metric and dimension
- Start with predefined metrics before custom ones

Avoid churn

- Works best with uniform distribution across replicas/shards
- Limit scale-in, use cooldowns, schedule where appropriate

Test!

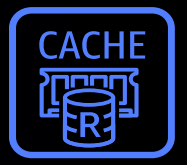
- Use at least 4 weeks of data to set target values

aws

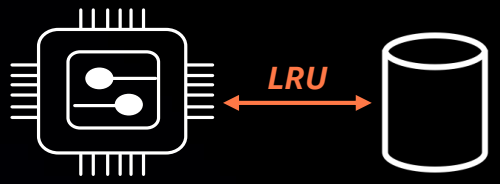
Introducing R6gd nodes with data tiering

- Ideal for workloads that access up to 20% of their dataset regularly
- Save over 60% per GB at full capacity utilization
- No application changes required, minimal performance impact
- Clusters scale up to 1 PiB in size

Example: R6gd.16xlarge



64 vCPU
25 Gbps network



RAM	SSD
419.1 GiB memory	1592.56 GiB NVMe

2011.66 GiB total capacity

Cost-effectively scale with R6gd

with data tiering

	R6g.xlarge	R6gd.xlarge
vCPU	4	4
RAM (GiB)	26.32	26.32
SSD (GiB)	0	99.33
Total capacity (GiB)	26.32	**125.65**
Hourly node price*	$0.411	$0.781
Price per GiB-hour*	**$0.016**	**$0.006**

4.8x capacity

over **60%** savings!

Offered in xlarge, 2xlarge, 4xlarge, 8xlarge, 12xlarge, and 16xlarge sizes

*Displayed pricing for on-demand nodes in the US East (N. Virginia) Region

aws

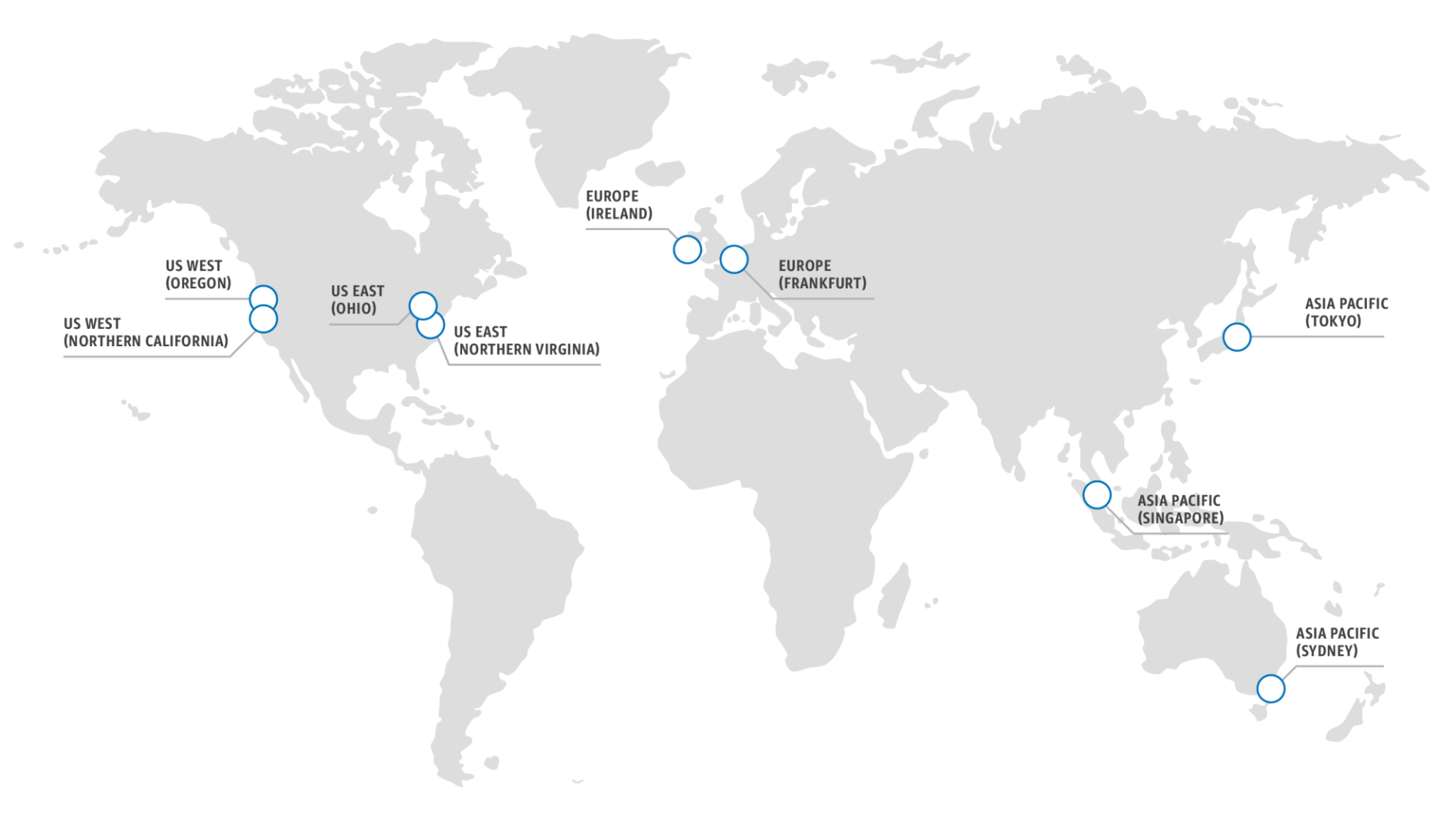
Data tiering node specs

Node type	vCPU	Network bandwidth	RAM (GiB)	SSD (GiB)	Total capacity (GiB)	Node hourly price*	Price* per GiB-hour
R6gd.xlarge	4	Up to 10 Gbps	26.32	99.33	125.65	$0.781	$0.006
R6gd.2xlarge	8	Up to 10 Gbps	52.82	199.07	251.89	$1.560	$0.006
R6gd.4xlarge	16	Up to 10 Gbps	105.81	398.14	503.95	$3.120	$0.006
R6gd.8xlarge	32	12 Gbps	209.55	796.28	1005.83	$6.240	$0.006
R6gd.12xlarge	48	20 Gbps	317.77	1194.42	1512.19	$9.358	$0.006
R6gd.16xlarge	64	25 Gbps	419.10	1592.56	2011.66	$12.477	$0.006

*Displayed pricing for on-demand nodes in the US East (N. Virginia) region

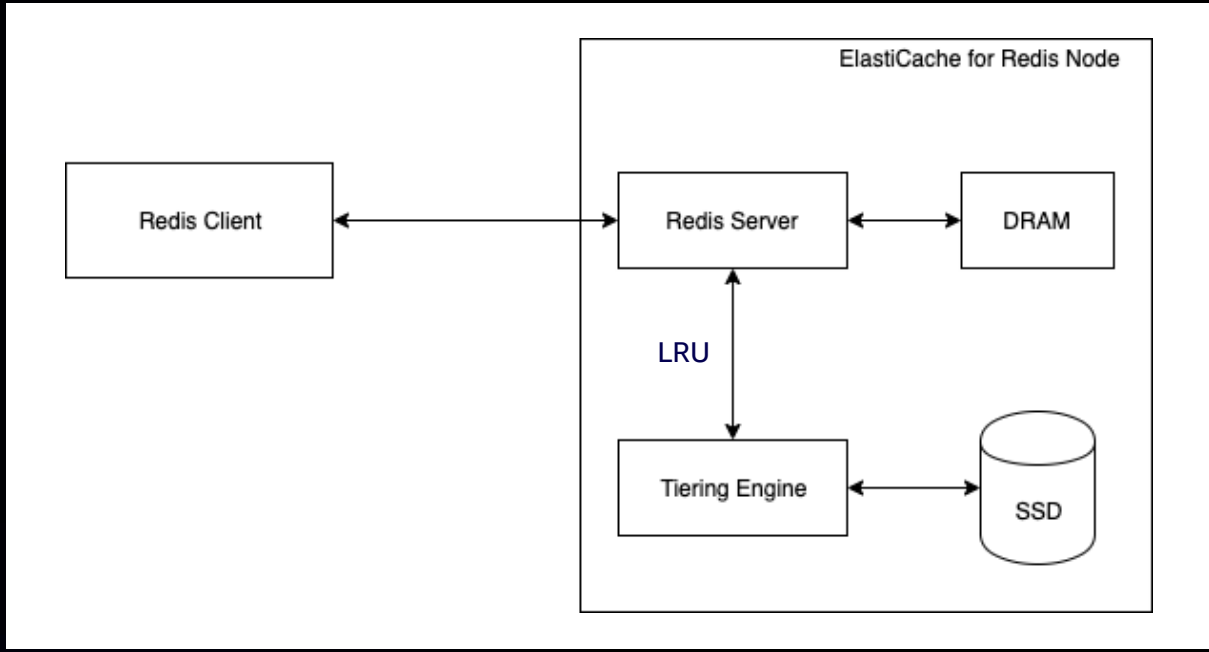
aws

ElastiCache for Redis R6gd Region availability

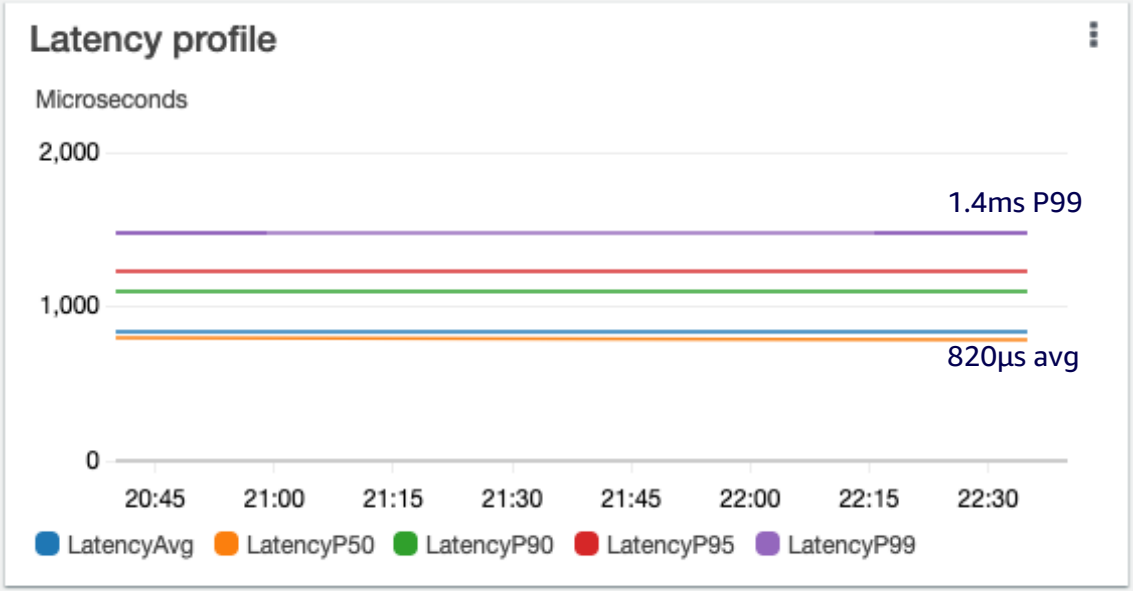


How data tiering works

- Fixed, 16-byte overhead per key
- Operates atomically at item level
- Keys are always stored in memory
- Asynchronous communication between Redis engine and tiering engine



Performance review



- R6xd.2xlarge
- 396M unique keys
- 16 byte keys, 500 byte string values
- 200 client connections
- 4:1 get:set ratio
- 10% of workload hits SSD tier

Monitoring with Amazon CloudWatch

4 **new** metrics

- NumItemsWrittenToDisk
- NumItemsReadFromDisk
- BytesWrittenToDisk
- BytesReadFromDisk

New metric **dimensions**

- CurrItems
  - Tier=Memory
  - Tier=SSD
- BytesUsedForCache
  - Tier=Memory
  - Tier=SSD

aws

Monitoring with Amazon CloudWatch

Current limitations

- Items larger than 128 MB are not eligible for tiering

  These items will always remain in memory

- Supported maxmemory-policy

  noeviction, allkeys-lru, and volatile-lru

- Cluster management

  Online migration, modifying node family, exporting backup, and auto scaling are not supported at launch

aws

Data tiering in a nutshell

- Use R6gd with data tiering to cost-effectively scale large capacity Redis workloads
- Ideal for applications that access up to 20% of data regularly
- Minimal performance impact, transparent to applications
- Available today in 9 AWS Regions

ROKT

*"ElastiCache allows Rokt to **handle our dataset's hyper growth** without impacting our clients' user experience. Data tiering is perfect for us, as we can store 5 times the data per node, and the **performance impact was virtually unnoticeable** given the nature of our workload. AWS enables Rokt to focus on product innovation rather than rethink underlying infrastructure which is paramount as leaders in the ecommerce technology space."*

Corey Bertram, CTO – Rokt

aws

Customer success with Amazon ElastiCache

Software and internet

Ride hailing

Financial services

Gaming

Media and entertainment

Social media

Telecommunications

Travel

Industrials and services

Logistics & operations

Publishing

Other

Amazon ElastiCache for Redis at Groupon

GROUPON

Platform Engineering | re:Invent 2021
Author | Lindsey Berg

aws

Groupon is an experiences marketplace where consumers discover fun things to do and local businesses thrive




For our customers, this means giving them an amazing selection of experiences at great values

For our merchants, this means making it easy for them to partner with Groupon and reach millions of consumers around the world

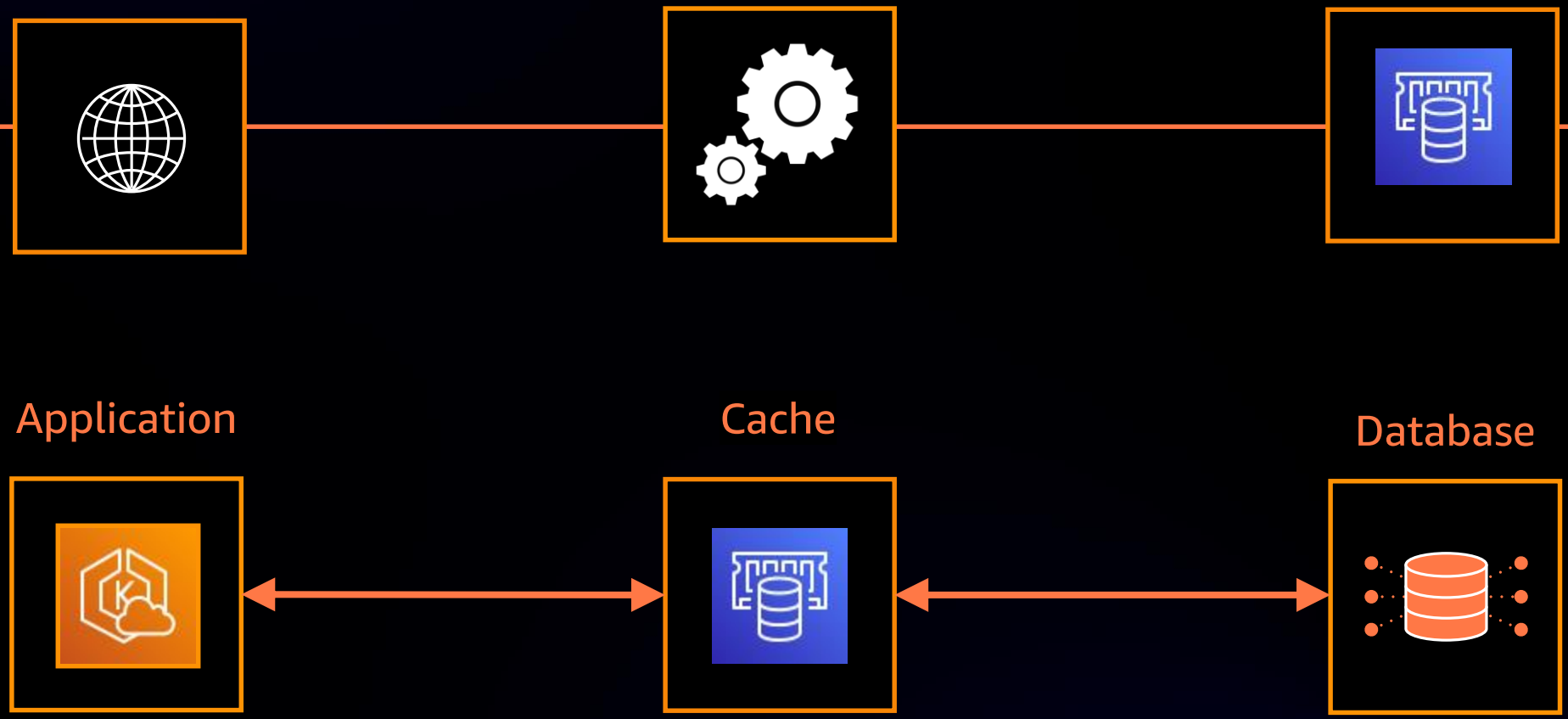
aws

Groupon technology ecosystem

24M customers

600 microservices

127 active caches



Flow of today's discussion

Cloud migration journey

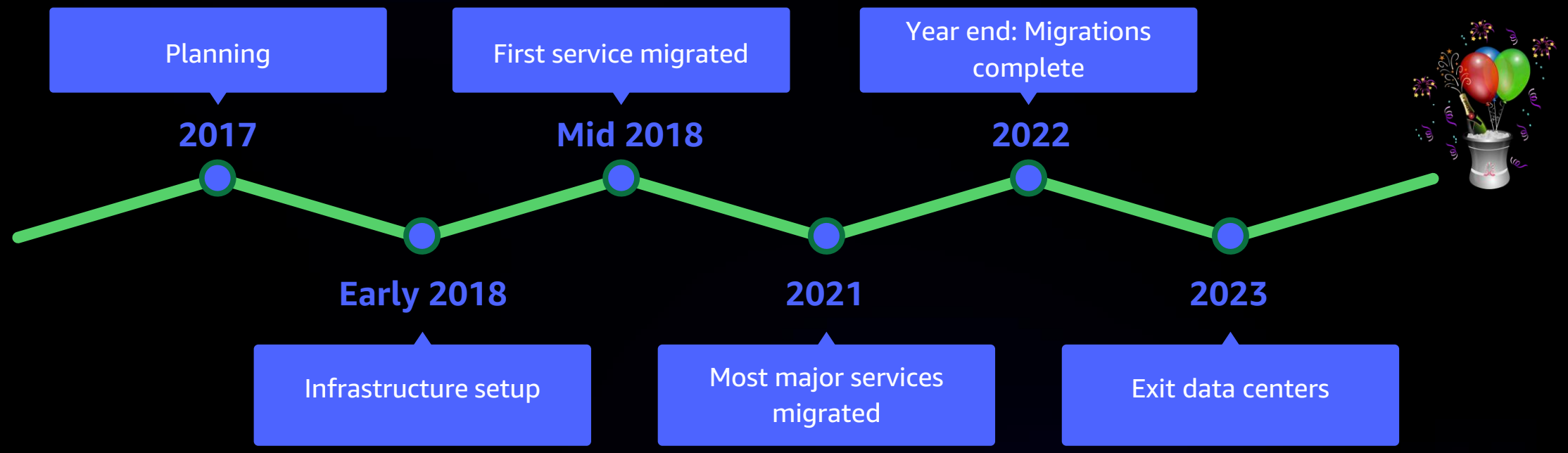
Redis journey

Lessons learned

What's next

Cloud migration journey

DRIVING FACTORS: COST, REDUCED ADMINISTRATION BURDEN, MODERNIZATION



aws

Modernize | Simplify | Standardize | Reduce costs

Before Redis

- Lots of flexibility
- DevOps model, owning from the OS & up
- Caching was front-end heavy
- Continued growth pushed limits

New needs

- Specialization, standardization, centralization
- Build in resilience and recovery
- Sharding offered
- Reduce costs
- Free teams to focus on new products, drive new revenue streams

aws

Modernize | Simplify | Standardize | Reduce costs

20% of our services use Redis today

- A cache
  - In-memory key-value store
  - Results of database calls
  - Results of API calls
  - Cache calls to third party services that have fees or throttled queries
  - Page rendering
- A message broker
- System configuration storage

aws

Why ElastiCache and not another solution

- Cost control
- Better visibility
- Increase in technical control

aws

Why ElastiCache

Cost

- Opportunity to move away from paid subscription
- Managed service – pay for what you use

Monitoring

- Tagging adds visibility – Which service? Which cost center?
- Be smarter about our data by deleting wasteful caches

Scaling

- Optimize costs

aws

Why ElastiCache

Quality of life

Why ElastiCache

Quality of life

Why ElastiCache: Upgrade control

Previous

- Required upgrade for all services every year
- Took a full resource month
- High risk: all or nothing

Now

- Have control over this process
- Per-service basis
- Upgrade when it’s right for us

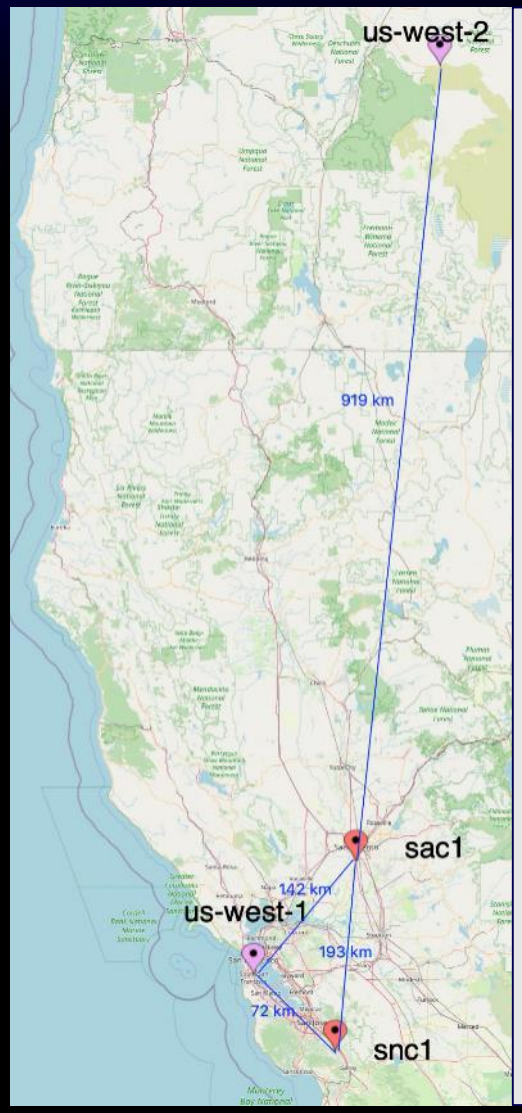
aws

Changes needed to migrate

Don’t be scared; they were easily mitigated!

Being aware removes room for doubt about our decision

Migration to ElastiCache: Temporary latency



On-premises
SNC1
(Santa Clara, CA)

On-premises
SAC1
(Sacramento, CA)

On-premises
DUB1
(Dublin, Ireland)

Latency
31.0 ms to 32.8 ms

Latency
24.1 ms to 25.8 ms

Latency
1.9 ms to 3.1 ms

Latency
4.2 ms to 5.2 ms

Latency
1.3 ms to 1.9 ms

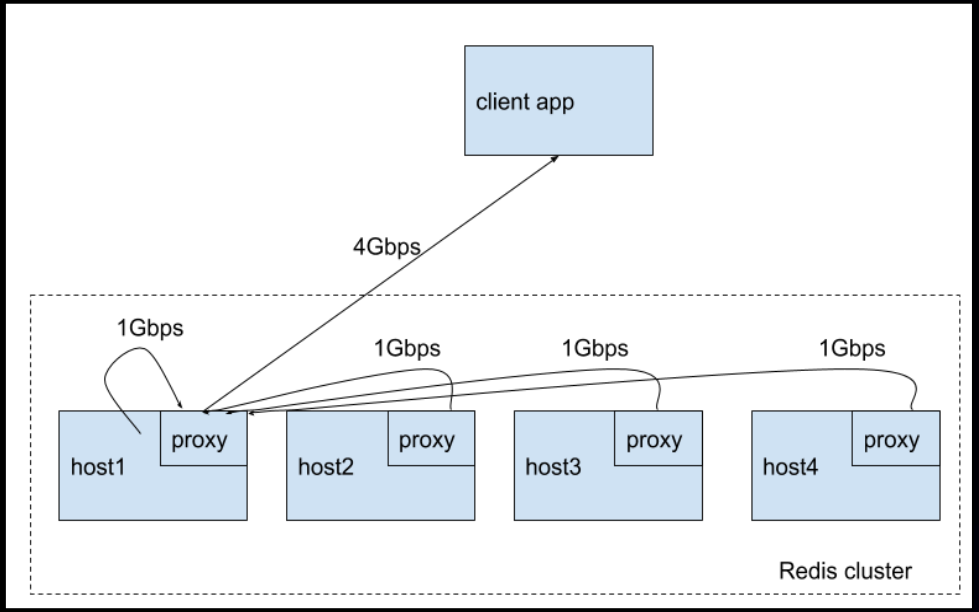
AWS us-west-2
(Oregon)
[only for staging, not recommended for prod]

AWS us-west-1
(N. California)

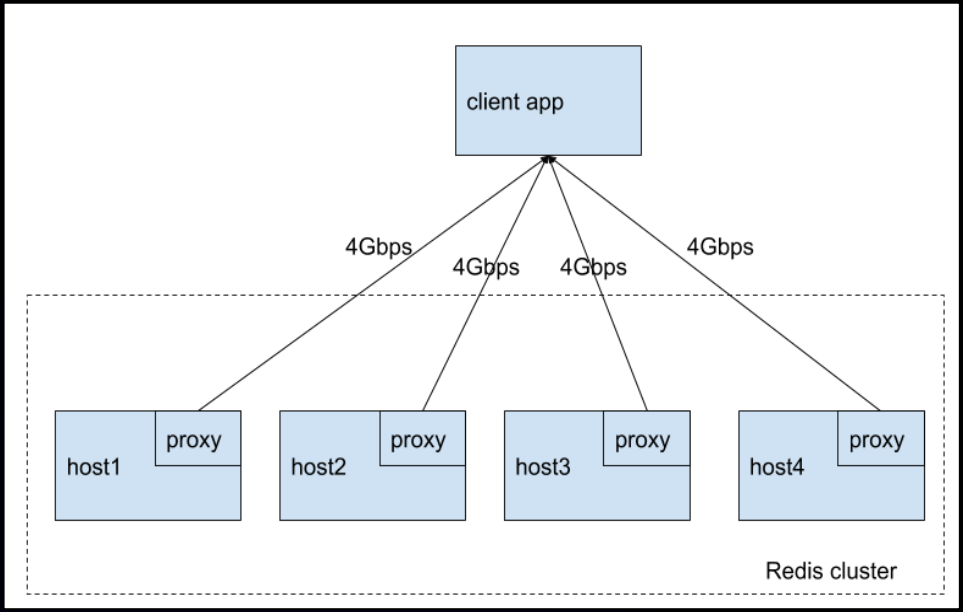
AWS eu-west-1
Europe (Ireland)

Migration: Cluster-aware clients

Smart proxy



Cluster-aware client



Migration to ElastiCache: Cost analysis

	Biggest AWS nodes	Smallest AWS nodes
2018 costs	~$74/year-GB	~$170/year-GB
	300 GB X 4	1100 GB
	$89,280 / year	$123,000 / year

Difference of

$33,720 / year

Far less than the cost of our annual subscription!

aws

ElastiCache migration milestones

Mid-2018: Comparing options, estimating costs

Late 2020: Platform setup

End of Q2 2021: All Redis instances migrated

Planning

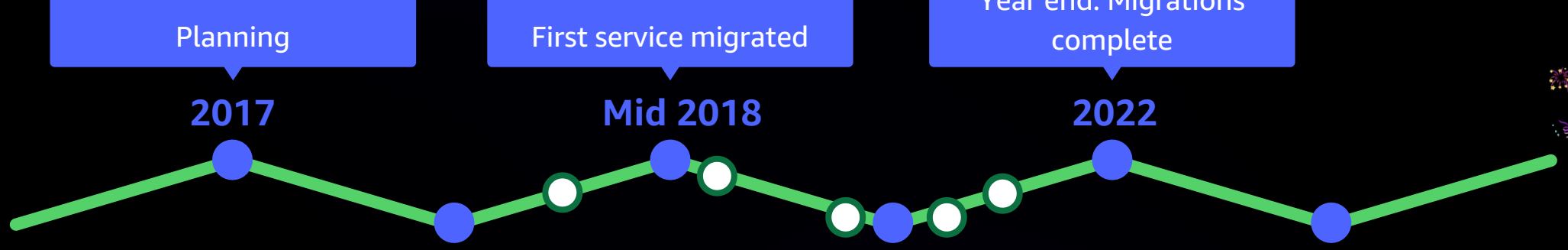
First service migrated

Year end: Migrations complete

2017

Mid 2018

2022



Early 2018

2021

2023

Infrastructure setup

Most major services migrated

Exit data centers

Early 2019: Final proposal for ElastiCache

Q1 2021: Customer engagement & provisioning

aws

Smooth migration

- Cache warmed itself
- No re-engineering required
- Configuration changes only
- Downstream services didn’t notice the change

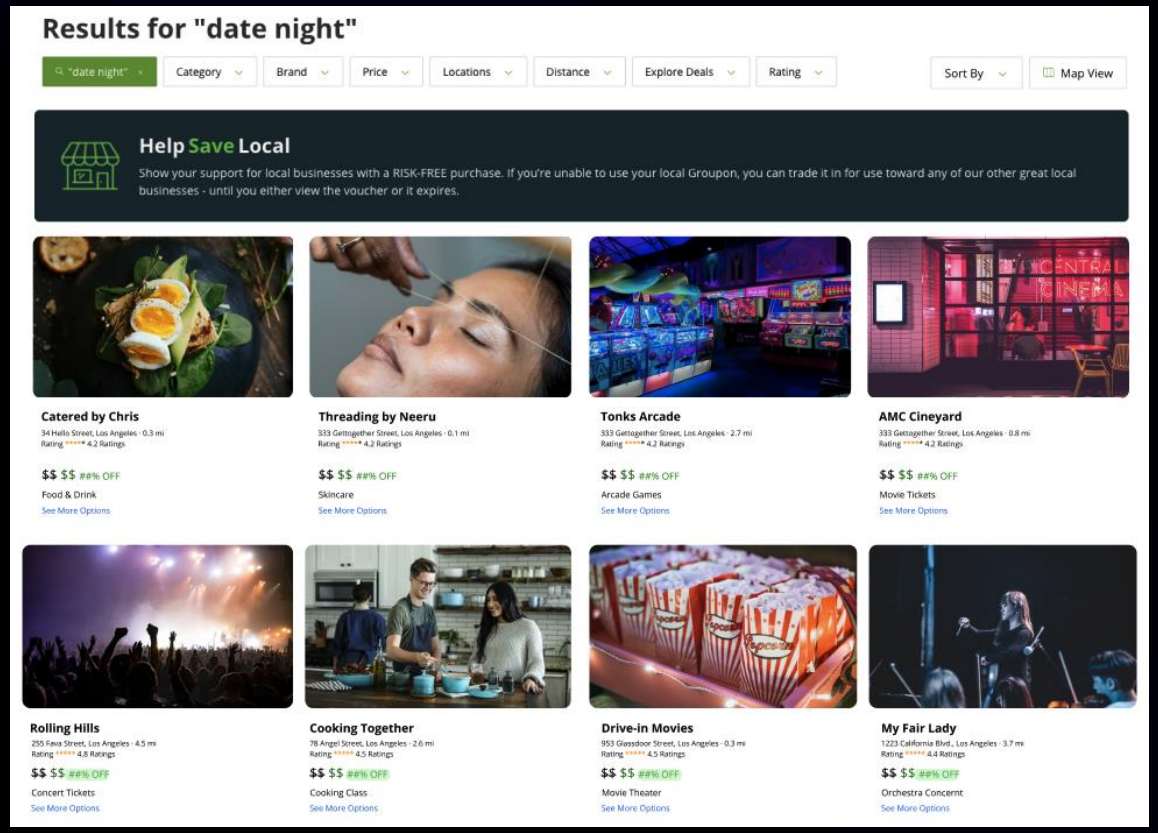
We migrated a total of 127 caches

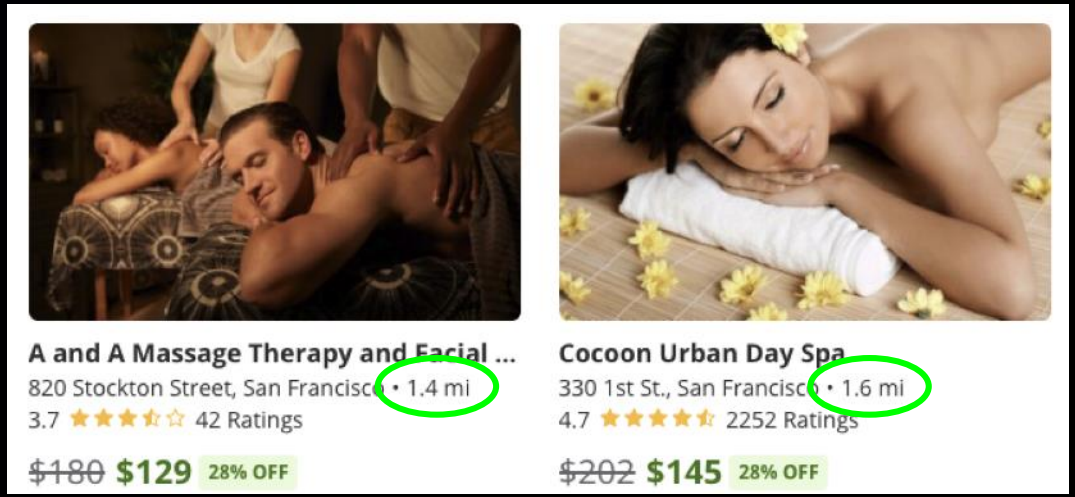
Decommed another 51 caches

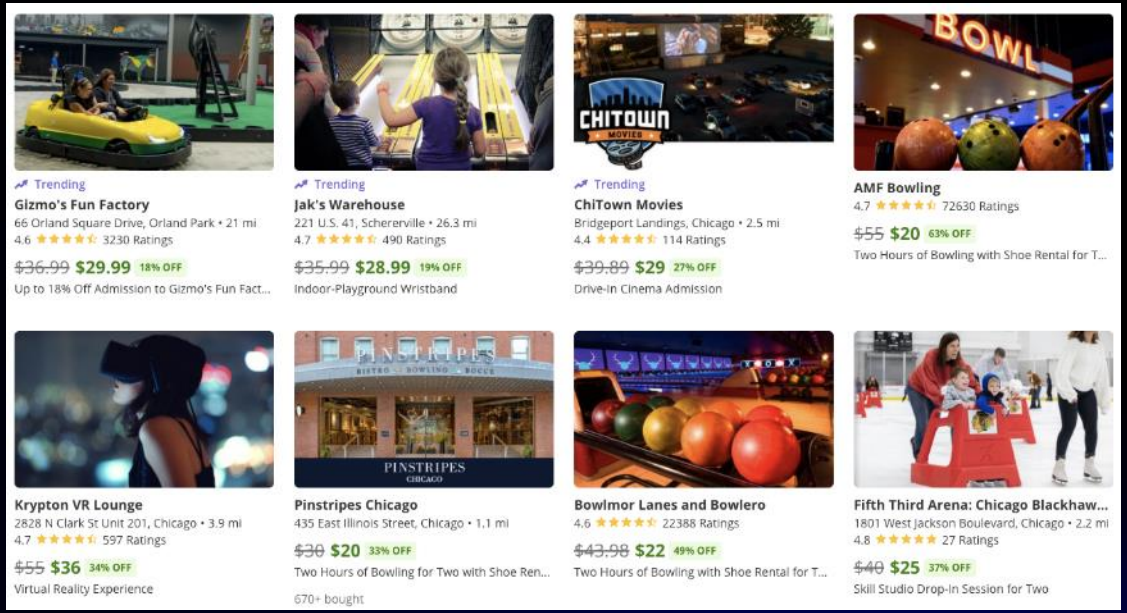
All in two quarters with only 2 engineers!

aws

Better user experience





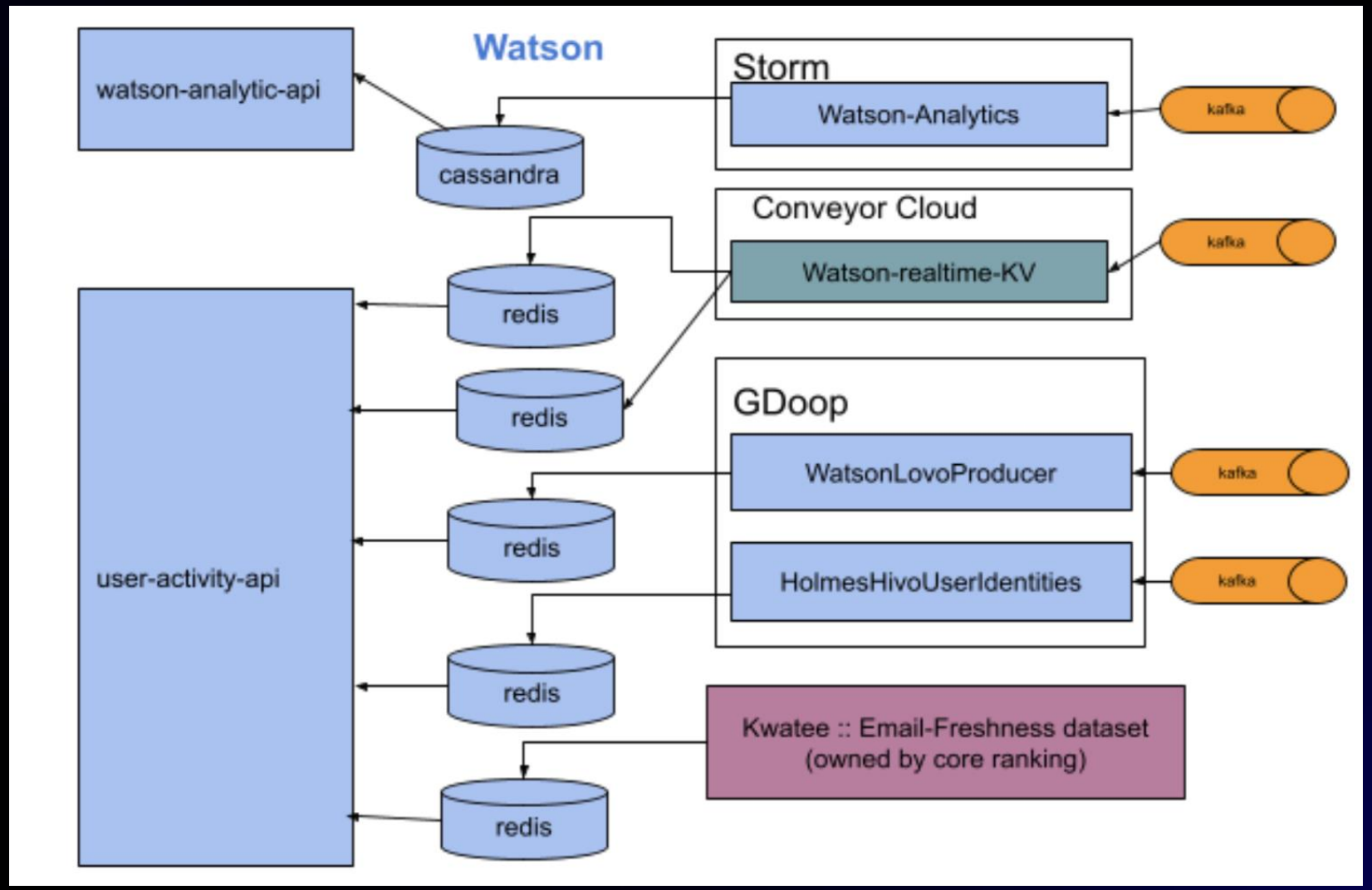


Watson

- Curates deals to show to each user depending on user and deal contexts
- Real-time lookup of user-indexed and deal-indexed data

aws

Watson architecture

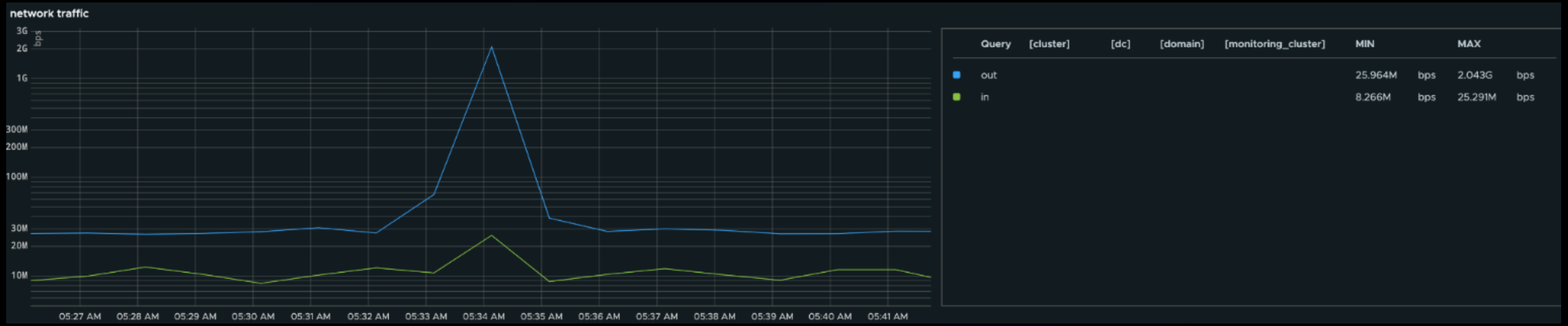


Watson migration: High network needs

2 GB/sec out, 25 MB/sec in

Size: 28 GB

Choice: 5x cache.r6g.large



Watson migration: High memory needs

Size: 940 GB

Choice: 36x cache.r6g.2xlarge

Watson migration: Cutover constraints

- Database statistics
  - ~500 GB of data
  - 248 shards
  - TTL of 1 year
- Communication across multiple teams
- Could incur no data loss
- On-premises instances available for reads
- Minimize time writes were paused

Goal: Complete within 2-hour time frame

aws

Watson migration: Cutover tools

- First benchmark attempt took 8 hours to complete

  → look for new solution

- Sysadmin command line tools
  - Bash
  - redis-cli – open-source Redis CLI
  - rdbtools – Python package

- Spare CentOS host

aws

Watson migration: Cadence

```
# for each new cluster slot, do a mass insertion of all shard dumps
# let the preconfigured nodes decide which keys they'll keep
rdb --command protocol $rdb | redis-cli --pipe -h 127.0.0.1
redis-cli --cluster call 127.0.0.1:10050 BGSAVE
```

Pause writes to the Redis databases

Dump all shards to flat files

Split the rdb files (backups) into cluster slots

Copy the processed rdb files to AWS

Update hosts to use ElastiCache for Redis

Resume writes to the Redis databases

```
# get the list of nodes
# from the nodes get the list of shards
# for each shard BGSAVE
```

```
rdb --command protocol $rdb | redis-cli --pipe -h $EC instance
```

aws

Watson migration: Cluster-aware clients

Standalone mode

Jedis

- Synchronous only
- Small and considerably faster

```
# single thread
Jedis j = new Jedis("localhost");

# pooling that is thread safe
final JedisPoolConfig poolConfig = buildPoolConfig();
JedisPool jedisPool = new JedisPool(poolConfig, "localhost");
```

Lettuce

- Both sync and async/thread safe
- Transparent reconnection handling

```
# note the package names are io.lettuce.core.*
import io.lettuce.core.api.StatefulRedisConnection;
import io.lettuce.core.api.async.RedisAsyncCommands;
import io.lettuce.core.api.reactive.RedisReactiveCommands;
import io.lettuce.core.api.sync.RedisCommands;

RedisClient redisStandaloneClient = RedisClient.create("redis://localhost");
StatefulRedisConnection<String, String> connection =
redisStandaloneClient.connect();

RedisCommands syncCommands =  connection.sync();
RedisAsyncCommands asyncCommands = connection.async();
RedisReactiveCommands reactiveCommands = connection.reactive();
```

aws

Watson migration: Cluster-aware clients

Cluster mode

Jedis

```
Set<HostAndPort> jedisClusterNodes = new HashSet<HostAndPort>();
# add connection string to the master
jedisClusterNodes.add(new HostAndPort("127.0.0.1", 6379));
JedisCluster jc = new JedisCluster(jedisClusterNodes);
```

Lettuce

```
# note the package names are io.lettuce.core.cluster.*
import io.lettuce.core.cluster.api.StatefulRedisClusterConnection;
import io.lettuce.core.cluster.api.async.RedisAdvancedClusterAsyncCommands;
import io.lettuce.core.cluster.api.reactive.RedisAdvancedClusterReactiveCommands;
import io.lettuce.core.cluster.api.sync.RedisAdvancedClusterCommands;

RedisClusterClient redisClusterClient = RedisClusterClient.create("redis://localhost");
StatefulRedisClusterConnection<String, String> connection = redisClusterClient.connect();

RedisAdvancedClusterCommands syncCommands =  connection.sync();
RedisAdvancedClusterAsyncCommands asyncCommands = connection.async();
RedisAdvancedClusterReactiveCommands reactiveCommands = connection.reactive();
```

aws

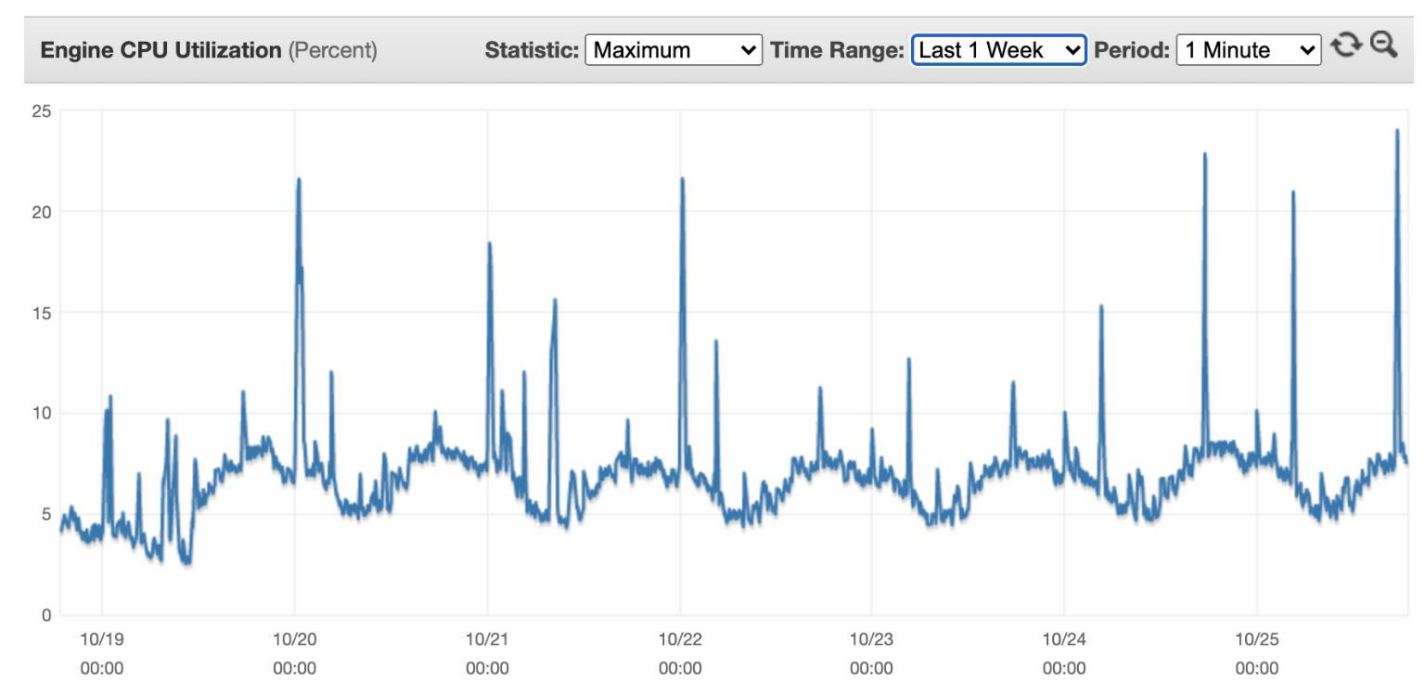
Auto scaling candidates

Inclusion criteria

- Most expensive
- Less than 40% of their memory utilized

Exclusion criteria

- Frequent CPU spikes
- Fast fluctuations in memory
- High network activity



[server]	MIN		MAX	
	19.522	%	21.347	%
	34.569	%	34.596	%
	13.609	%	13.613	%
	35.413	%	35.600	%
	18.172	%	18.206	%
	10.256	%	10.280	%
	10.253	%	10.274	%
	16.465	%	16.517	%
	33.719	%	33.934	%
	16.339	%	16.367	%

Watson: Auto scaling candidate

Ideal candidate

- 36 cache.r6g.2xlarge nodes
- 1.4 TB total capacity
- Only storing 382 GB

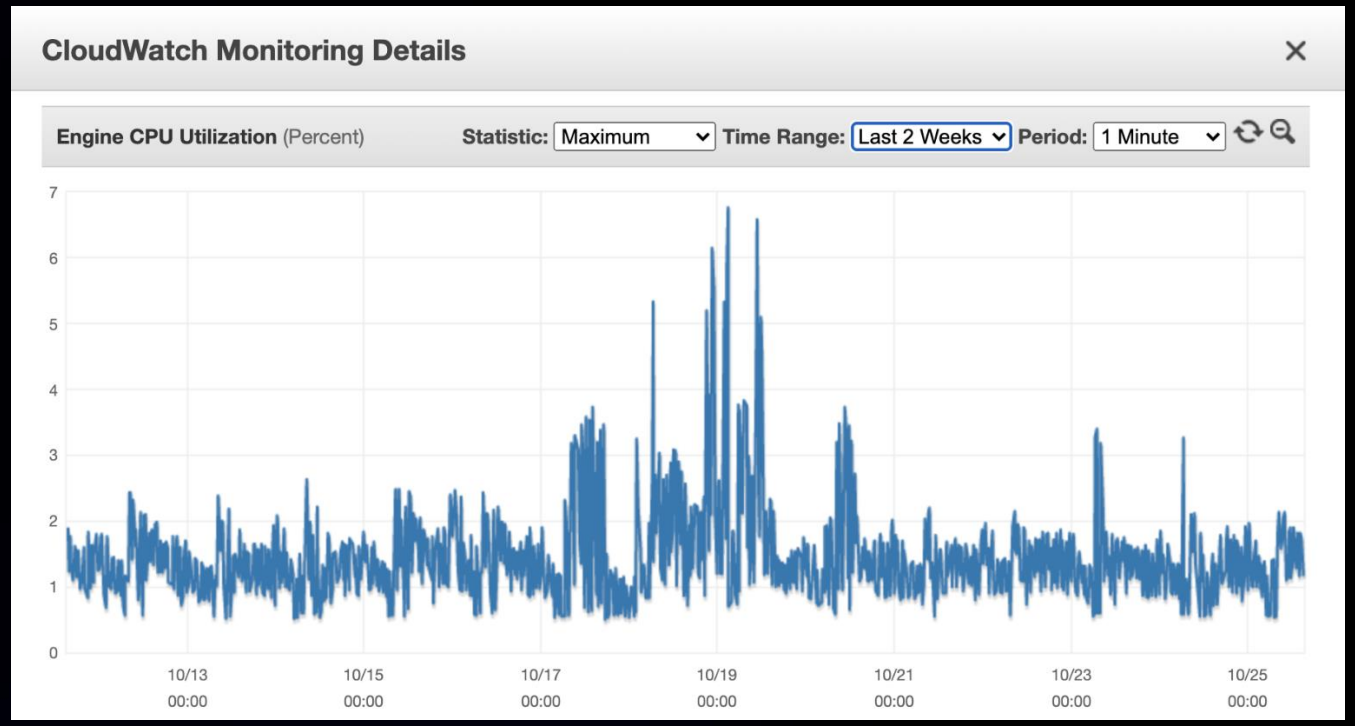
But what should the scaling policy be?

aws

Watson: Auto scaling policy

Goal: keep CPU below 25%

Choice: min 10 nodes



Watson: Auto scaling policy

Goal: max scale rate of 1 node/hour

Watson: Auto scaling policy

Goal: min set of nodes does not peg the NIC

Watson: Auto scaling policy

Before

- 36 cache.r6g.2xlarge nodes
- 1.4 TB total capacity
- Only storing 382 GB

After

- Minimum: 10 nodes
- Maximum: 36 nodes

Estimated cost savings: ~$14k/month

aws

Wrapping up

- Modernize
- Simplify
- Standardize
- Reduce costs

Thank you!

Joe Travaglini
joetrav@amazon.com

Lindsey Berg
lberg@groupon.com

aws