Architecting for the Cloud – Best Practices

Approaches for architecting for AWS

Lift-and-shift

- **Deploy existing apps in AWS with minimal re-design**
- Good strategy if starting out on AWS, or if application can't be re-architected due to cost or resource constraints
- Primarily use core services such as EC2, EBS, VPC

Cloud-optimized

- **Evolve architecture for existing app to leverage AWS services**
- Gain cost and performance benefits from using AWS services such as RDS, SQS, and so on

Cloud-native architecture

- **Architect app to be cloud-native from the outset**
- Leverage the full AWS portfolio
- Truly gain all the benefits of AWS (security, scalability, cost, durability, low operational burden, etc)

So what does **architecting for the cloud** mean?

Scalability

A scalable architecture is critical to take advantage of a scalable infrastructure

Scalability

A scalable architecture is critical to take advantage of a scalable infrastructure

Characteristics of a scalable architecture

- Increase resources → Increase in performance
  - This leads to economy of scale, increasing cost-effectiveness with growth
- Operationally efficient
- Resilient

Cloud Architecture Best Practices

There are 7 best practices to remember…

Cloud Architecture Best Practices

1. Design for failure and nothing fails
2. Build security in every layer
3. Leverage different storage options
4. Implement elasticity
5. Think parallel
6. Loose coupling sets you free
7. Don’t fear constraints

Let’s cover these principles in context…

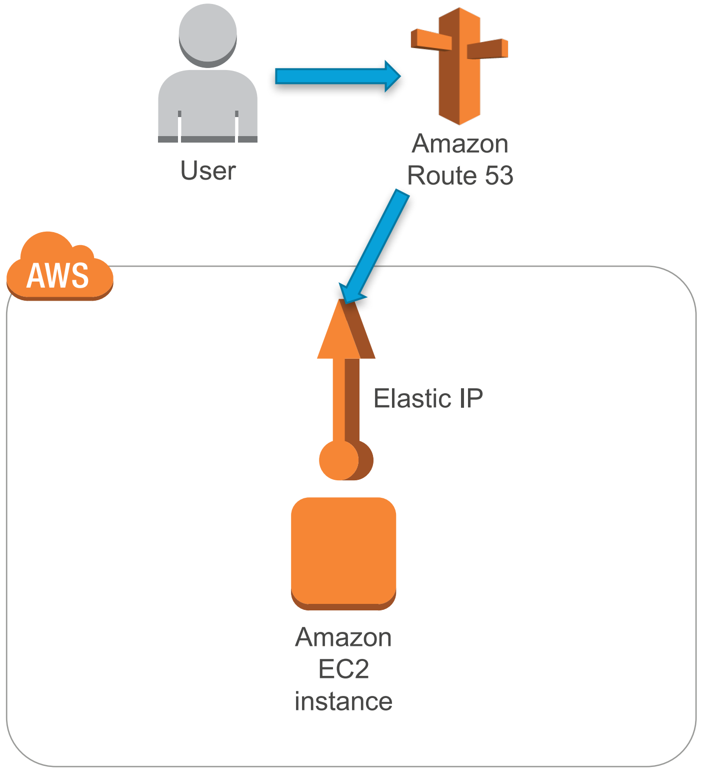
(We’ll show a web-app architecture in many of these examples, but the principles still apply for other cloud architectures as well.)

1

Design for Failure

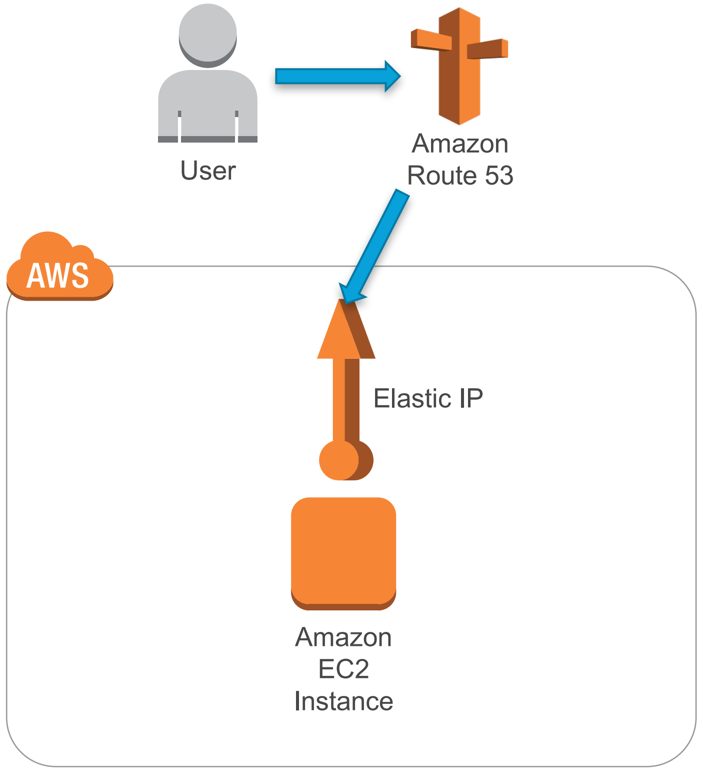
Let’s start from Day 1, with 1 User

- Amazon Route 53 for DNS
- A <u>single</u> Elastic IP
- A <u>single</u> Amazon EC2 instance
  - With full stack on this host
    - Web app
    - Database
    - Management
    - And so on…



Design for Failure

- We could potentially get to a few hundred to a few thousand depending on application complexity and traffic, but…
- **No failover**
- **No redundancy**
- **Too many eggs in one basket**



Design for Failure

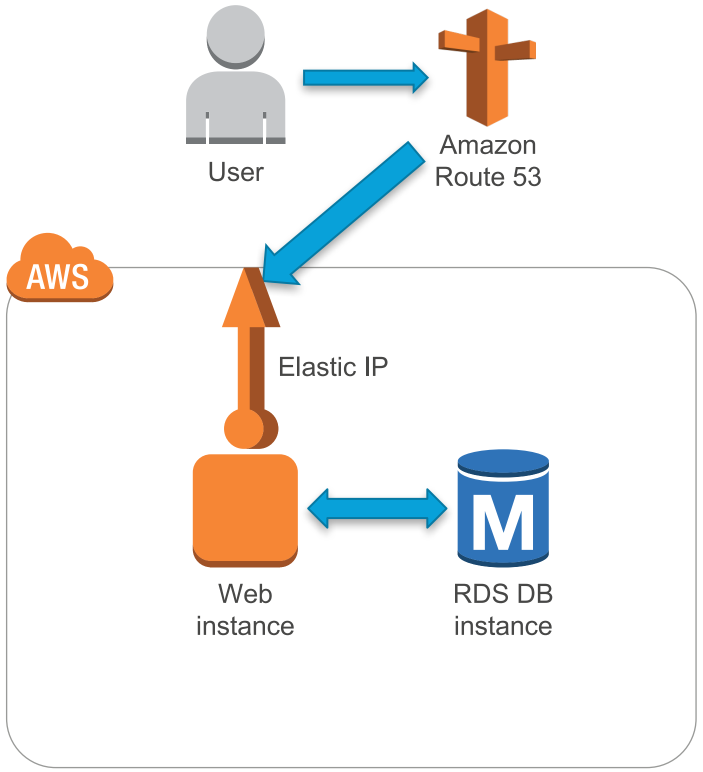
"Everything fails, all the time."

Werner Vogels, CTO Amazon.com

Design for Failure

First, let's separate out our single host into more than one:

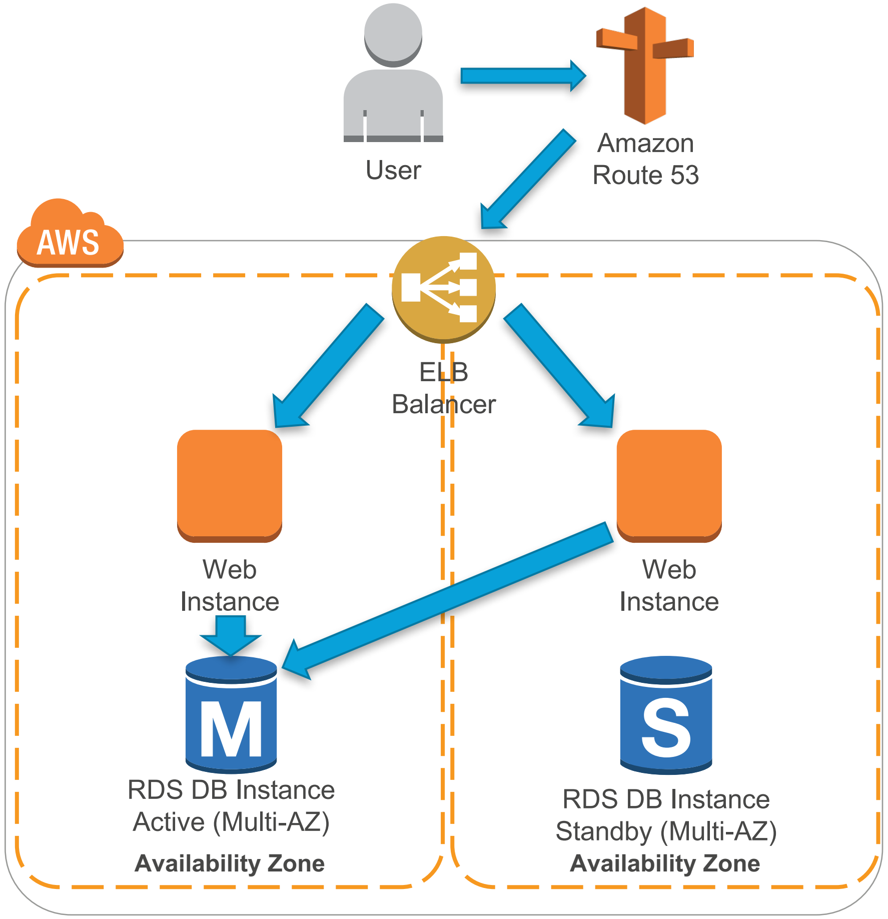
- Web
- Database
    - Use Amazon RDS to make your life easier



Design for Failure

Next, let's address our lack of failover and redundancy issues:

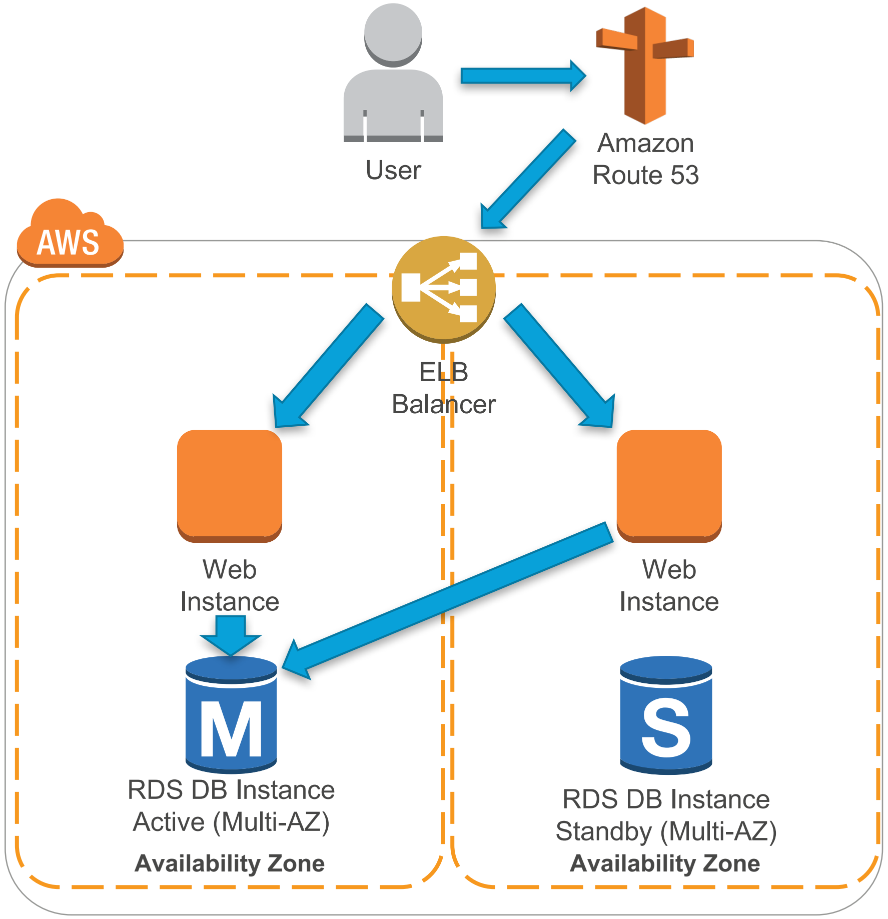
- Another web instance
  - In another Availability Zone
- RDS Multi-AZ
- Elastic Load Balancing (ELB)



Design for Failure

Best Practices

- Use multiple Availability Zones
- Use Elastic Load Balancing
- Use Elastic IP addresses
- Do real-time monitoring with CloudWatch
- Use Simple Notification Service (SNS) for real-time alarms based on CloudWatch metrics
- Create database slaves across Availability Zones



Design for Failure

- Avoid single points of failure

Design for Failure

- 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/replaced
  - When, not if, an individual component fails, the application does not fail
  - Leverage Route 53 Pilot-light or Warm-standby strategies to implement DR
  - Self healing Auto Scaling groups can be used to protect against AZ level outages.

②

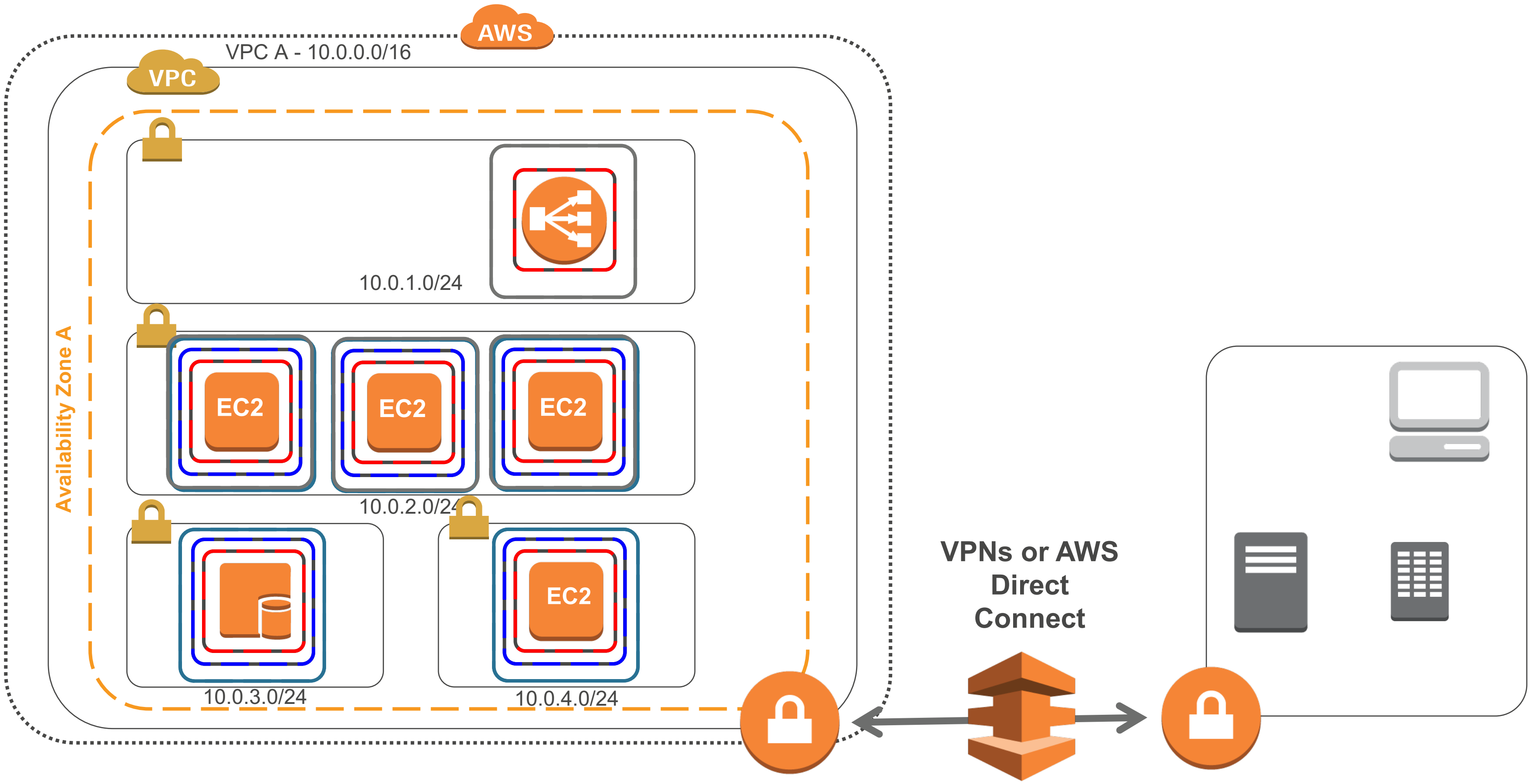
Build Security in Every Layer

Build Security in Every Layer

You have the control to implement in important layers:

- Encrypt data in transit and at rest
  - Use Key Management Service to create and control encryption keys used to encrypt your data
- Enforce principle of least privilege with Identity and Access Management (IAM) users, groups, roles, and policies
- Create distinct, restricted Security Groups for each application role
  - Restrict external access via these Security Groups
- Create Network ACLs to restrict traffic at subnet level
- Use Multi-Factor Authentication

Build Security in Every Layer



③

Leverage Many Storage Options

Leverage Many Storage Options

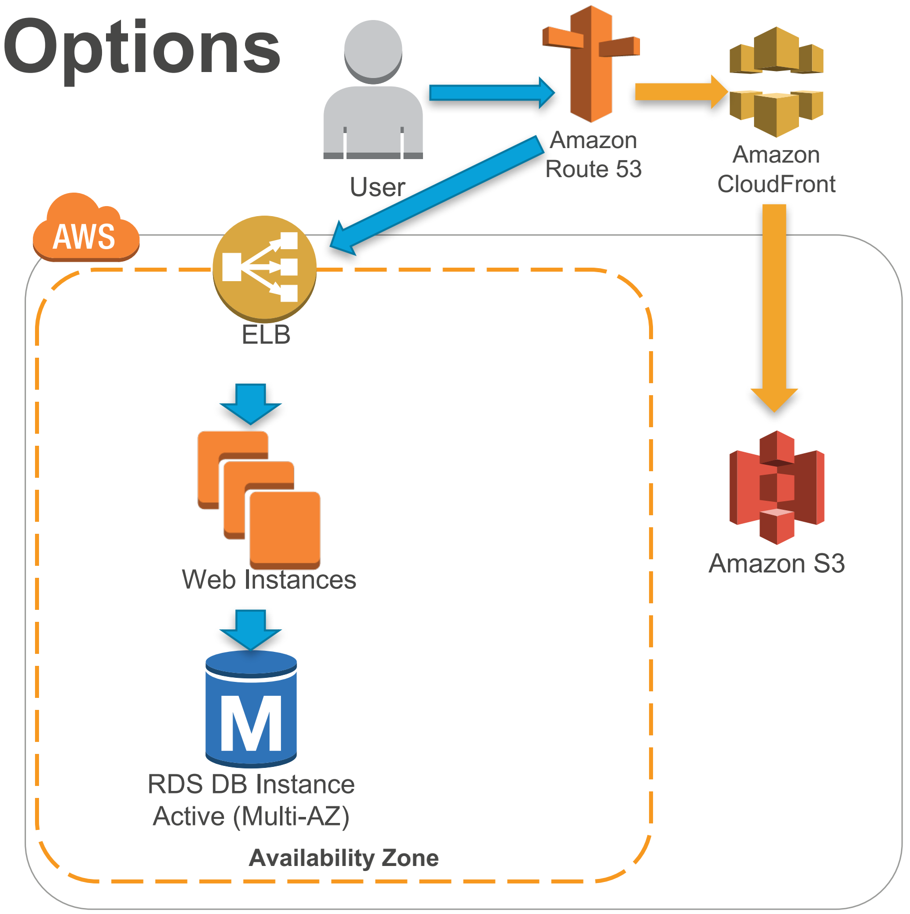
One size does NOT fit all

- **Amazon S3** – large objects
- **Amazon Glacier** – archive data
- **Amazon CloudFront** – content distribution
- **Amazon DynamoDB** – simple non-relational data
- **Amazon EC2 Ephemeral Storage** – transient data
- **Amazon EBS** – persistent block storage with snapshots
- **Amazon RDS –** Automated, managed MySQL
- **Amazon Redshift –** Data warehouse workloads

Leverage Many Storage Options

We can shift some load around…

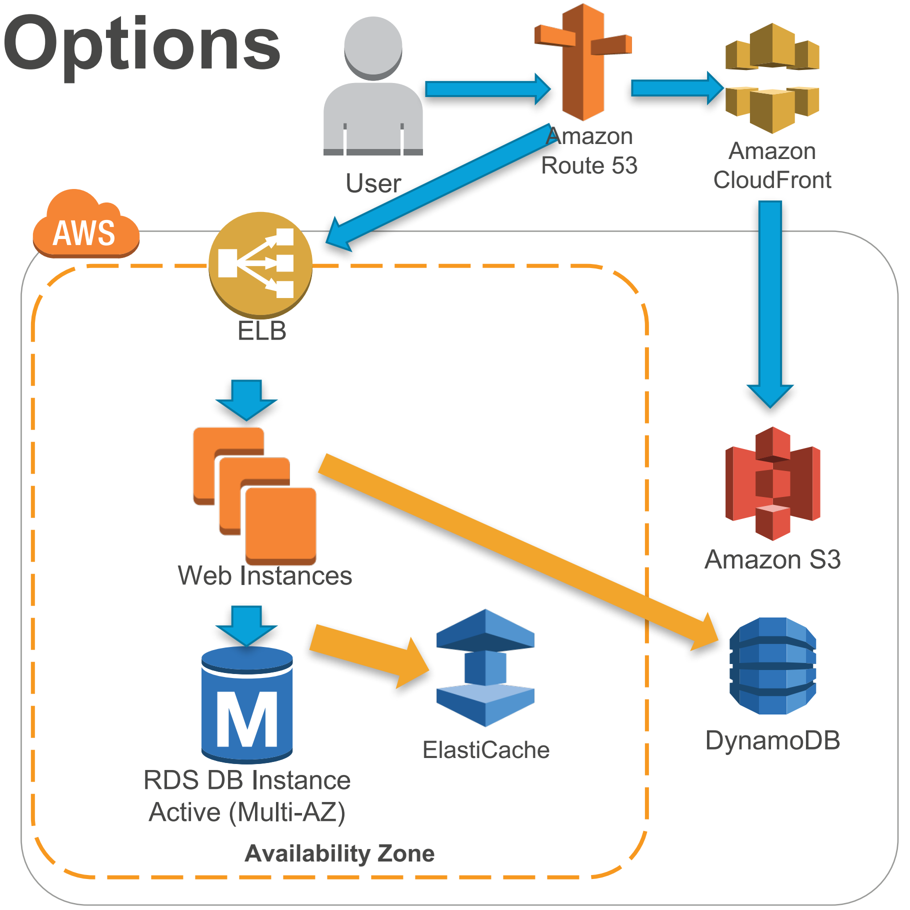
- static content to Amazon S3 and Amazon CloudFront



Leverage Many Storage Options

We can shift some load around…

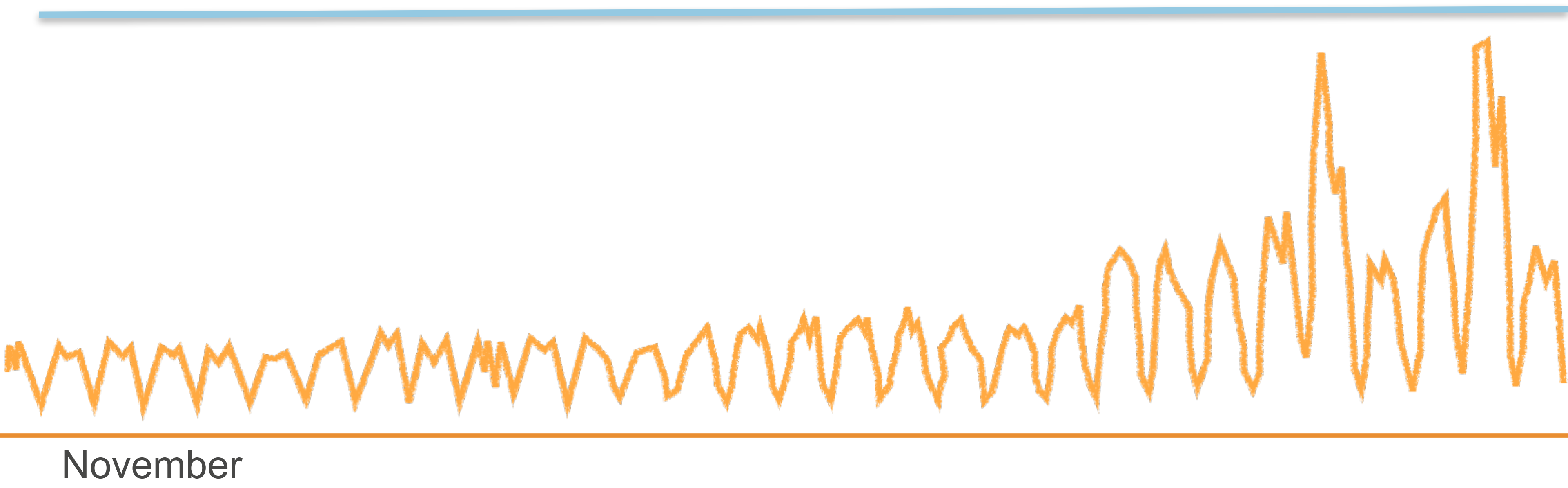
- static content to Amazon S3 and Amazon CloudFront
- session/state to Amazon DynamoDB
- DB caching to Amazon ElastiCache



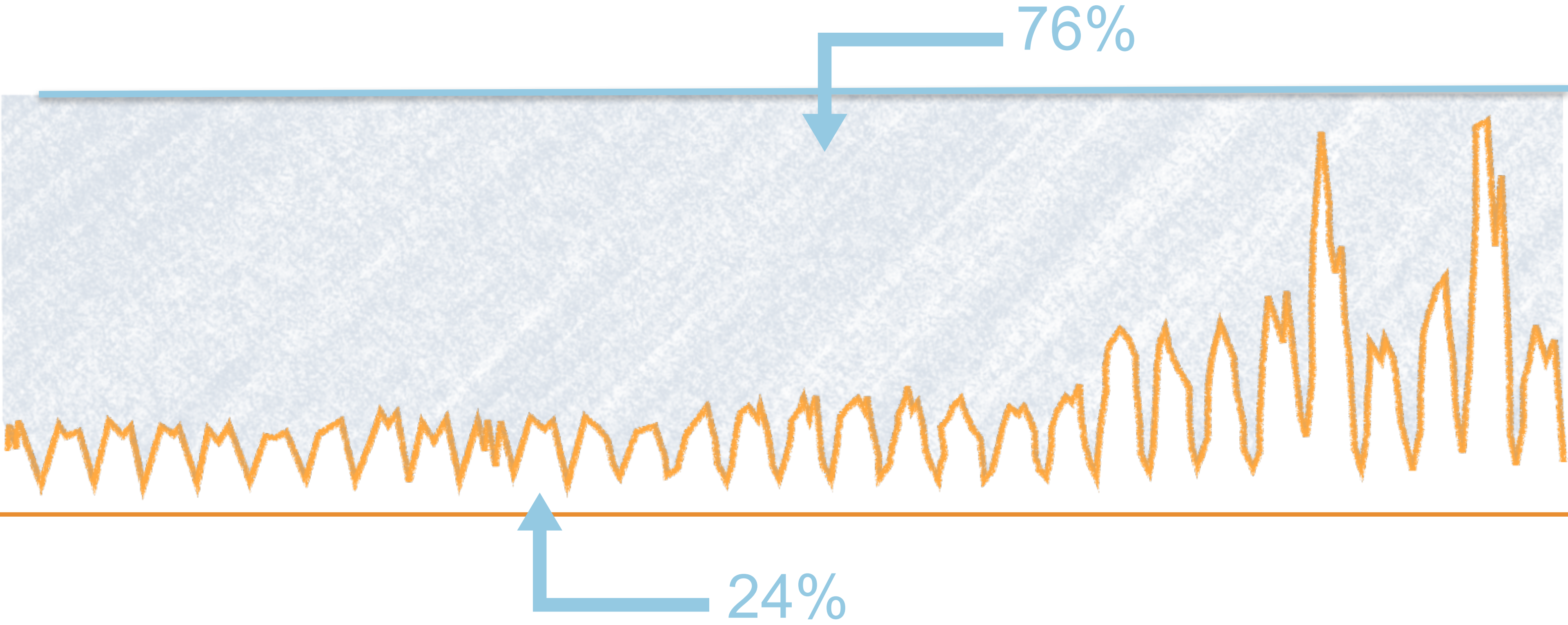
④

Implement Elasticity

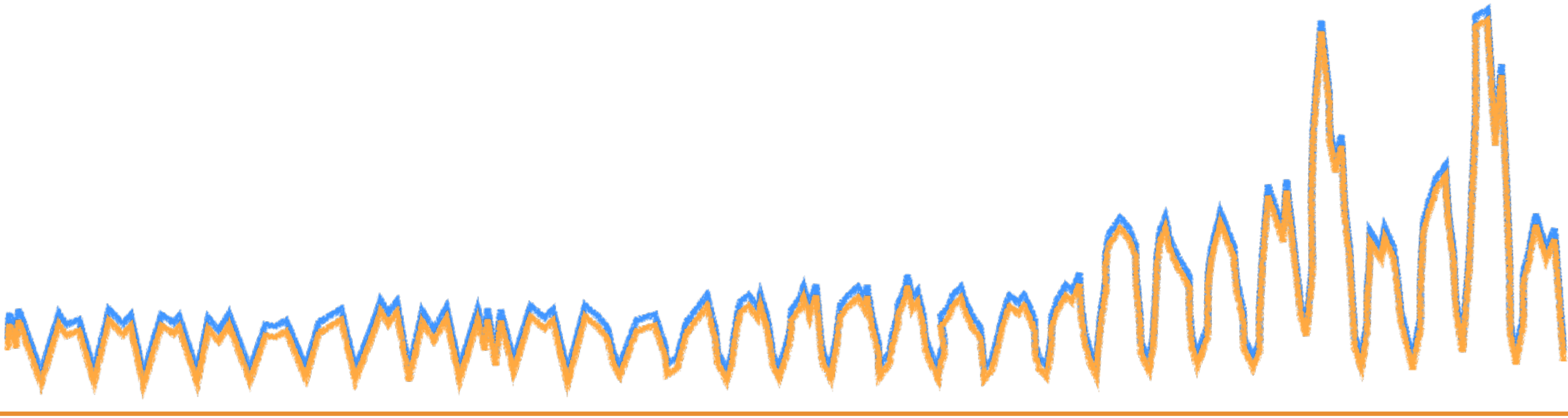
November traffic to Amazon.com



November traffic to Amazon.com



November traffic to Amazon.com



Implement Elasticity

Using AWS to implement an elastic architecture

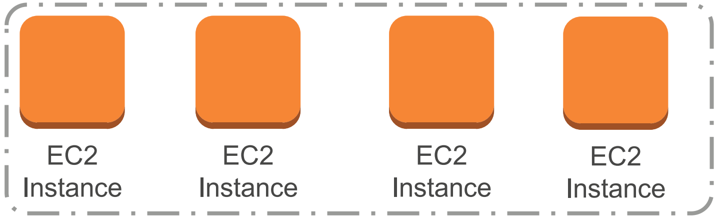
- Auto Scaling
- Elastic Load Balancing
- Amazon DynamoDB or Amazon S3 for dynamic configuration

Auto Scaling

Automatic resizing of compute clusters

Define min/max pool sizes

CloudWatch metrics drive scaling



Implement Elasticity

To implement elasticity:

- Don’t assume the health, availability, or fixed location of components

Implement Elasticity

To implement elasticity:

- Don’t assume the health, availability, or fixed location of components
- Use designs that are resilient to reboot and relaunch

Implement Elasticity

To implement elasticity:

- Don’t assume the health, availability, or fixed location of components
- Use designs that are resilient to reboot and relaunch
- Bootstrap your instances
  - When an instance launches, it should ask “*Who am I and what is my role?*”

Implement Elasticity

To implement elasticity:

- Don’t assume the health, availability, or fixed location of components
- Use designs that are resilient to reboot and relaunch
- Bootstrap your instances
  - When an instance launches, it should ask “*Who am I and what is my role?*”
- Favor dynamic configuration

5

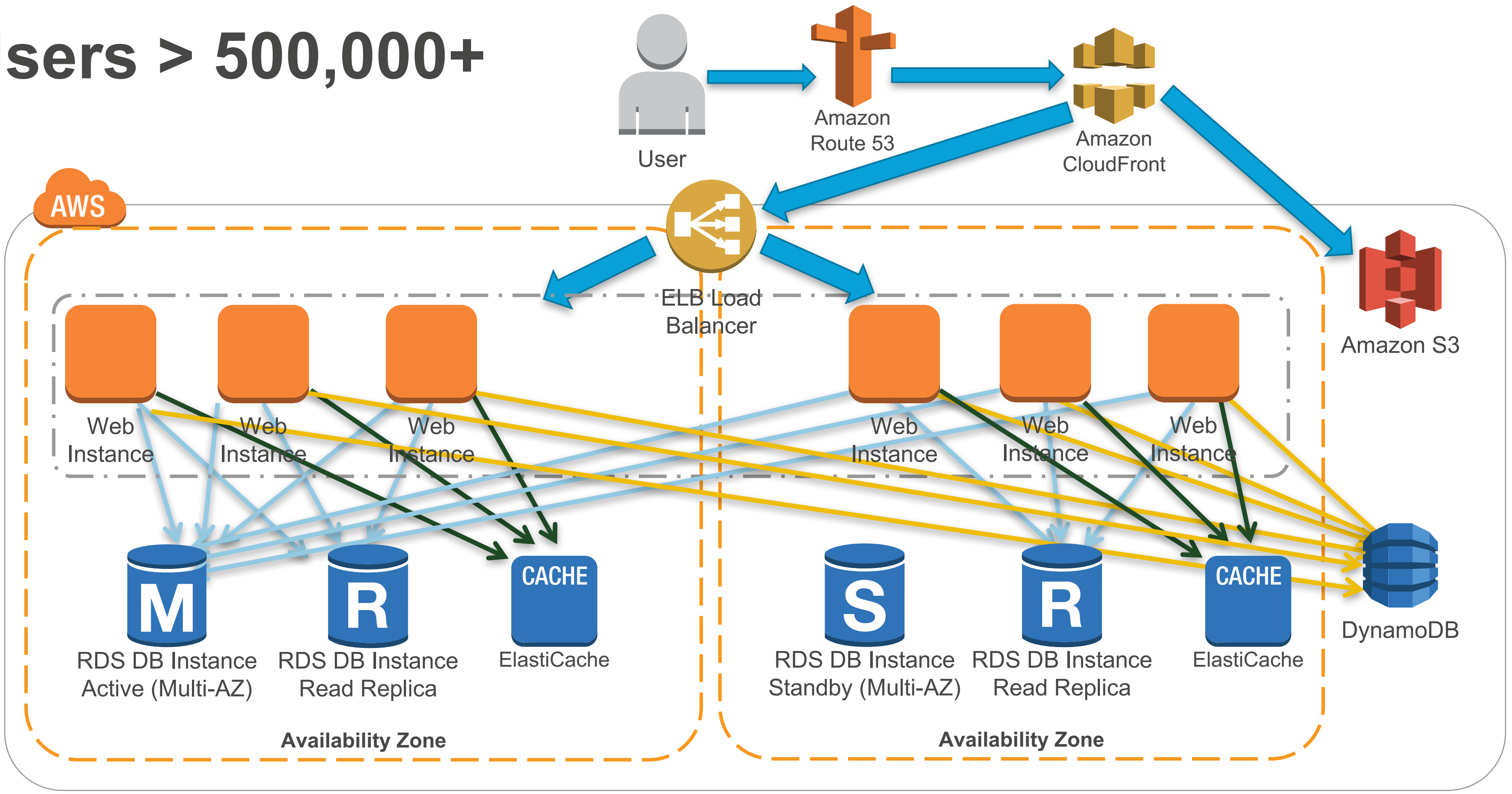
Think Parallel

Think Parallel

Serial and sequential processes are history

- Experiment with different parallel architectures
- Use multi-threading and concurrent requests to cloud services
- Run parallel **MapReduce** jobs
- Use **Elastic Load Balancing** to distribute load
- Using **Kinesis** you can have multiple applications process a stream of data concurrently
- **Lambda** lets you run thousands of functions run in parallel and performance remains consistently high regardless of the frequency of events.
- Amazon **S3** multi-part upload and ranged Gets

Users > 500,000+



6

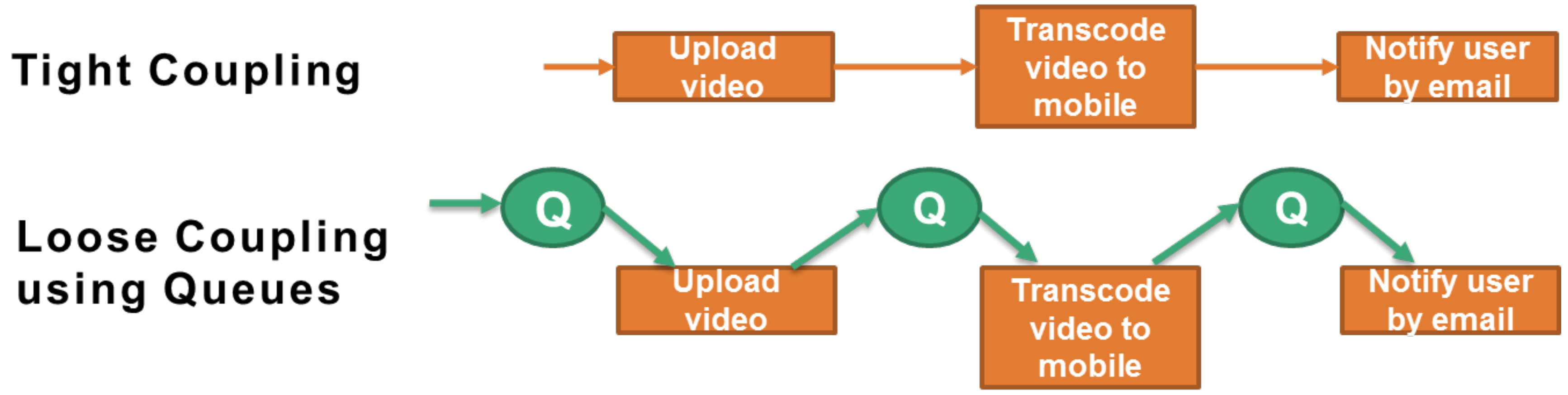
Loose Coupling Sets You Free

Loose Coupling Sets You Free

- Design architectures with independent components
  - The looser they're coupled, the larger they scale
- Design every component as a black box
- Load balanced clusters

Loose Coupling Sets You Free

Use Amazon Simple Queue Service (SQS) to pass messages between loosely coupled components



Loose Coupling Sets You Free

Don’t reinvent the wheel – leverage AWS purpose-built services wherever possible

- Email
- Queuing
- Transcoding
- Search
- Databases
- Monitoring
- Metrics
- Logging
- Compute

AWS Lambda

Amazon SNS

Amazon CloudSearch

Amazon SQS

Amazon SES

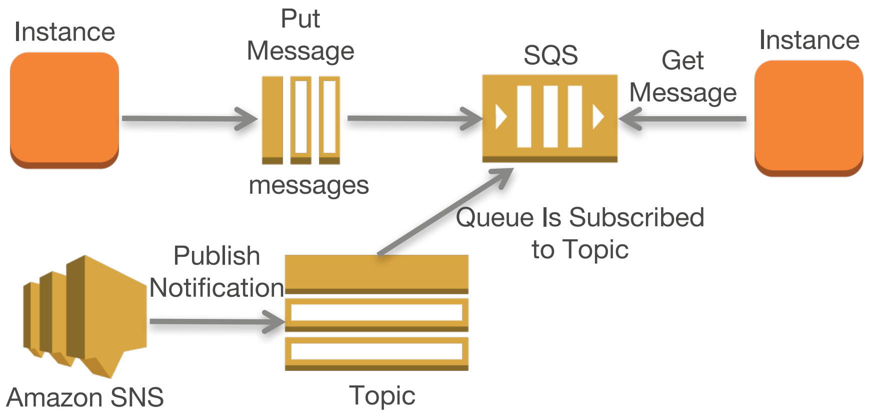
Amazon SWF

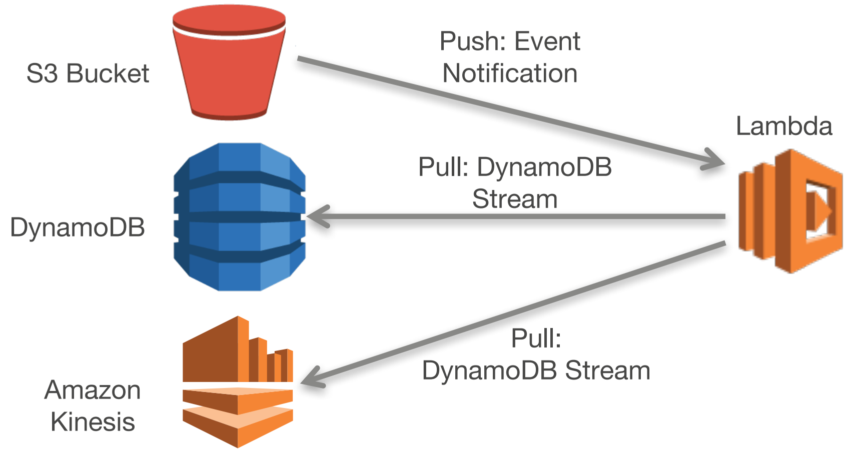
Amazon Elastic Transcoder

Loose Coupling Sets You Free

The looser they're coupled, the bigger they scale

- Independent components
- Design everything as a black box
- Decouple interactions
- Favor services with built-in redundancy and scalability rather than building your own





7

Don’t Fear Constraints

Don’t Fear Constraints

Rethink traditional architectural constraints

- Need more RAM?
  - Consider distributing load across machines or a shared cache

Don’t Fear Constraints

Rethink traditional architectural constraints

- Need more RAM?
  - Consider distributing load across machines or a shared cache
- Need better IOPS for database?
  - Consider multiple read replicas, sharding, or DB clustering
  - PIOPS, SSD-backed instance storage
  - Database caching with ElastiCache

Don’t Fear Constraints

Rethink traditional architectural constraints

- Need more RAM?
  - Consider distributing load across machines or a shared cache
- Need better IOPS for database?
  - Consider multiple read replicas, sharding, or DB clustering
  - PIOPS, SSD-backed instance storage
  - Database caching with ElastiCache
- Hardware failed or config got corrupted
  - “Rip and replace” – Simply toss bad instance and instantiate replacement

Don’t Fear Constraints

Rethink traditional architectural constraints

- Need more RAM?
  - Consider distributing load across machines or a shared cache
- Need better IOPS for database?
  - Consider multiple read replicas, sharding, or DB clustering
  - PIOPS, SSD-backed instance storage
  - Database caching with ElastiCache
- Hardware failed or config got corrupted
  - “Rip and replace” – Simply toss bad instance and instantiate replacement
- Cost Effective Disaster Recovery (DR) strategy
  - Consider using Route-53 for failover to Pilot light or warm standby DR stacks

Don’t Fear Constraints

Rethink traditional architectural constraints

- One size does not fit all
  - Scale Instance size and type as necessary with minimal or no downtime

Don’t Fear Constraints

Rethink traditional architectural constraints

- One size does not fit all
  - Scale Instance size and type as necessary with minimal or no downtime
- Create inexpensive HA using Elastic Network Interface (ENI)
  - Swap network interface between EC2 instances for rapid service recovery without making any DNS or routing changes

Cloud Architecture Quiz

What are the 7 architectural principles?

1. Design for f and nothing fails
2. Build s in every layer
3. Leverage different s options
4. Implement e
5. Think p
6. Loose c sets you free
7. Don't fear c

Cloud Architecture Quiz

What are the 7 architectural principles?

1. Design for failure and nothing fails
2. Build security in every layer
3. Leverage different storage options
4. Implement elasticity
5. Think parallel
6. Loose coupling sets you free
7. Don’t fear constraints

Any Questions?