Architecting for the Cloud– Best Practices

Architecting Approaches 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 or VMWare on AWS

Cloud-optimized

- **Evolve architecture for existing app to leverage AWS services**
- Gain cost and performance benefits from using AWS services such as Auto Scaling Groups, 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)

aws

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

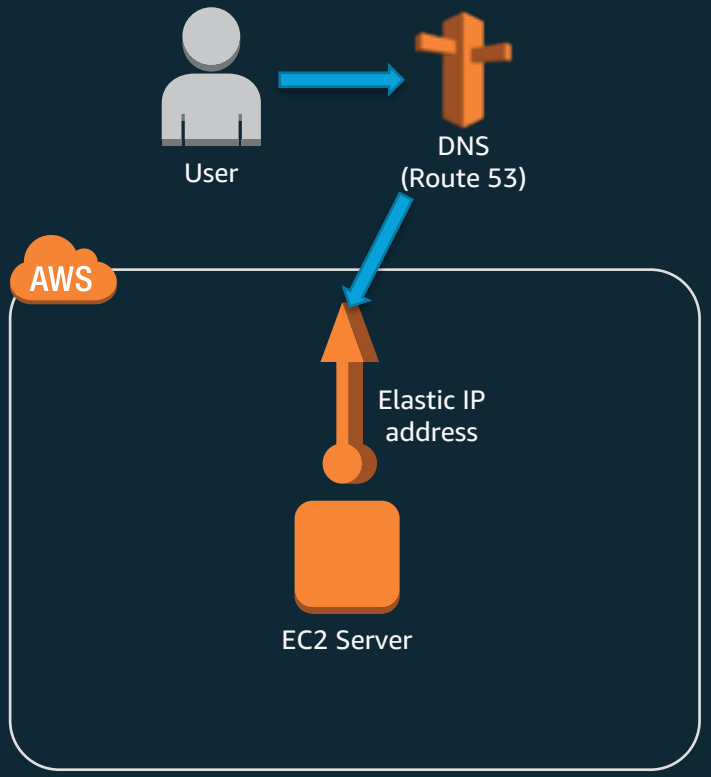
aws

Design for Failure and Nothing Fails

Design for Failure: A Single User

Single Points of Failure:

- A single Elastic IP
  - Gives a server a static Public IP address
- A single Amazon Elastic Compute Cloud (EC2) instance
  - Full stack on single host
    - Web application
    - Database
    - Management, etc…

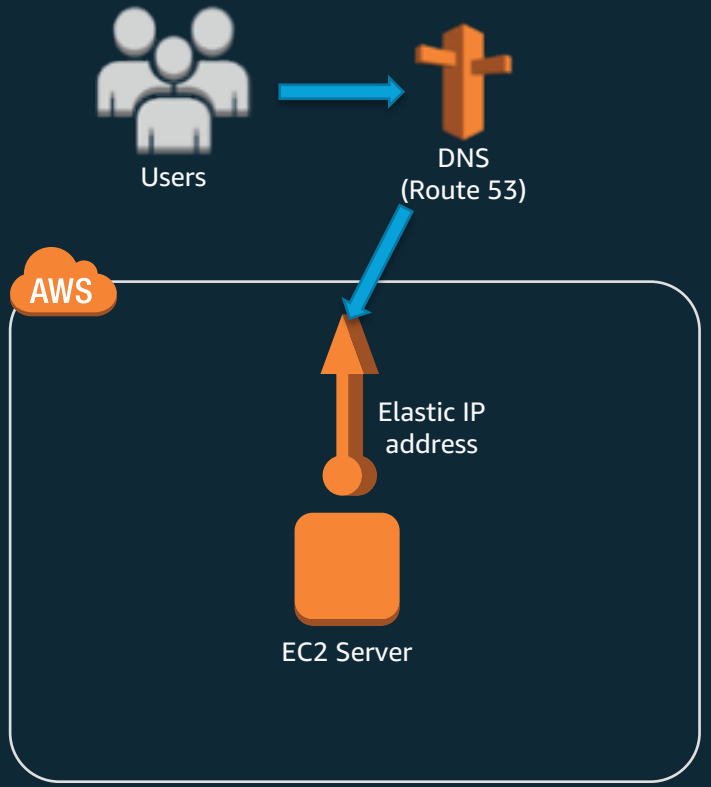


Design for Failure: Difficulties Scaling to Many Users

We could potentially get to a few hundred to a few thousand users depending on application complexity and traffic, but…

There may be difficulty scaling to many more users due to:

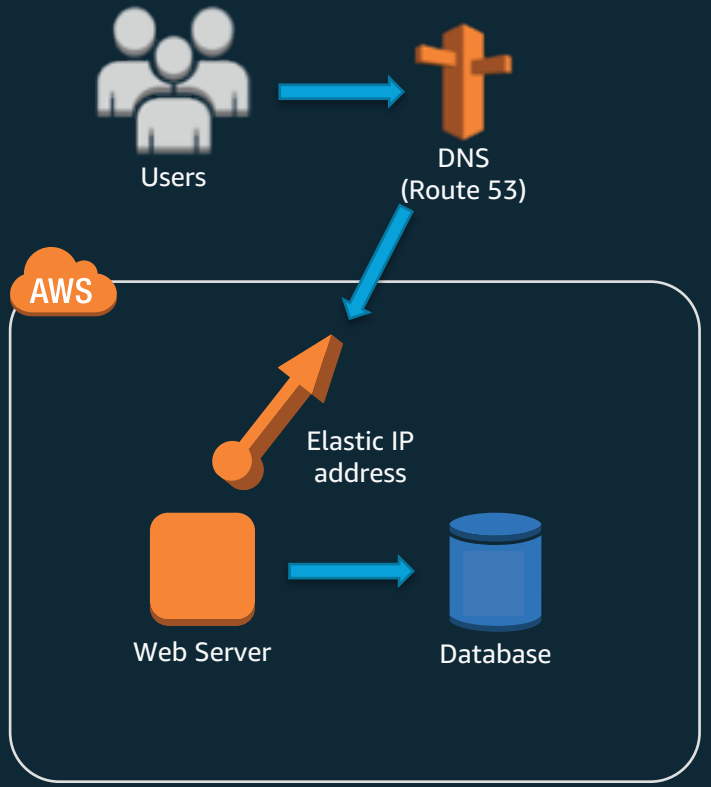
- **All eggs in one basket**
- **No failover or redundancy**



Design for Failure: Solving “All Eggs in One Basket”

Separate single EC2 Server into web and database tiers:

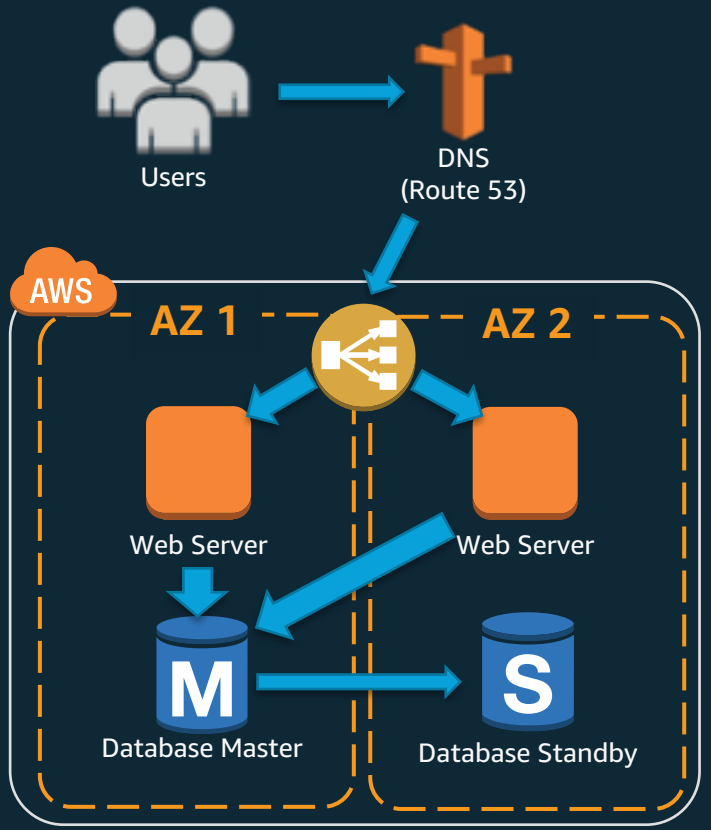
- Web Server on EC2
- Database on EC2 or RDS
  - Amazon Relational Database Service (RDS) can take care of management overhead such as patching, backups, and failure detection



Design for Failure: Solving No Failover/Redundancy

Leverage multiple Availability Zones for redundancy and high availability.

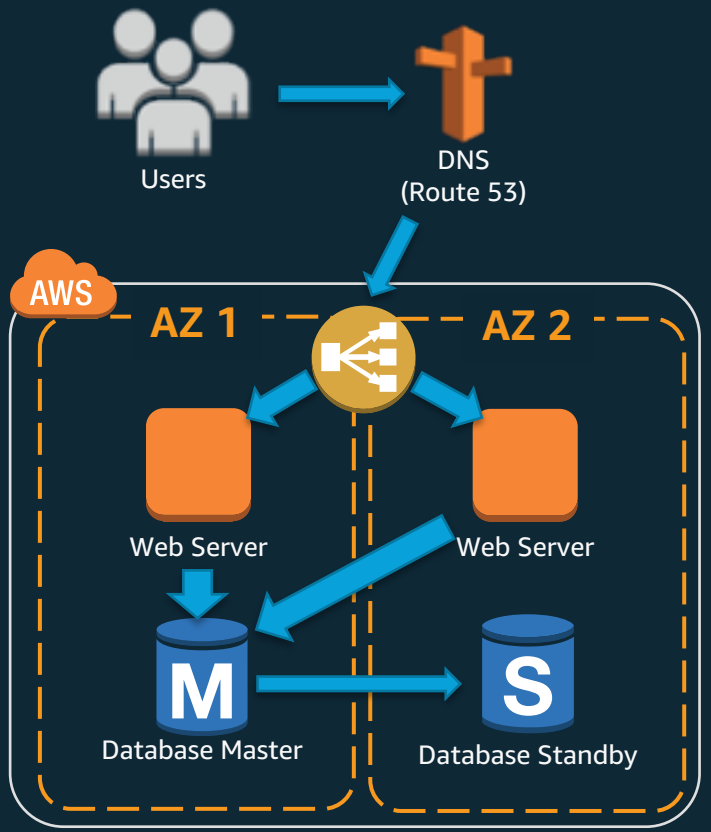
- Use an Elastic Load Balancer (ELB) across AZs for availability and failover
- If using RDS, use the Multi-AZ feature for managed replication and a standby instance
    - If not, use failover and replication features native to your database engine



Design for Failure: Best Practices

Best Practices:

- Eliminate single points of failure
- Use multiple Availability Zones
- Use Elastic Load Balancing
- Do real-time monitoring with CloudWatch
- Create a database standby across Availability Zones



Design for Failure

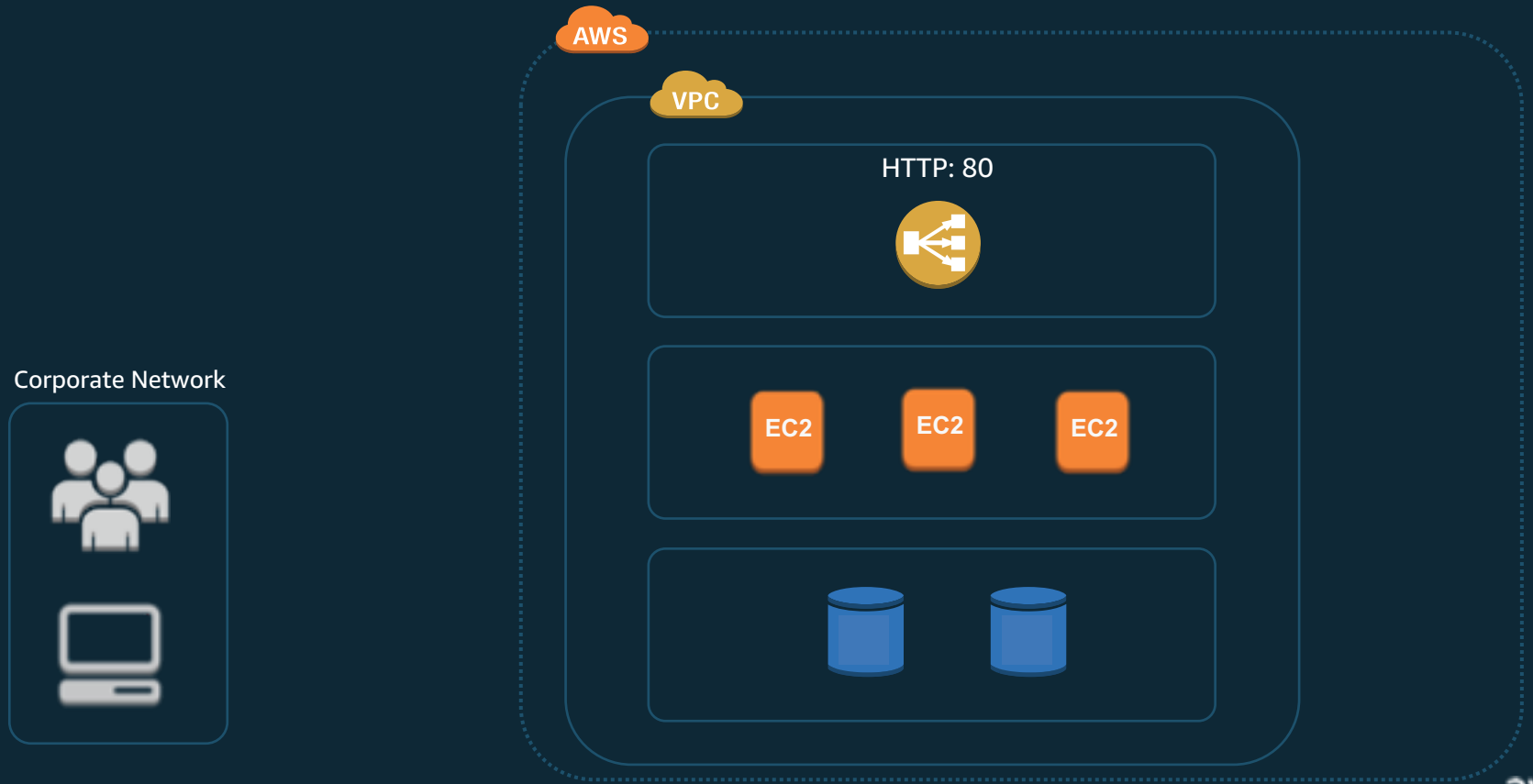
Avoid single points of failure

Assume everything fails and design backwards

- When, not if, an individual component fails, the application does not fail
  - Think of your servers as cattle, not pets
- Leverage Route 53 DNS Pilot-light or Warm-standby strategies to implement Disaster Recovery
- Auto Scaling groups can be used to detect failures and self-heal, thus protecting against AZ level outages

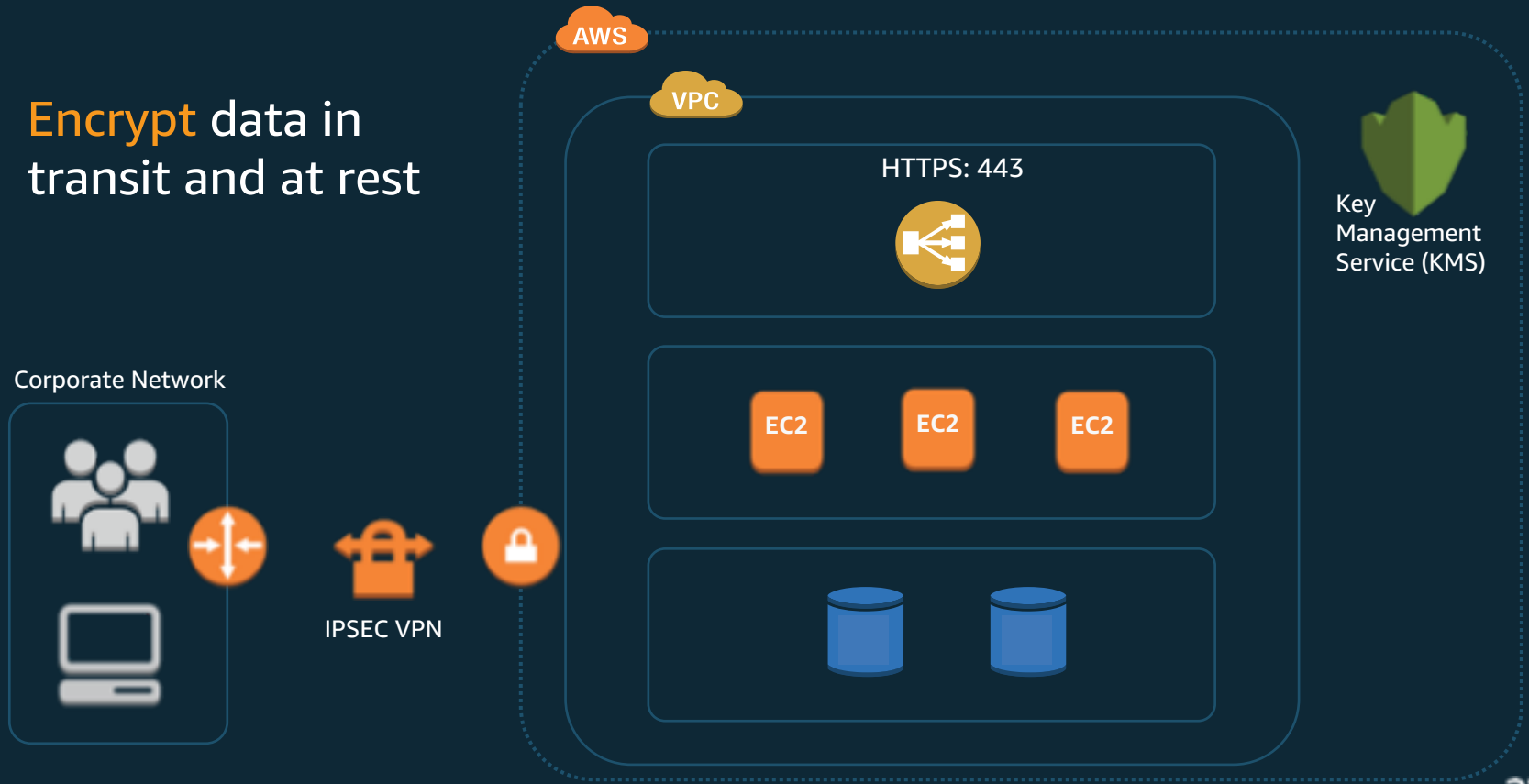
Build Security in Every Layer

Build Security in Every Layer



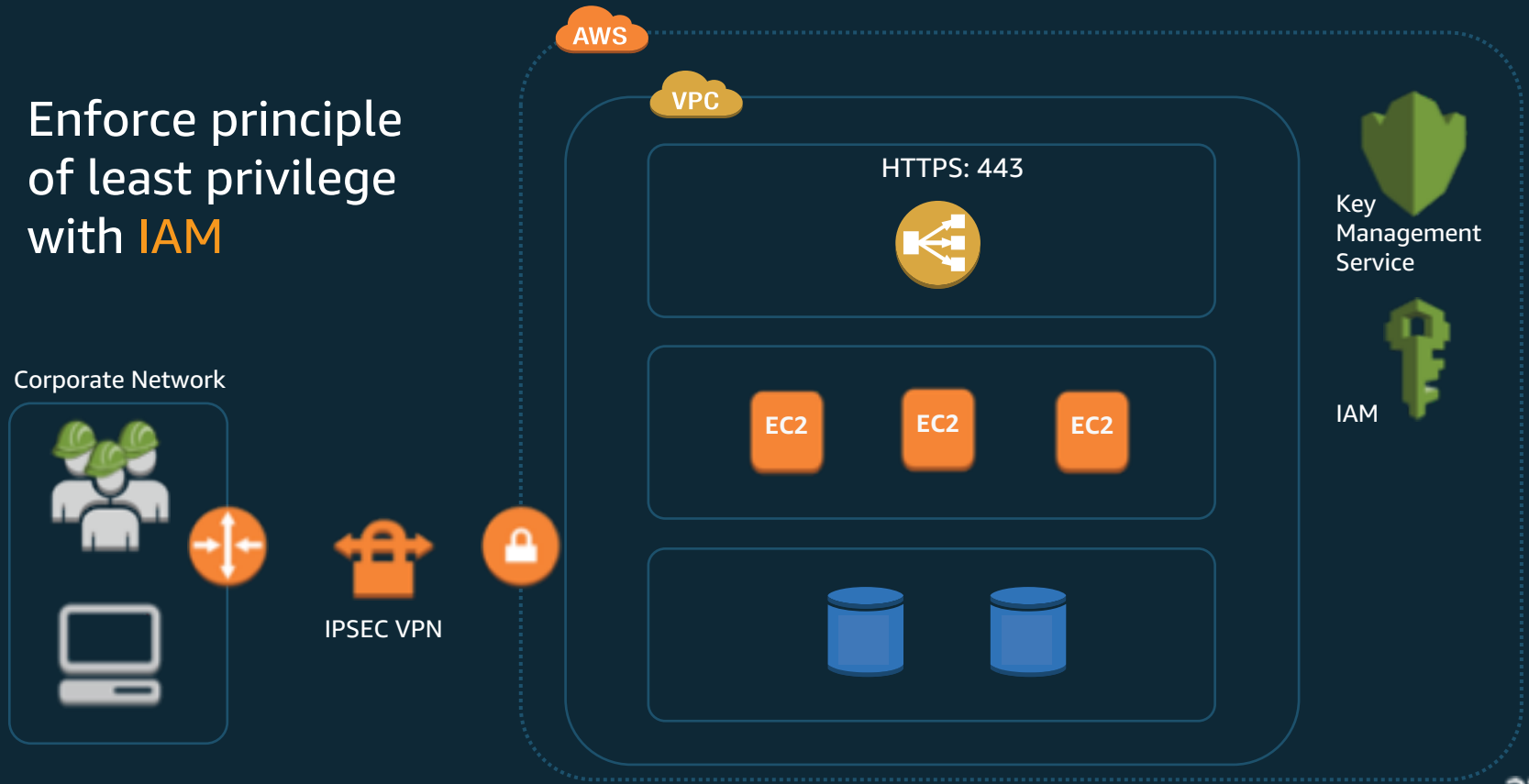
Build Security in Every Layer

Encrypt data in transit and at rest



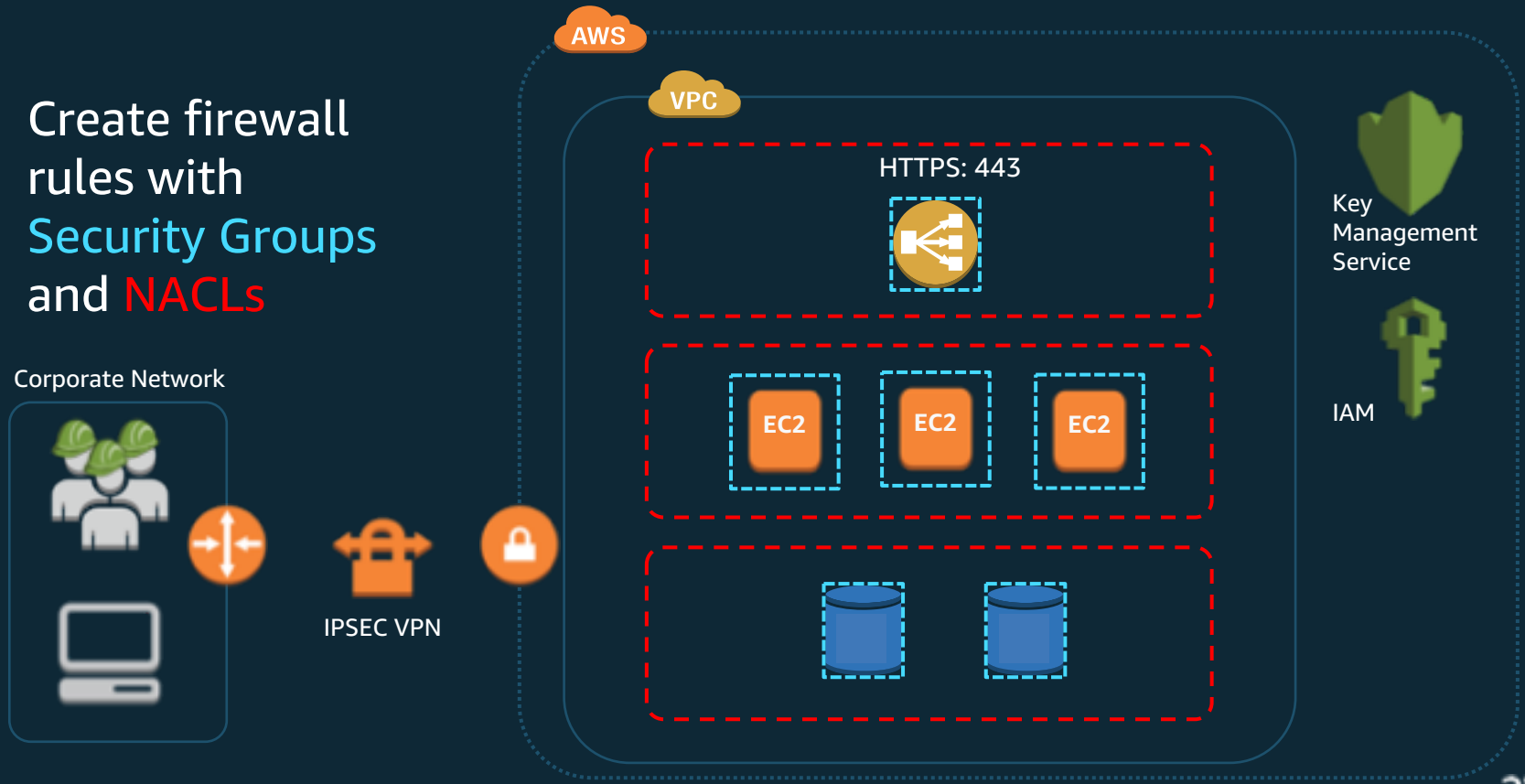
Build Security in Every Layer

Enforce principle of least privilege with IAM



Build Security in Every Layer

Create firewall rules with Security Groups and NACLs



Build Security in Every Layer

More Tools for your Security Toolbox:

- Amazon Inspector
- Amazon Certificate Manager
- AWS Shield
- AWS Web Application Firewall (WAF)
- Amazon Macie
- Amazon GuardDuty
- AWS Config

Leverage Many Storage Options

Leverage Many Storage Options

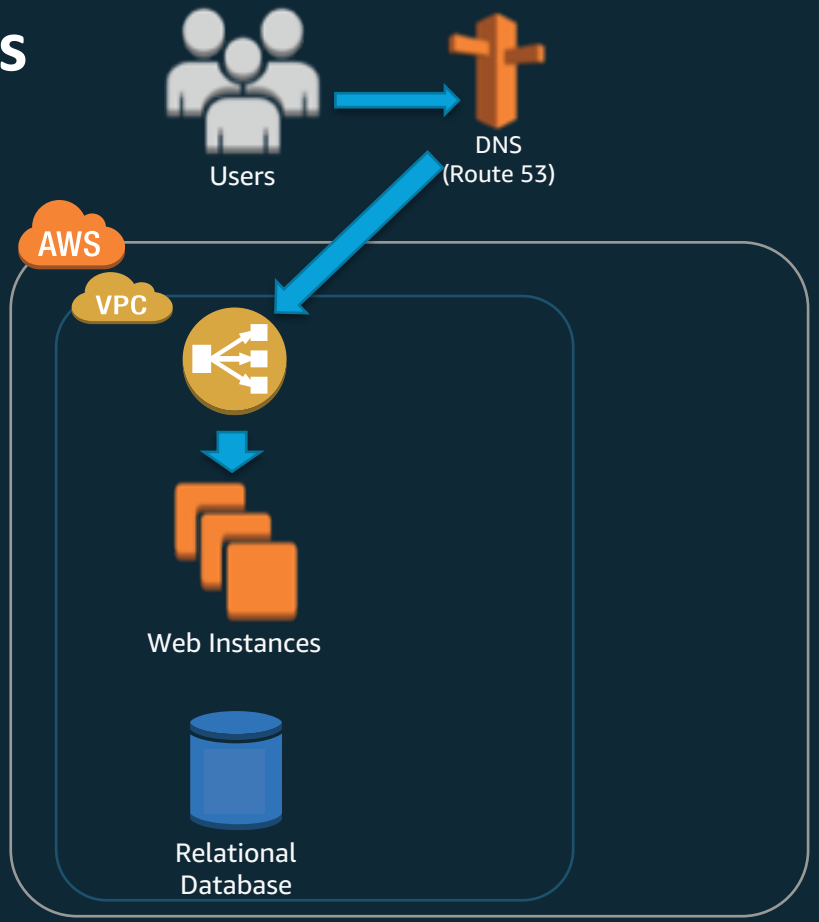
One size does NOT fit all

- **Amazon Elastic Block Storage (EBS)** – persistent block storage
- **Amazon EC2 Instance Storage** – ephemeral block storage
- **Amazon RDS** – managed relational database
- **Amazon CloudFront** – content distribution network
- **Amazon S3** – object/blob store, good for large objects
- **Amazon DynamoDB** – non-relational data (key-value)
- **Amazon ElastiCache** – managed Redis or Memcached

Leverage Many Storage Options

Current State:

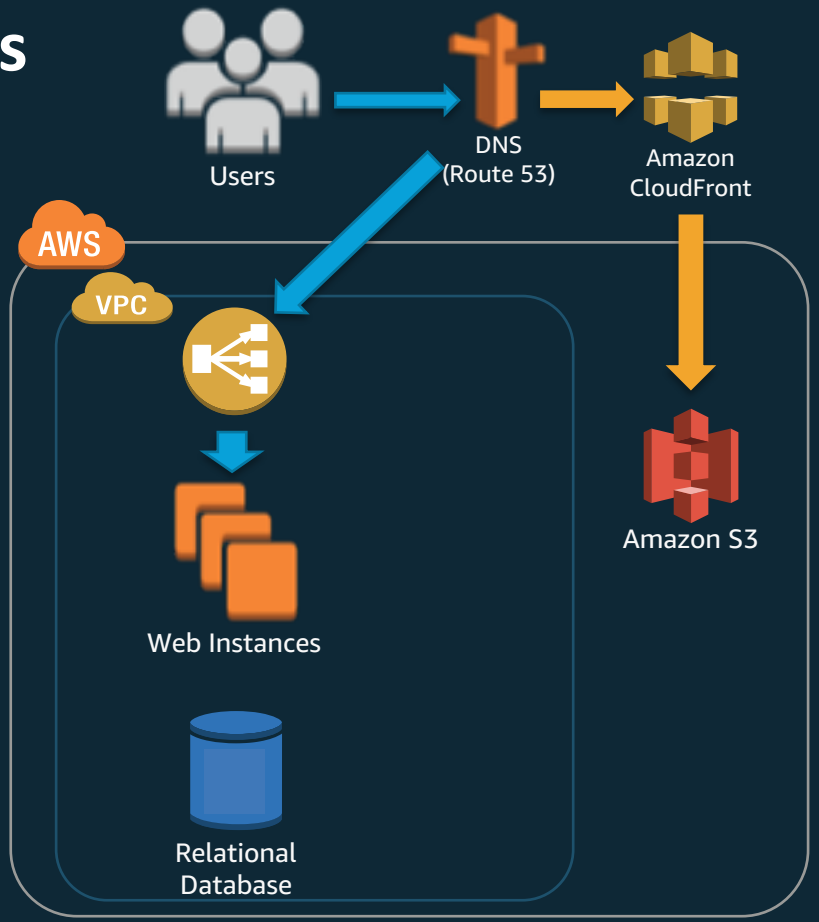
- All load handled by one stack
  - Elastic Load Balancer (ELB)
  - EC2 Web App cluster
  - Relational Database
- No caching layer(s)
- All persistent data in database or Web instances' Elastic Block Storage (EBS) volumes



Leverage Many Storage Options

Offload and cache requests for static assets:

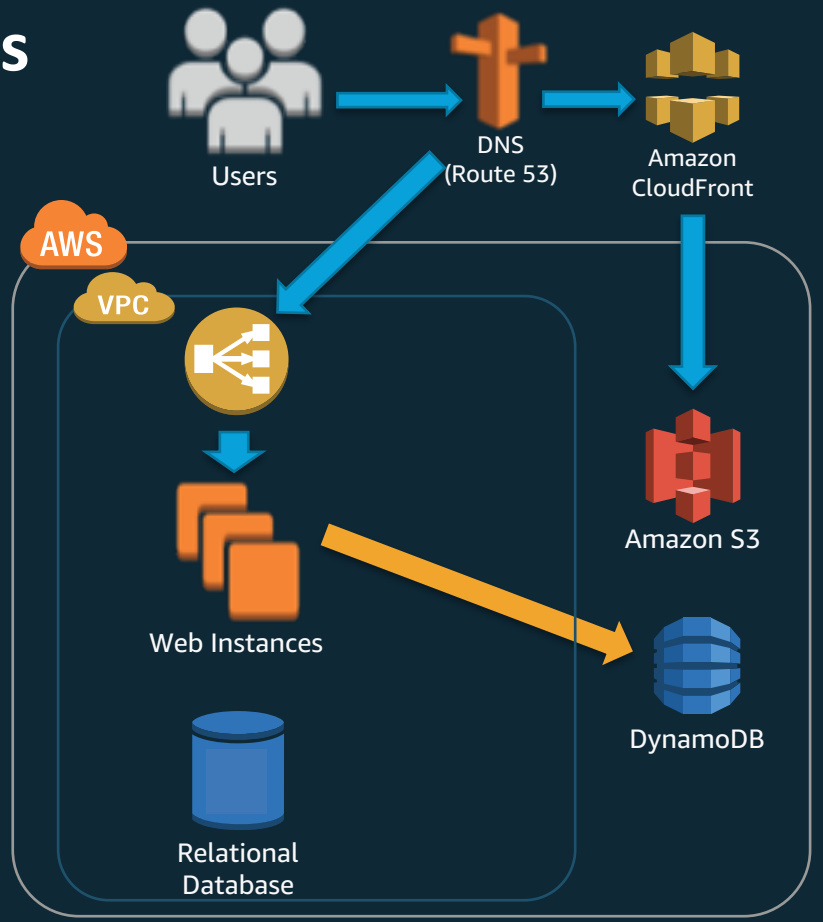
- Store large/static objects in Simple Storage Service (S3)
- Use a Content Delivery Network (CDN) like CloudFront to cache responses using points of presence all around the world



Leverage Many Storage Options

Save user session data in a database to avoid interrupting the user experience if a web host becomes unresponsive:

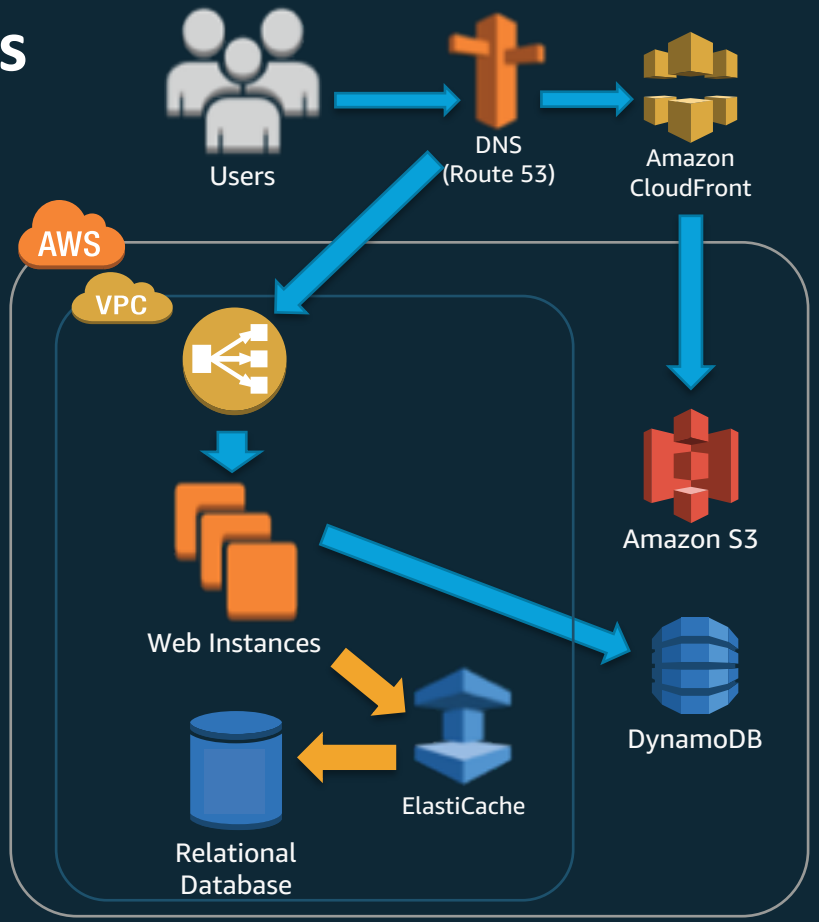
- Store session/state data in DynamoDB, a managed NoSQL key-value store



Leverage Many Storage Options

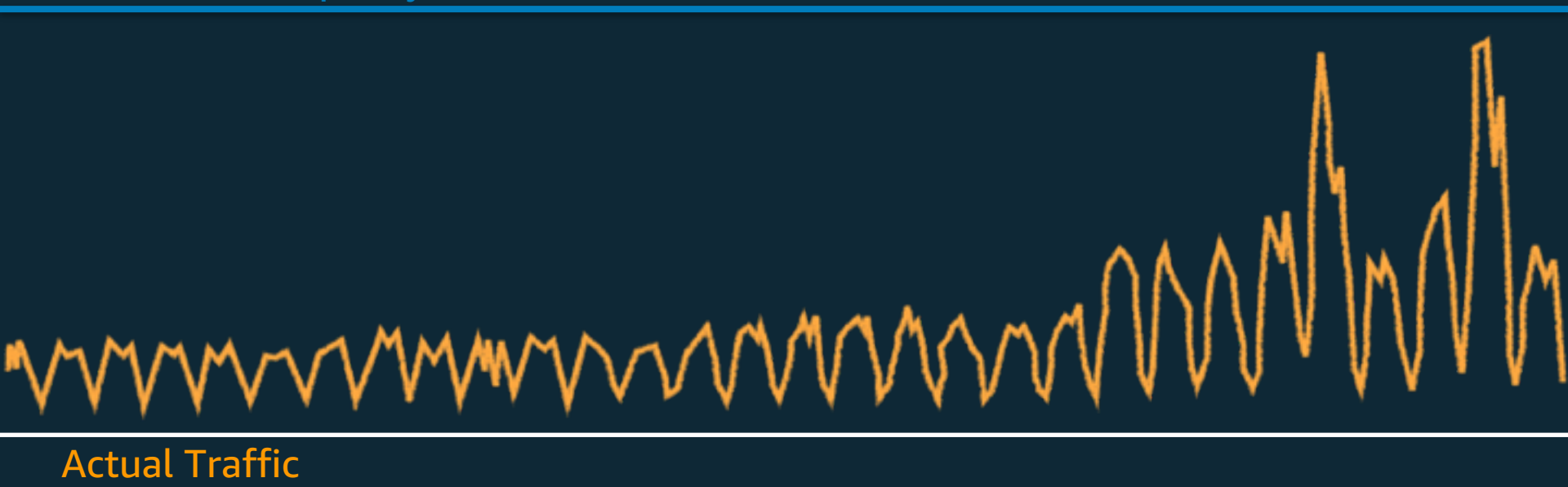
Cache frequent queries to shift the load off of your database:

- Put ElastiCache as a caching layer between the web hosts and the database

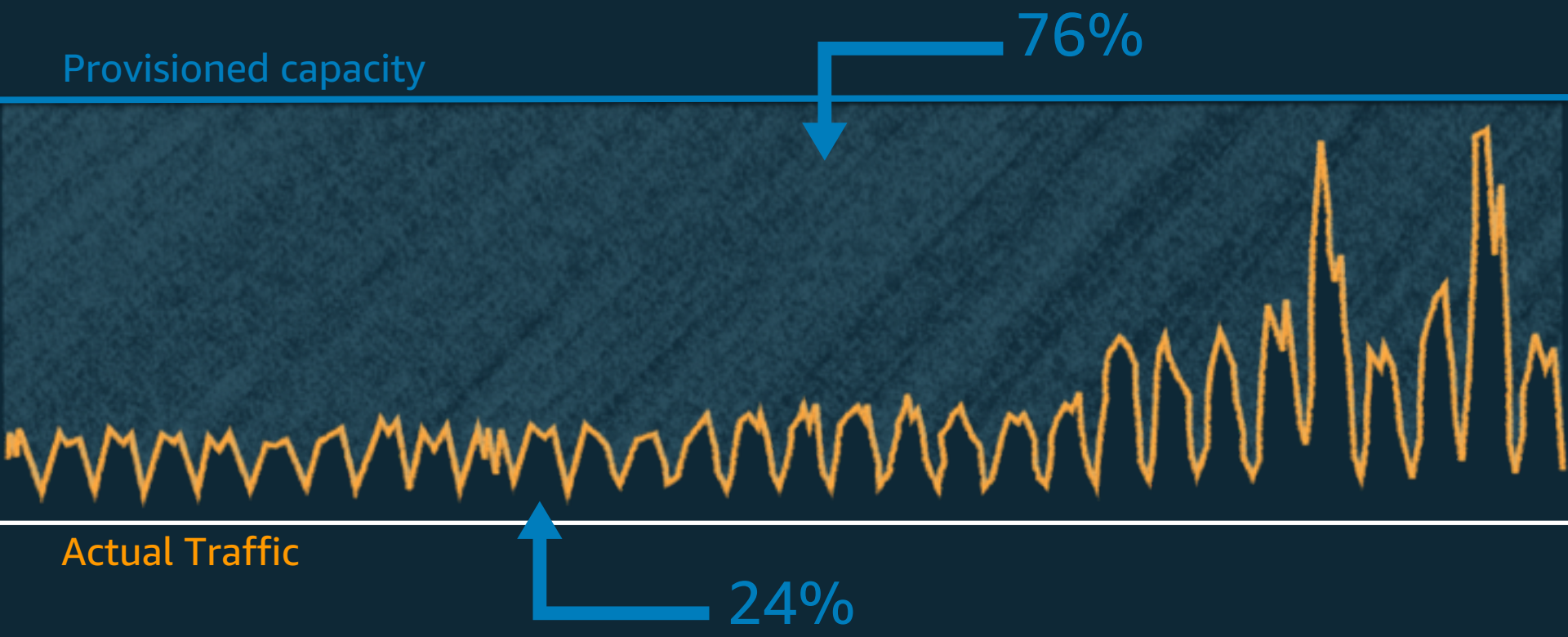


Implement Elasticity

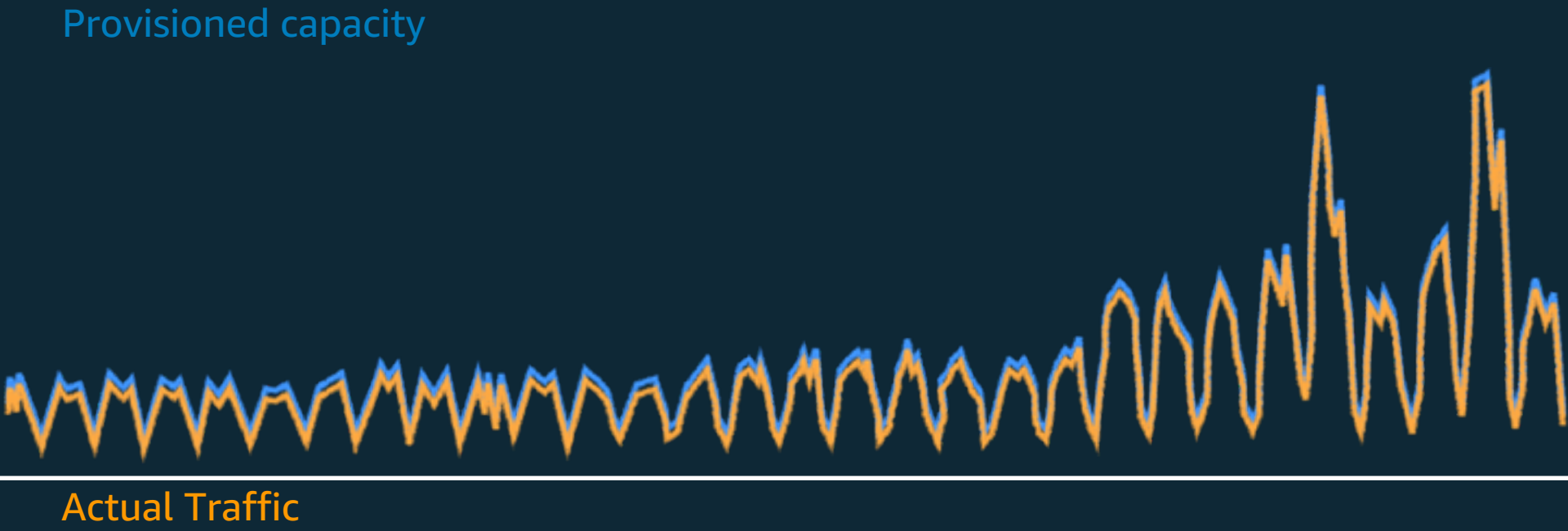
November traffic to Amazon.com



November traffic to Amazon.com



November traffic to Amazon.com



Implement Elasticity

How To Guide:

- Write Auto Scaling policies with your specific application access patterns in mind
- Prepare your application to be flexible: don't assume the health, availability, or fixed location of components
- Architect resiliency to reboot and relaunch
    - When an instance launches, it should ask "*Who am I and what is my role?*"
- Leverage highly scalable, managed services such as S3 and DynamoDB

Think Parallel

Think Parallel

Scale Horizontally, Not Vertically

- Decouple compute from state/session data
- Use ELBs to distribute load
- Break up big data into pieces for distributed processing

aws

Think Parallel

Faster doesn’t need to mean more expensive

- With EC2 On Demand, the following will cost the same:
  - 12 hours of work using 4 vCPUs
  - 1 hour of work using 48 vCPUs
- Right Size your infrastructure to your workload to get the best balance between cost and performance

Think Parallel

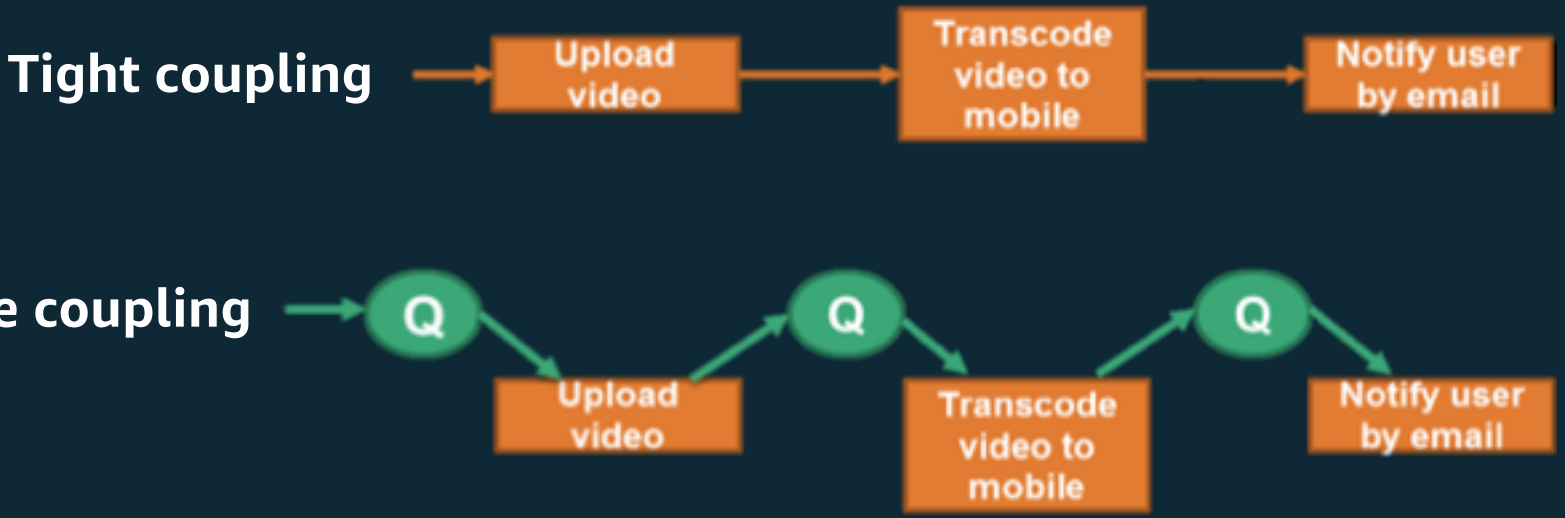
Parallelize using native managed services

- Get the best performance out of S3 with parallelized reads/writes
    - Multi-part uploads (API) and byte-range GETs (HTTP)
- Take on high concurrency with Lambda
    - Initial soft limit: 1000 concurrent requests per region

Loose Coupling Sets You Free

Loose Coupling Sets You Free: Queueing

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



Loose Coupling Sets You Free: Don’t Reinvent the Wheel

Nearly everything in AWS is an API call. Leverage AWS Native Services for…

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

Amazon SQS

Amazon CloudWatch

Amazon ElasticSearch

Amazon SES

AWS CloudTrail

Amazon Elastic Transcoder

Amazon SNS

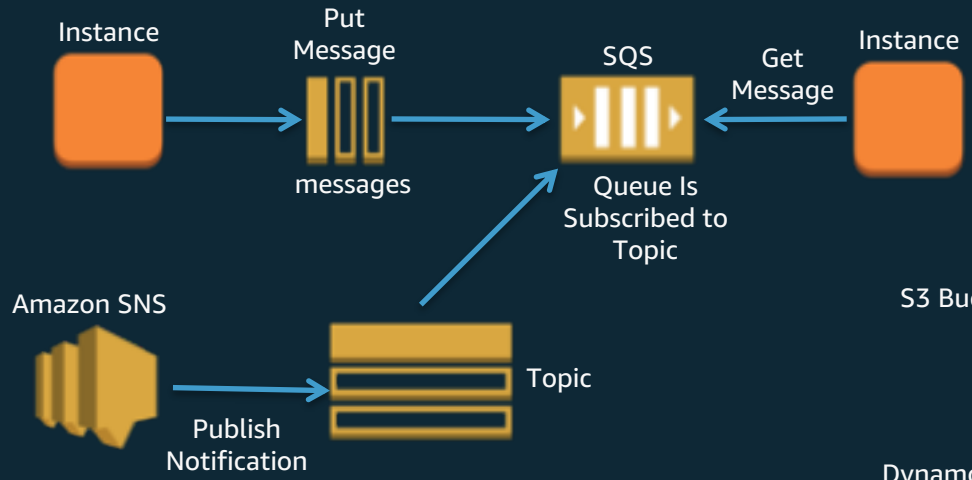
Amazon RDS

AWS Lambda

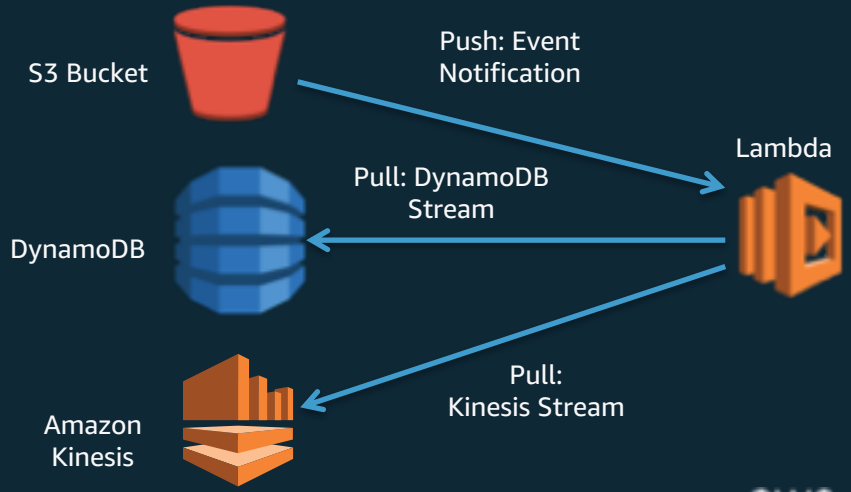
aws

Loose Coupling Sets You Free

Using SNS and SQS to asynchronously scale:



Using Lambda triggers to decouple actions:



Don’t Fear Constraints

Don’t Fear Constraints

Rethink traditional architectural constraints

Need more RAM?

- Don’t: vertically scale
- Do: distribute load across machines or a shared cache

Need better IOPS for database?

- Don’t: rework schema/indexes or vertically scale
- Do: create read replicas, implement sharding, add a caching layer

Hardware failed or config got corrupted?

- Don’t: waste production time diagnosing the problem
- Do: “Rip and replace” – stop/terminate old instance and relaunch

Need a Cost Effective Disaster Recovery (DR) strategy?

- Don’t: double your infrastructure costs when you don’t need to
- Do: implement Pilot Light or Warm Standby DR stacks

aws

Any Questions?