DVO401

Deep Dive into Blue/Green Deployments on AWS

Andy Mui, Vlad Vlasceanu

October 2015

amazon web services

What to expect from the session

- Overview of common deployment risks
- Blue/green deployment concepts
- Benefits of blue/green with AWS
- Deploying apps using blue/green patterns on AWS
- Best practices for the data tier
- Cost optimization

Deployments are not easy

```
[stormtrooper1337@deathstar ~]$ cat deploy.sh
#!/bin/bash
#
# Last Updated: Luke S; Jan 13, 2010
# Desc: Use me to deploy code to app servers. I will install needed
# dependencies and ship the code. Before using me, please make sure
# your local unit tests pass. Backup directory is /backup/* in case
# something fails and you need to roll back. Talk to Tim if the host
# becomes unreachable. Good luck and feel free to improve me!
#
# Usage:
# deploy.sh <app version> <QA|STAGE|PROD>
#
APPDIR="/src/superlaser/$2"
HOSTNAME=$(hostname)
BEGIN="$(date)"

echo "Starting upgrade process - ${BEGIN}"

case $2 in
  QA)
    echo "Deploying QA code to ${HOSTNAME}"
    ;;
```

- Traditional environments favor in-place upgrades
- Resource constraints
- Downtime
- Dependencies
- Process coordination
- Difficult rollback

Common deployment risks

Challenges	Business Impacts
Application failure	Downtime
Infrastructure failure	Data loss
Capacity issues	Bad customer experience
Scaling issues	Lost revenue
People failure	Grumpy managers
Process failure	Burned out staff
Rollback issues	Wasted time/resources

Defining blue/green deployment on AWS

What is blue/green deployment?

"Blue"

(existing production environment)

"Green"

(parallel environment running a different version of the application)

"Deployment"

(ability to switch traffic between the two environments)

What is an environment?

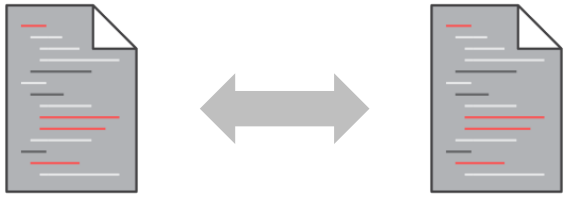
Boundary for where things changed and what needs to be deployed

Examples:

App component, app tier, microservice

Examples:

DNS, load balancer



Environment scope in AWS

Narrow

Wide

AMI

Auto Scaling group

Amazon EC2
Container Service

AWS Elastic
Beanstalk

AWS
OpsWorks

AWS
CloudFormation

Define your environment boundary

Factors	Criteria
Application architecture	Dependencies, loosely/tightly coupled
Organizational	Speed and number of iterations
Risk and complexity	Blast radius and impact of failed deployment
People	Expertise of teams
Process	Testing/QA, rollback capability
Cost	Operating budgets

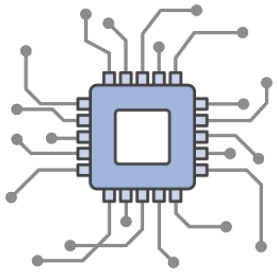
Every deployment is different in scope and risk.

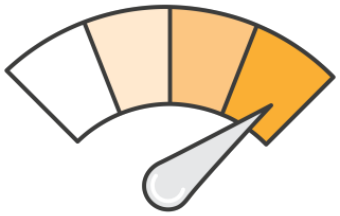
➔ Need a platform that is flexible and has powerful automation tools.

Benefits of blue/green deployment on AWS

AWS:

- Agile deployments
- Flexible options
- Scalable capacity
- Pay for what you use
- Automation capabilities



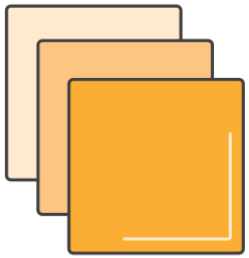


Deploying apps using blue/green patterns on AWS

Deploying apps using blue/green patterns

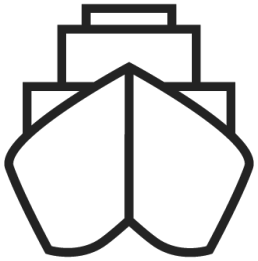
Using EC2 instances

1. Classic DNS cutover
2. Swap Auto Scaling groups
3. Swap launch configurations



Using EC2 Container Service

1. Swap ECS services via DNS
2. Swap ECS services behind ELB
3. Swap ECS task definitions



Common thread: Environment automation

Deployment success depends on mitigating risk for:

- Application issues (functional)
- Application performance
- **People/process errors**
- **Infrastructure failure**
- Rollback capability
- Large costs

Strength of automation platform

CloudFormation most comprehensive automation platform

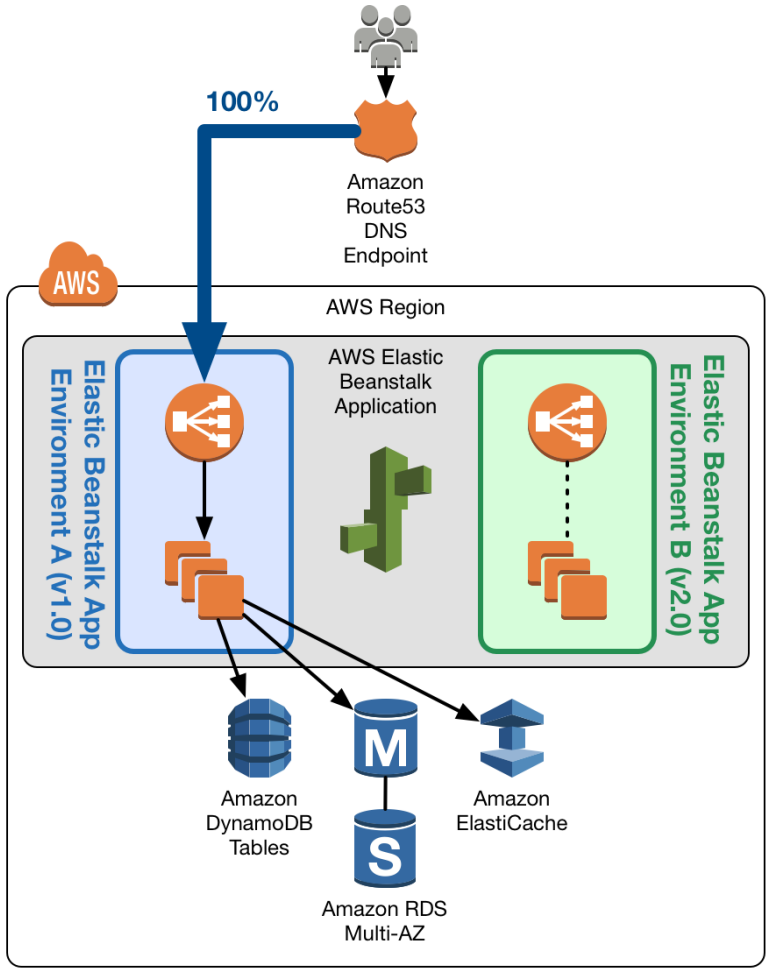
- Scope stacks from network to software
- Control higher-level automation services: Elastic Beanstalk, ECS, OpsWorks, Auto Scaling

Blue/green deployment patterns address these risks differently

Patterns: classic DNS cutover

Deployment process:

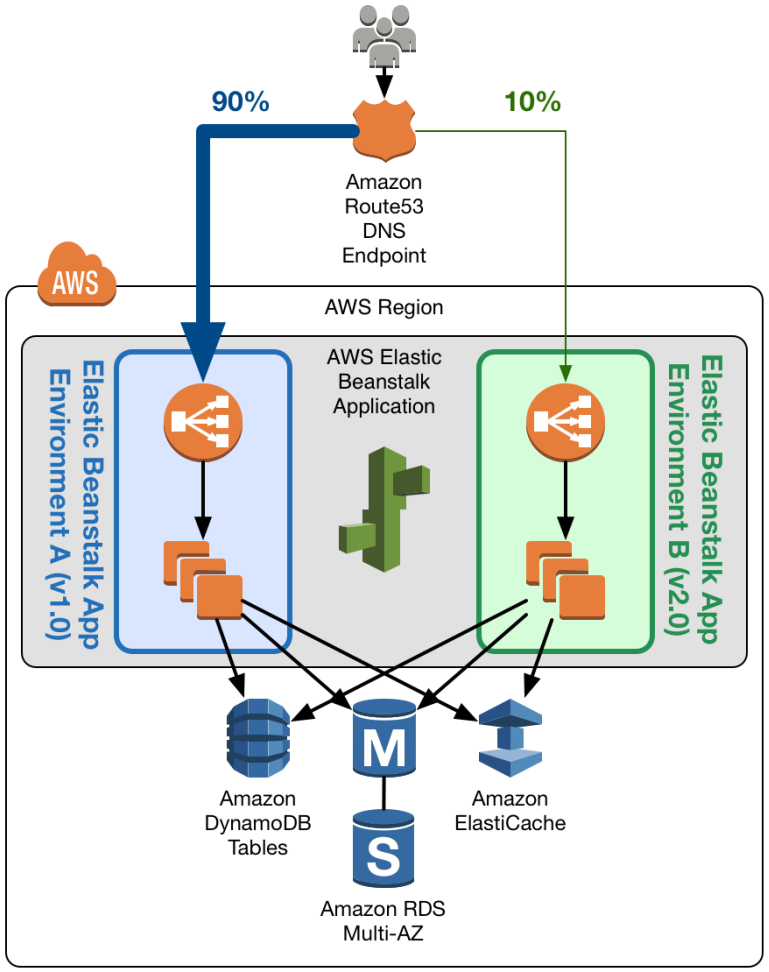
- Start with **current** app environment
- Deploy a **new** app environment
- **Test** the green stack
- Gradually **cut traffic over** via DNS
- **Monitor** your environments
- If needed, **roll back** to blue



Patterns: classic DNS cutover

Deployment process:

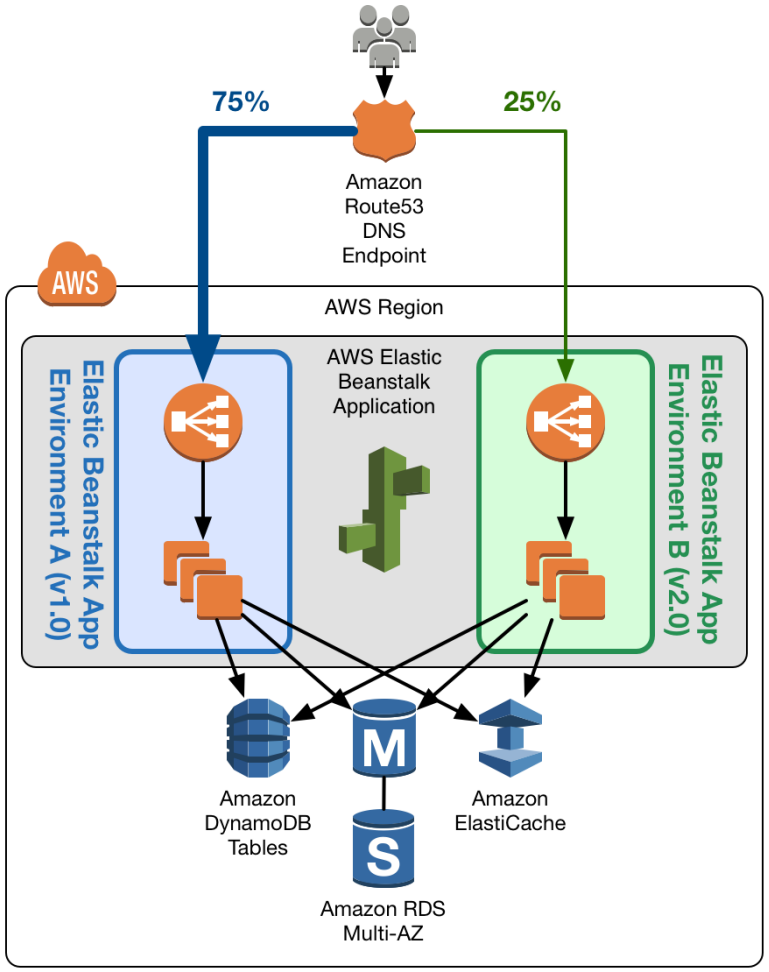
- Start with **current** app environment
- Deploy a **new** app environment
- **Test** the green stack
- Gradually **cut traffic over** via DNS
- **Monitor** your environments
- If needed, **roll back** to blue



Patterns: classic DNS cutover

Deployment process:

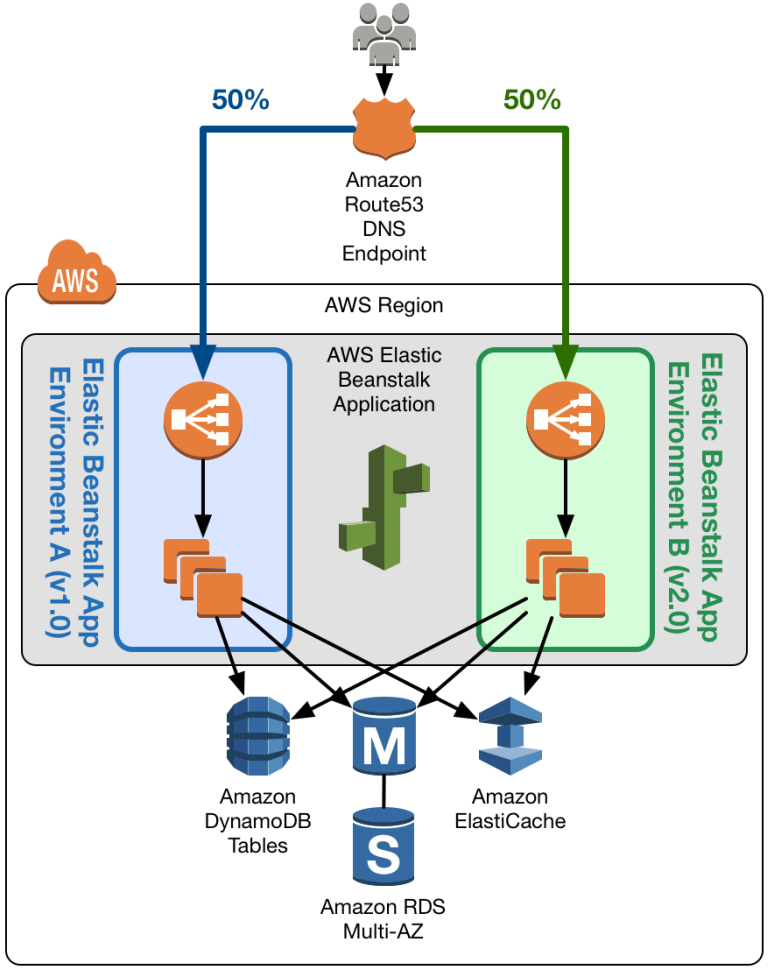
- Start with **current** app environment
- Deploy a **new** app environment
- **Test** the green stack
- Gradually **cut traffic over** via DNS
- **Monitor** your environments
- If needed, **roll back** to blue



Patterns: classic DNS cutover

Deployment process:

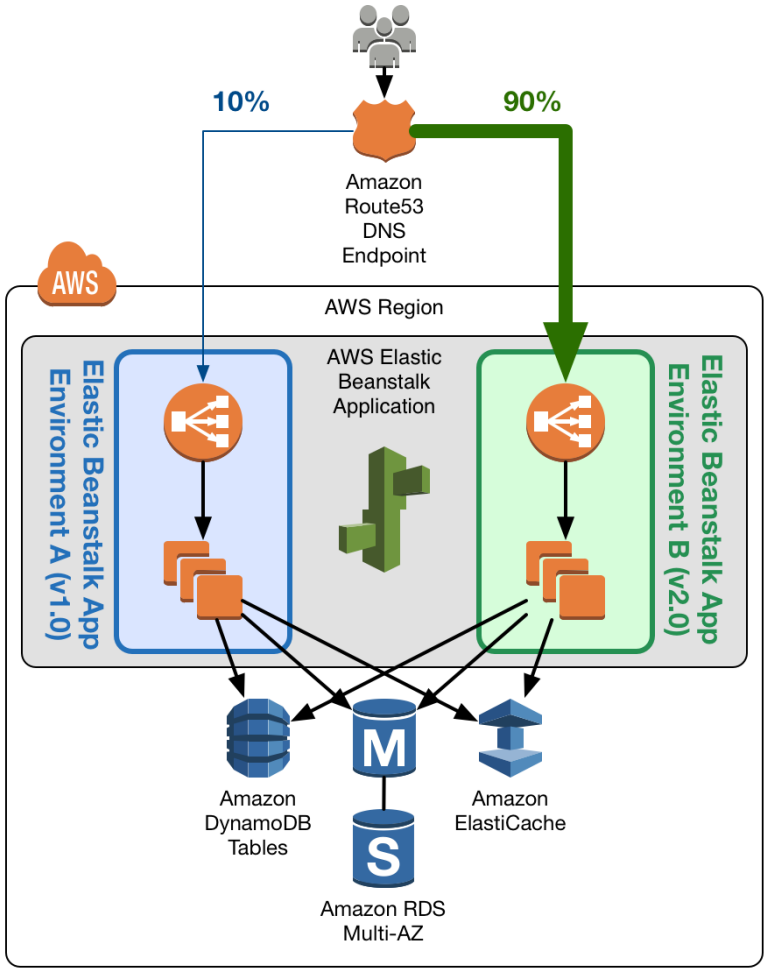
- Start with **current** app environment
- Deploy a **new** app environment
- **Test** the green stack
- Gradually **cut traffic over** via DNS
- **Monitor** your environments
- If needed, **roll back** to blue



Patterns: classic DNS cutover

Deployment process:

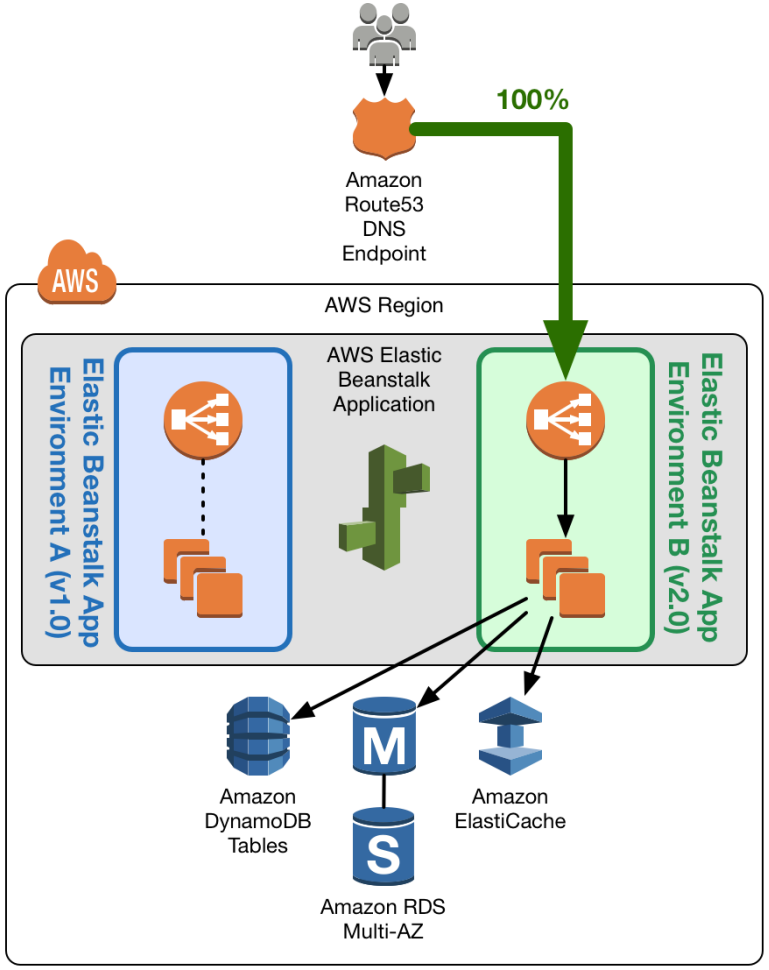
- Start with **current** app environment
- Deploy a **new** app environment
- **Test** the green stack
- Gradually **cut traffic over** via DNS
- **Monitor** your environments
- If needed, **roll back** to blue



Patterns: classic DNS cutover

Deployment process:

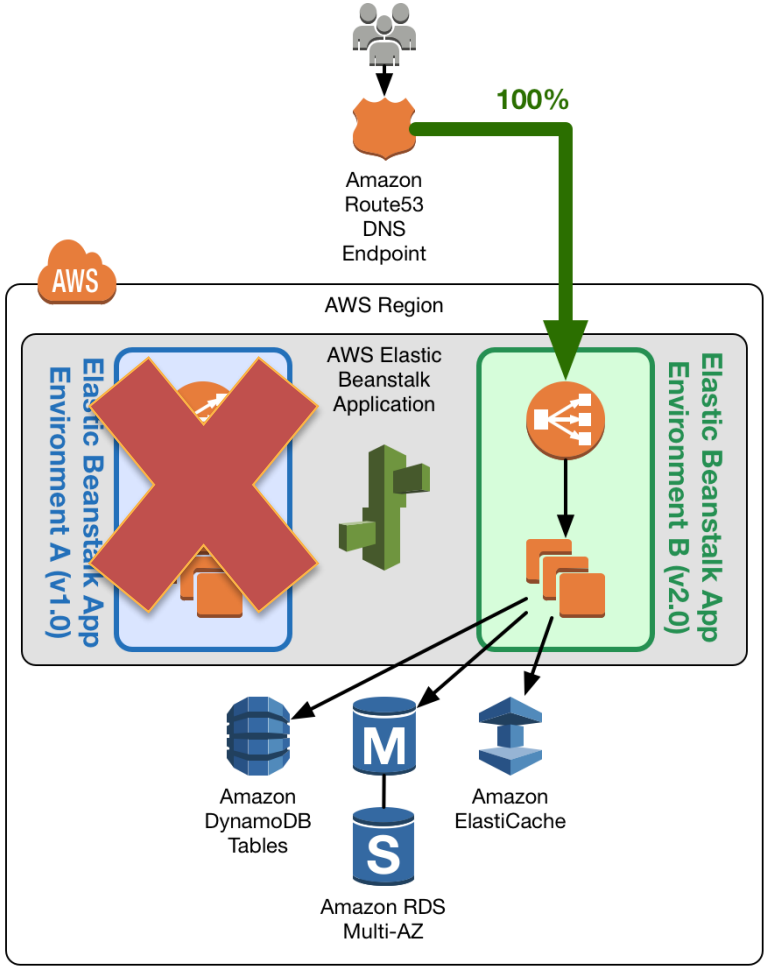
- Start with **current** app environment
- Deploy a **new** app environment
- **Test** the green stack
- Gradually **cut traffic over** via DNS
- **Monitor** your environments
- If needed, **roll back** to blue



Patterns: classic DNS cutover

Deployment process:

- Start with **current** app environment
- Deploy a **new** app environment
- **Test** the green stack
- Gradually **cut traffic over** via DNS
- **Monitor** your environments
- If needed, **roll back** to blue



Automating your environment

- Use **CloudFormation** templates to model your environment
- Version-control your templates
- Use Elastic Beanstalk or OpsWorks to model your applications inside the template
- Update CloudFormation stack from updated template containing green environment

```
"Resources": {
  "myApp": { "Type": "AWS::ElasticBeanstalk::Application" },
  "myConfigTemplate": {
    "Type": "AWS::ElasticBeanstalk::ConfigurationTemplate"
  },
  "myBlueAppVersion": {
    "Type": "AWS::ElasticBeanstalk::ApplicationVersion"
  },
  "myBlueEnvironment": {
    "Type":"AWS::ElasticBeanstalk::Environment"
  },
  "myBlueEndpoint": {"Type": “AWS::Route53::RecordSet" },
  "myGreenAppVersion": {
    "Type": "AWS::ElasticBeanstalk::ApplicationVersion"
  },
  "myGreenEnvironment": {
    "Type": "AWS::ElasticBeanstalk::Environment"
  },
  "myGreenEndpoint": {"Type": "AWS::Route53::RecordSet" }
  ...
```

Amazon Route 53 weighted DNS switching

- AWS Elastic Beanstalk environment endpoint swap
- DNS record time-to-live (TTL)
  - Reaction time = (TTL × no. of DNS caches) + Route53 propagation time, up to 1min
  - Beware of misbehaving DNS clients
- Auto Scaling and Amazon Elastic Load Balancing (ELB) need time to scale
- Measurable metrics
  - ELB: Latency, SurgeQueueLength, SpillOverCount, BackendConnectionErrors
  - Your application metrics
- Your deployment goals

Amazon Route 53 weighted DNS switching

Using CloudFormation:

- Update template record sets with initial weighting information
- Consider using parameters for the weight values – reuse the same template
- Update CloudFormation stack with new weighting

```
"myBlueEndpoint": {
  "Type": "AWS::Route53::RecordSet",
  "Properties": {
    "HostedZoneId": { "Ref": "parameterHostedZoneId" },
    "Name": "www.example.com.", "Type": "CNAME", "TTL": "60",
    "SetIdentifier": "stack-blue", "Weight": "90",
    "ResourceRecords": [
      { "Fn::GetAtt": [ "myBlueEnvironment", "EndpointURL" ] }
] } },
"myGreenEndpoint": {
  "Type": "AWS::Route53::RecordSet",
  "Properties": {
    "HostedZoneId": { "Ref": "parameterHostedZoneId" },
    "Name": "www.example.com.", "Type": "CNAME", "TTL": "60",
    "SetIdentifier": "stack-green", "Weight": "10",
    "ResourceRecords": [
      { "Fn::GetAtt": [ "myGreenEnvironment", "EndpointURL" ] }
] } }
```

Pattern review: Classic DNS cutover

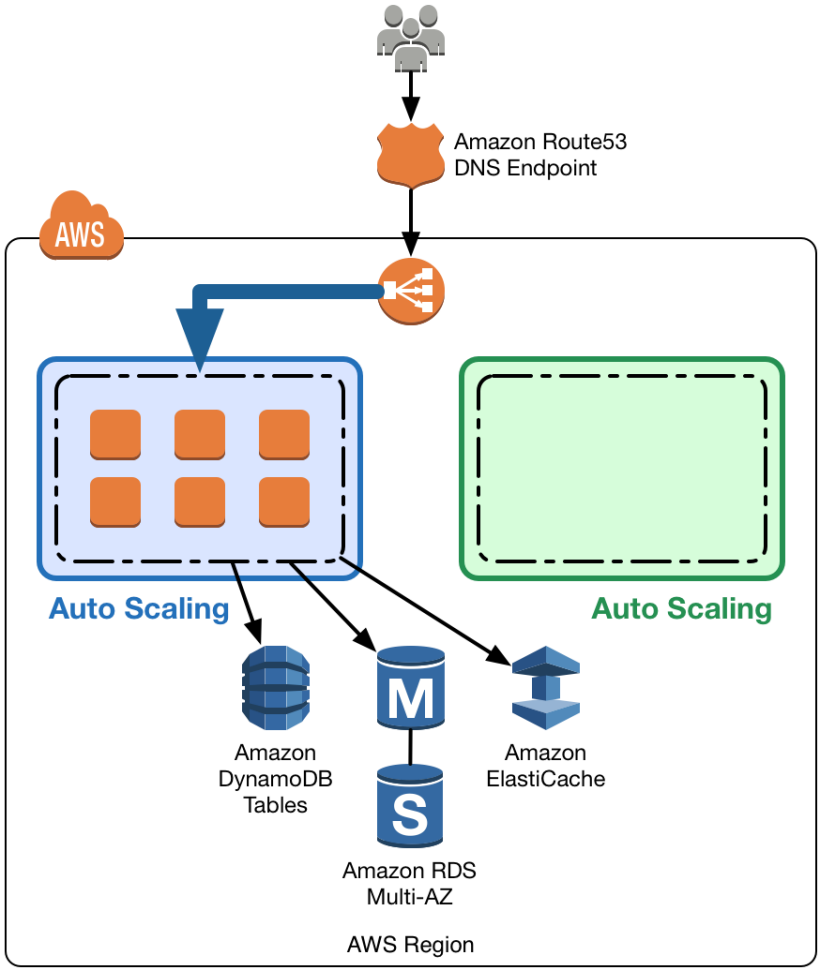
Risk category	Mitigation level	Reasoning
Application issues	Great	Facilitates canary analysis
Application performance	Great	Gradual switch, traffic split management
People/process errors	Good	Depends on automation framework
Infrastructure failure	Good	Depends on automation framework
Rollback	Fair	DNS TTL complexities (reaction time, flip/flop)
Cost	Great	Optimized via auto-scaling

Let’s remove DNS from the solution…

Pattern: swap Auto Scaling Groups

Deployment process:

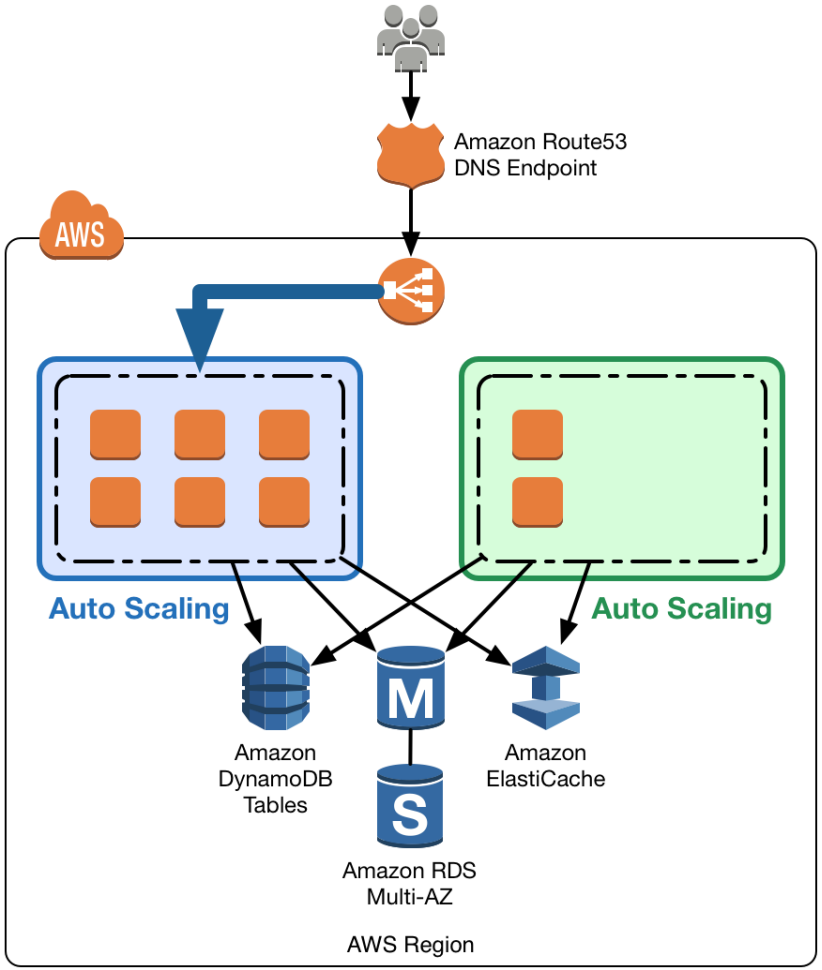
- Amazon Elastic Load Balancer (ELB) outside the environment boundary
- Start with **current** Auto Scaling Group (ASG)
- Deploy & scale out **new** ASG
- **Test** green stack
- **Register** green ASG with ELB
- **Remove** blue ASG from ELB



Pattern: swap Auto Scaling Groups

Deployment process:

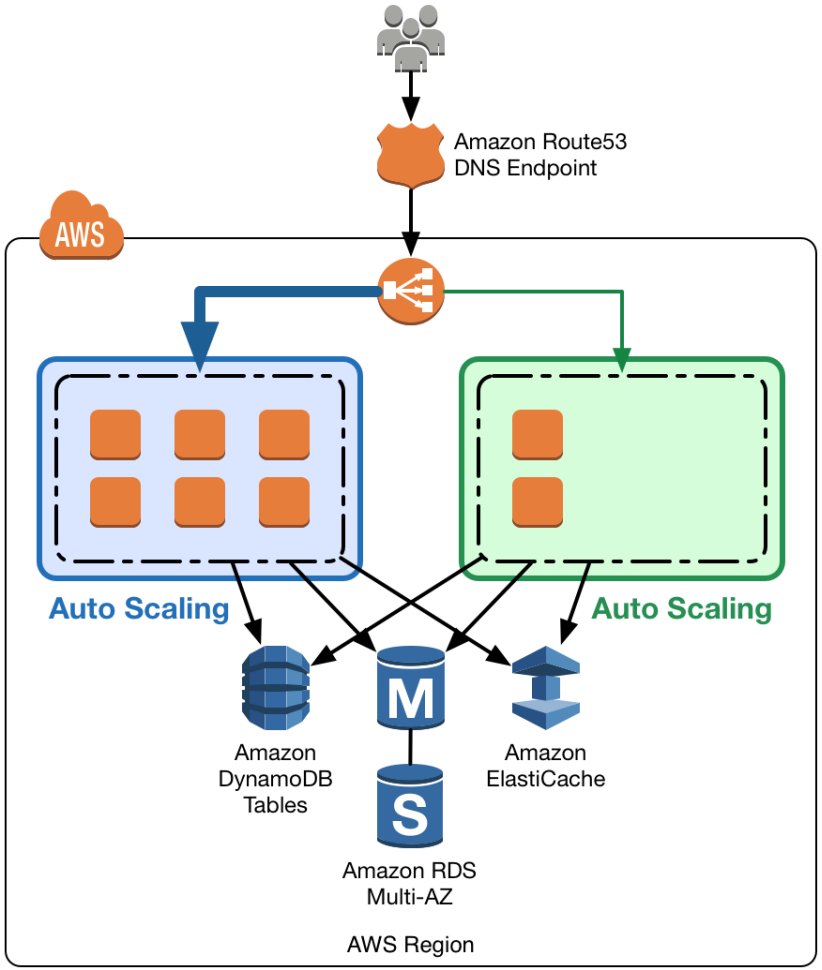
- Amazon Elastic Load Balancer (ELB) outside the environment boundary
- Start with **current** Auto Scaling Group (ASG)
- Deploy & scale out **new** ASG
- **Test** green stack
- **Register** green ASG with ELB
- **Remove** blue ASG from ELB



Pattern: swap Auto Scaling Groups

Deployment process:

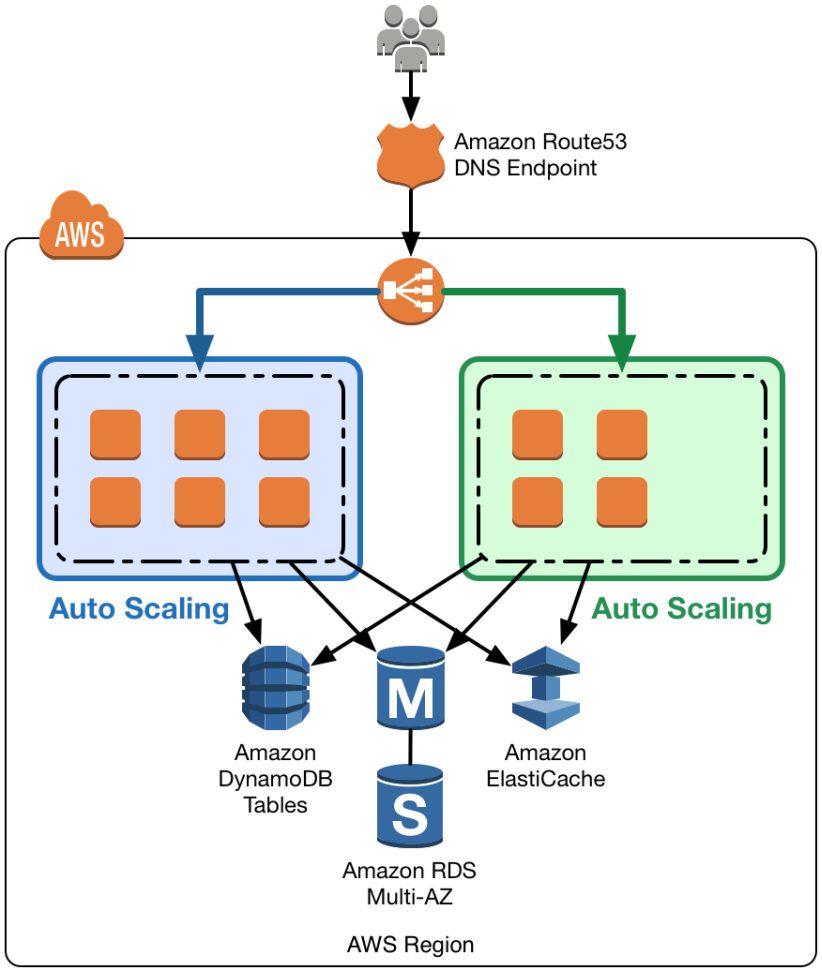
- Amazon Elastic Load Balancer (ELB) outside the environment boundary
- Start with **current** Auto Scaling Group (ASG)
- Deploy & scale out **new** ASG
- **Test** green stack
- **Register** green ASG with ELB
- **Remove** blue ASG from ELB



Pattern: swap Auto Scaling Groups

Deployment process:

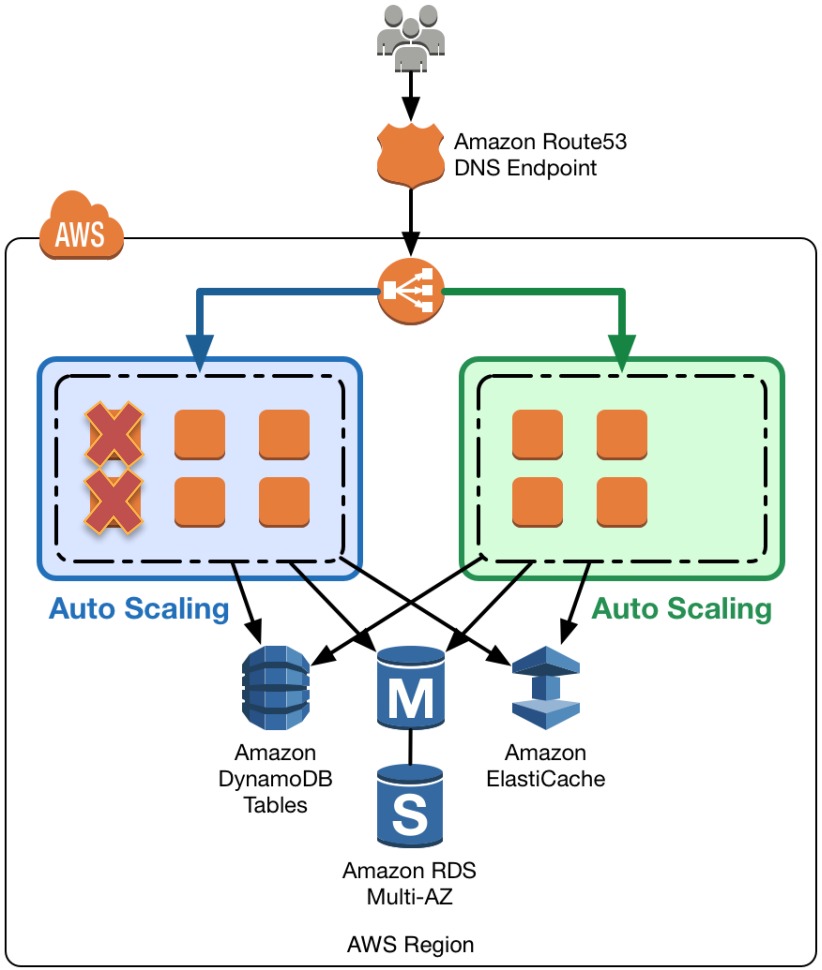
- Amazon Elastic Load Balancer (ELB) outside the environment boundary
- Start with **current** Auto Scaling Group (ASG)
- Deploy & scale out **new** ASG
- **Test** green stack
- **Register** green ASG with ELB
- **Remove** blue ASG from ELB



Pattern: swap Auto Scaling Groups

Deployment process:

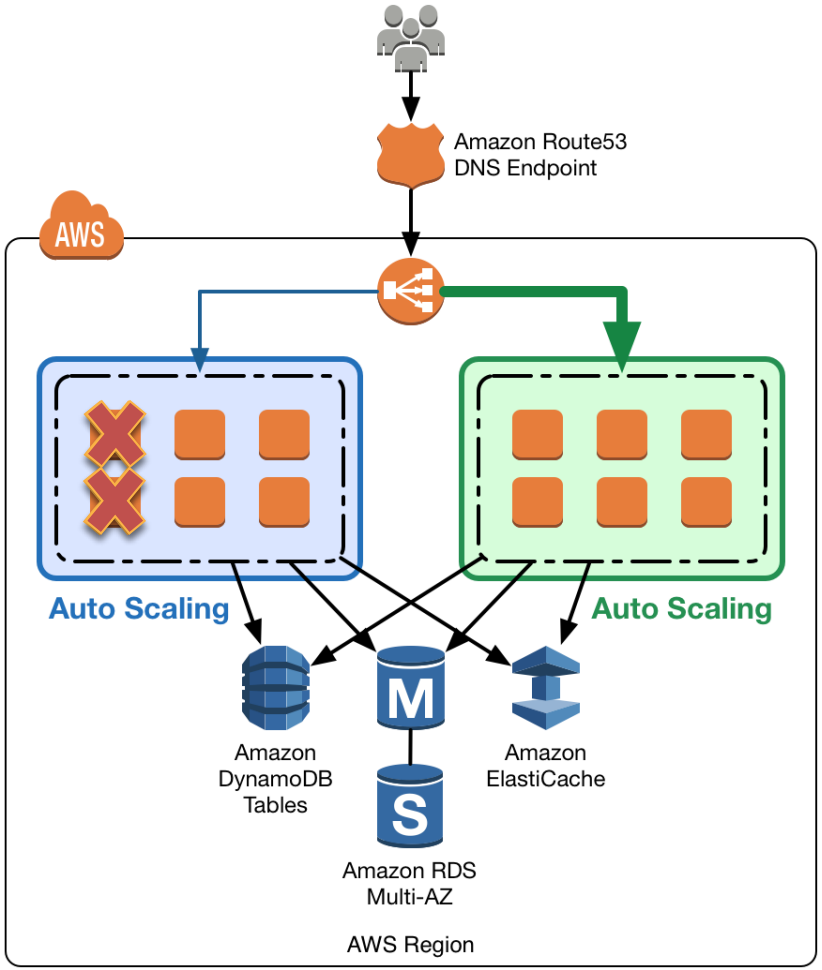
- Amazon Elastic Load Balancer (ELB) outside the environment boundary
- Start with **current** Auto Scaling Group (ASG)
- Deploy & scale out **new** ASG
- **Test** green stack
- **Register** green ASG with ELB
- **Remove** blue ASG from ELB



Pattern: swap Auto Scaling Groups

Deployment process:

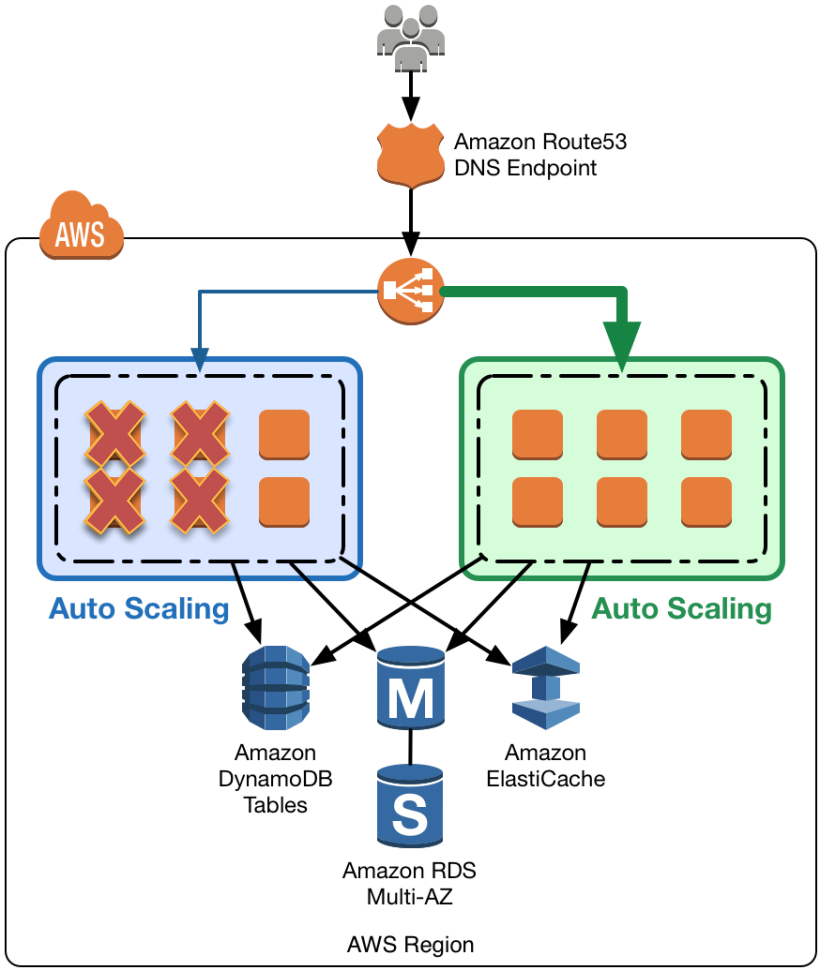
- Amazon Elastic Load Balancer (ELB) outside the environment boundary
- Start with **current** Auto Scaling Group (ASG)
- Deploy & scale out **new** ASG
- **Test** green stack
- **Register** green ASG with ELB
- **Remove** blue ASG from ELB



Pattern: swap Auto Scaling Groups

Deployment process:

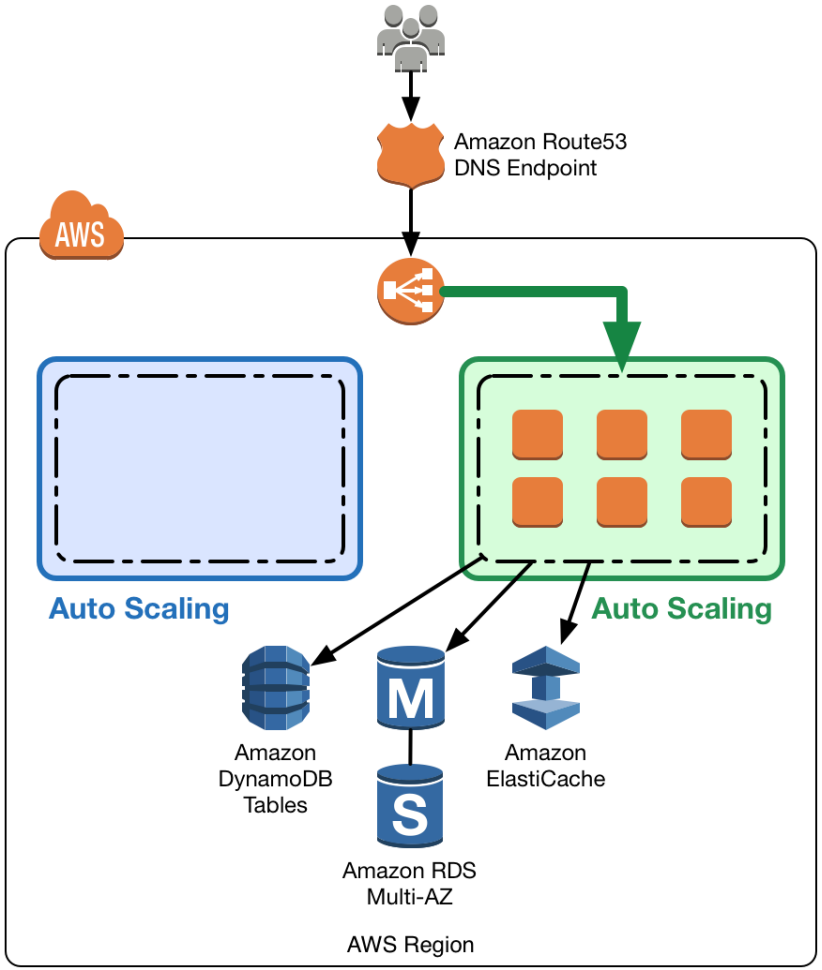
- Amazon Elastic Load Balancer (ELB) outside the environment boundary
- Start with **current** Auto Scaling Group (ASG)
- Deploy & scale out **new** ASG
- **Test** green stack
- **Register** green ASG with ELB
- **Remove** blue ASG from ELB



Pattern: swap Auto Scaling Groups

Deployment process:

- Amazon Elastic Load Balancer (ELB) outside the environment boundary
- Start with **current** Auto Scaling Group (ASG)
- Deploy & scale out **new** ASG
- **Test** green stack
- **Register** green ASG with ELB
- **Remove** blue ASG from ELB



Swapping Auto Scaling groups behind ELB

- Register with ELB:
  - One or more EC2 instances
  - One or more Auto Scaling groups
- Least outstanding requests algorithm favors green ASG instances for new connections
- Connection draining - gracefully stop receiving traffic
- Scale out green ASG before ELB registration
- Put blue instances in standby

```
$ aws autoscaling attach-load-balancers \
    --auto-scaling-group-name "green-asg" \
    --load-balancer-names "my-app-elb"

$ aws autoscaling set-desired-capacity \
    --auto-scaling-group-name "green-asg" \
    --desired-capacity X

$ aws autoscaling detach-load-balancers \
    --auto-scaling-group-name "blue-asg" \
    --load-balancer-names "my-app-elb"

$ aws autoscaling enter-standby \
    --instance-ids i-xxxxxxxx \
    --auto-scaling-group-name "blue-asg" \
    --should-decrement-desired-capacity
```

Pattern review: Swap Auto Scaling groups

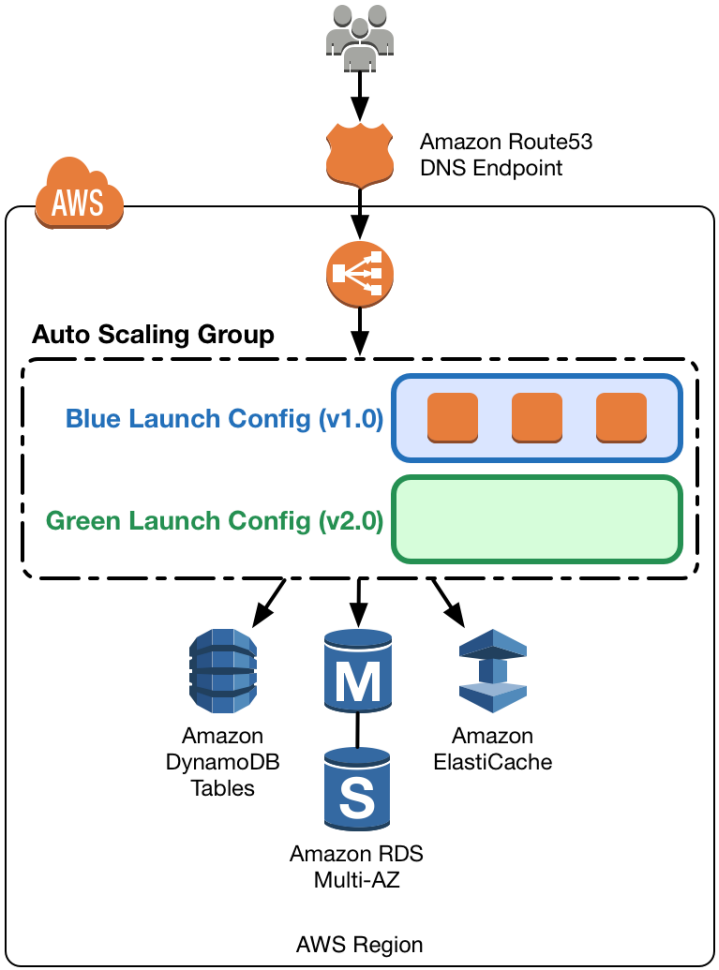
Risk category	Mitigation level	Reasoning
Application issues	Great	Facilitates canary analysis w/ additional ELB
Application performance	Good	Traffic split management, but less granular, pre-warmed ELB
People/process errors	Good	Depends on automation framework
Infrastructure failure	Great	Auto-scaling
Rollback	Great	No DNS complexities
Cost	Great	Optimized via auto-scaling

Let’s reduce the environment boundary further…

Pattern: swap Launch Configurations

Deployment process:

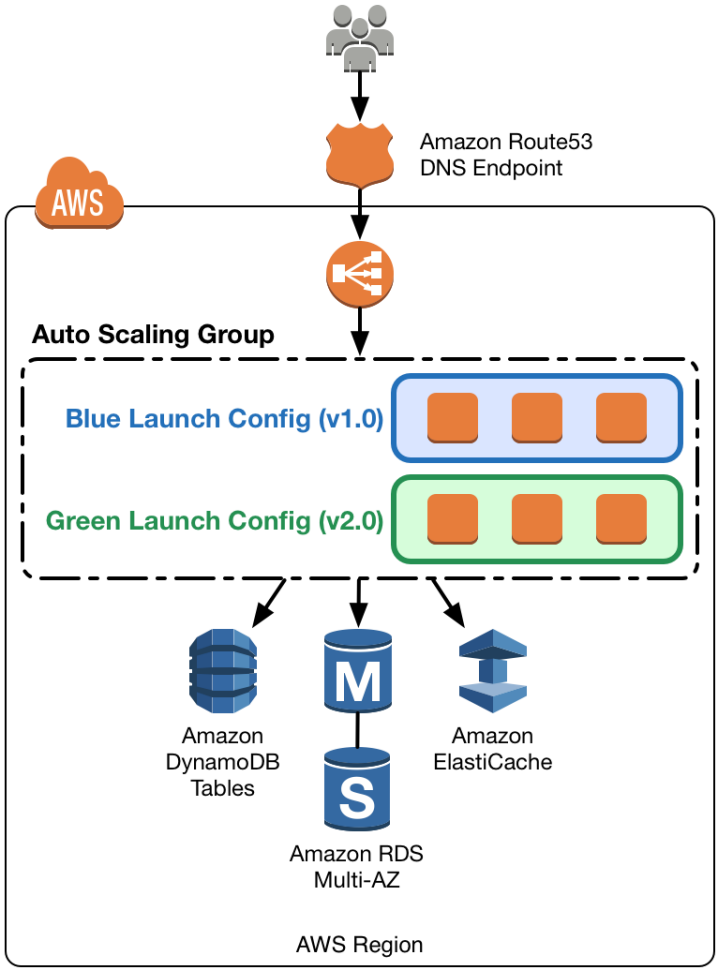
- Start with **current** ASG & Launch Configuration behind the ELB
- **Attach** updated **green** Launch Configuration to the ASG
- Grow the ASG gradually to 2x original size
- Shrink the ASG back to original size
- For more control, put old instances into Standby



Pattern: swap Launch Configurations

Deployment process:

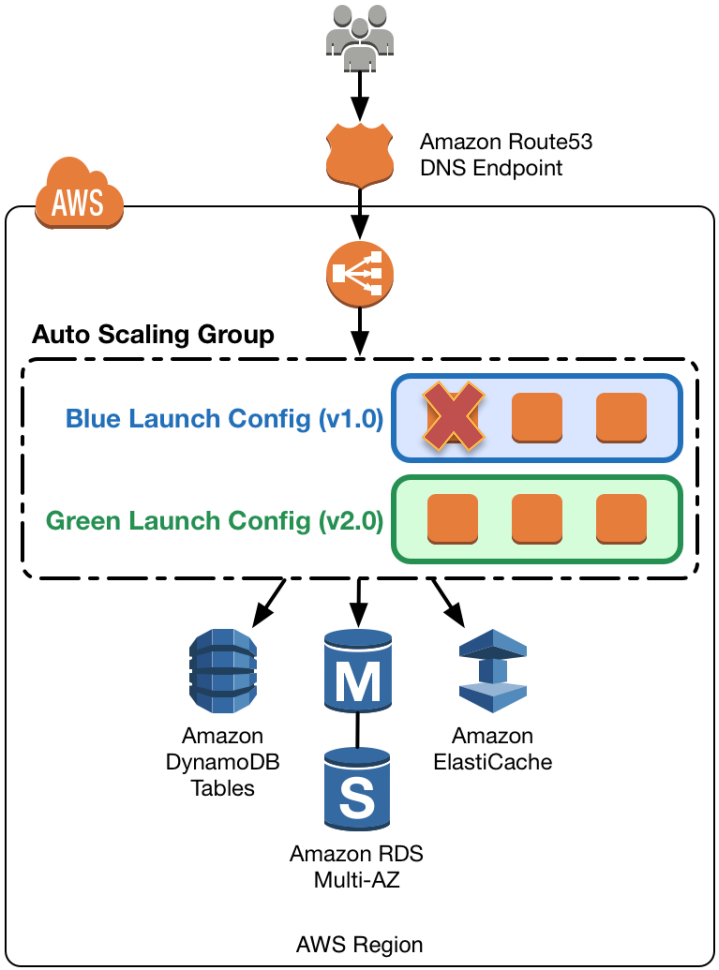
- Start with **current** ASG & Launch Configuration behind the ELB
- **Attach** updated **green** Launch Configuration to the ASG
- Grow the ASG gradually to 2x original size
- Shrink the ASG back to original size
- For more control, put old instances into Standby



Pattern: swap Launch Configurations

Deployment process:

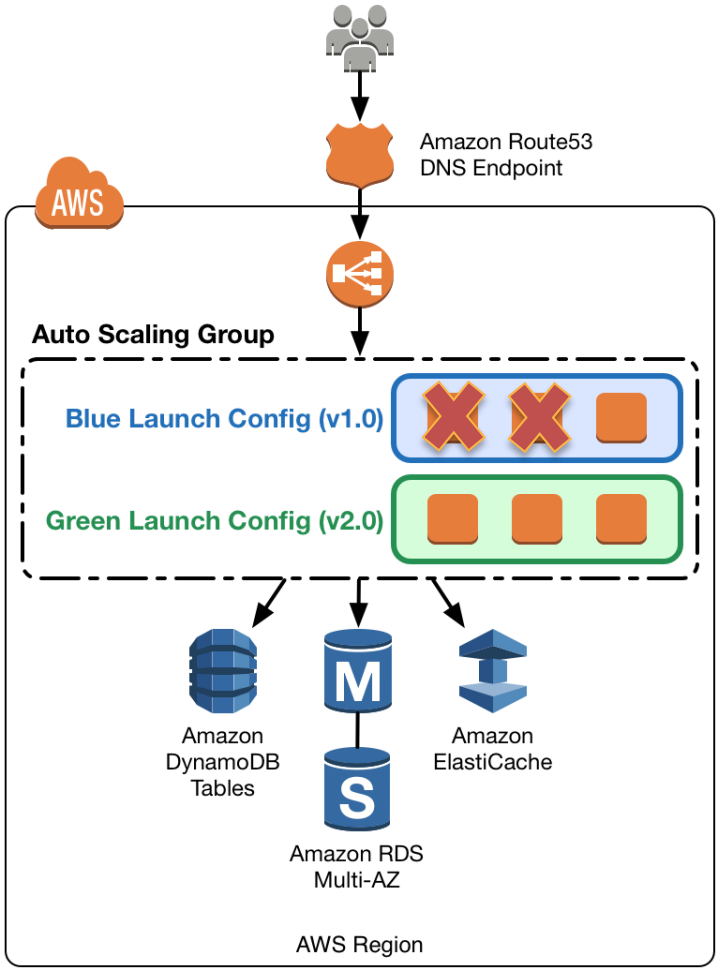
- Start with **current** ASG & Launch Configuration behind the ELB
- **Attach** updated **green** Launch Configuration to the ASG
- Grow the ASG gradually to 2x original size
- Shrink the ASG back to original size
- For more control, put old instances into Standby



Pattern: swap Launch Configurations

Deployment process:

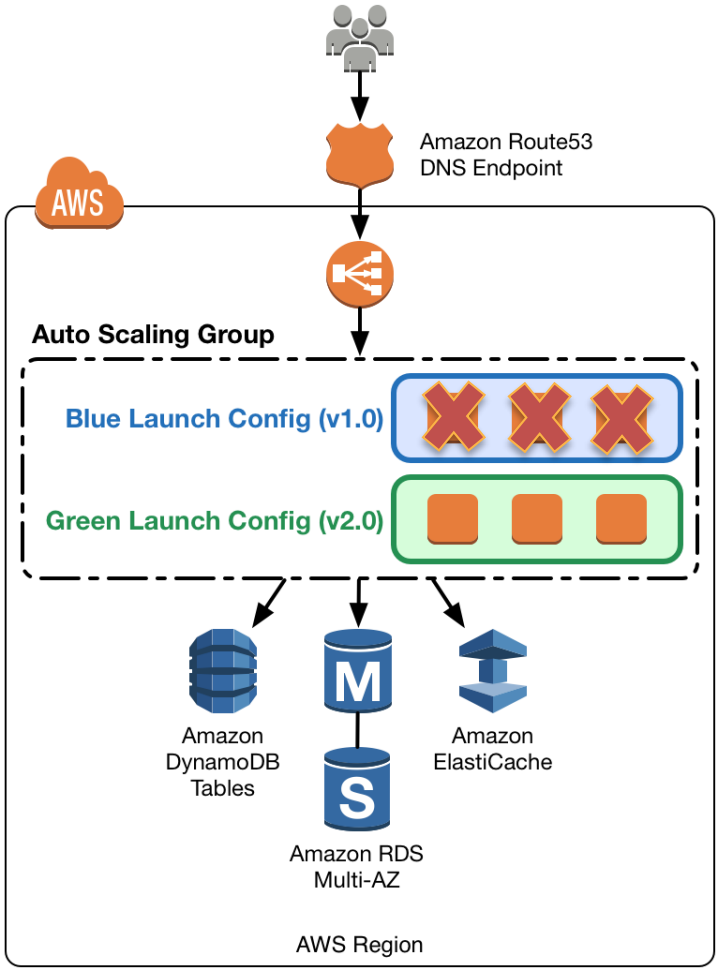
- Start with **current** ASG & Launch Configuration behind the ELB
- **Attach** updated **green** Launch Configuration to the ASG
- Grow the ASG gradually to 2x original size
- Shrink the ASG back to original size
- For more control, put old instances into Standby



Pattern: swap Launch Configurations

Deployment process:

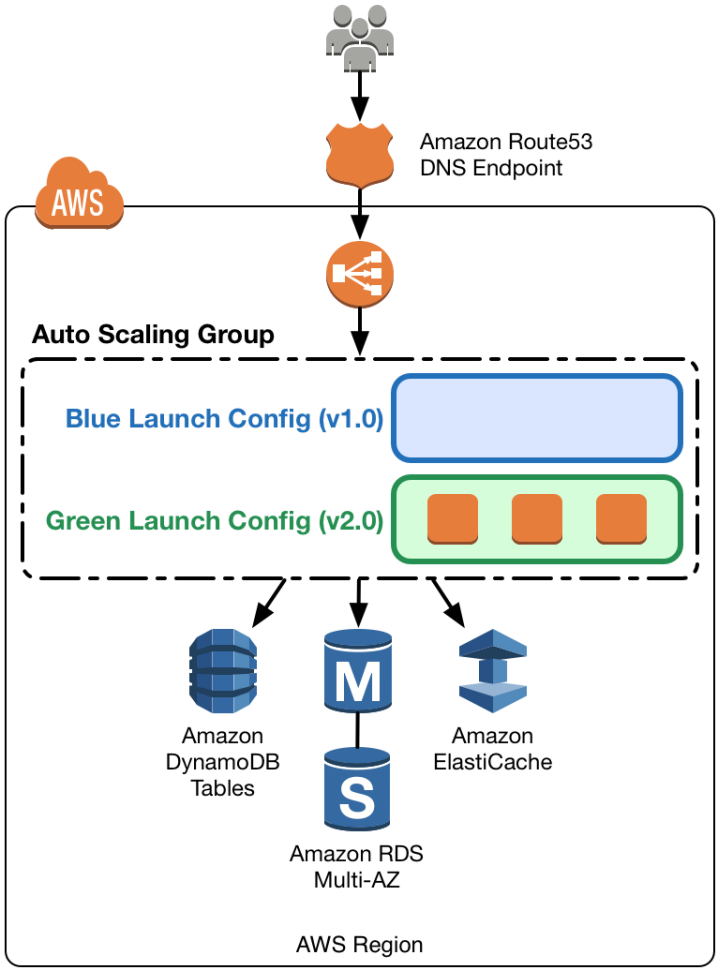
- Start with **current** ASG & Launch Configuration behind the ELB
- **Attach** updated **green** Launch Configuration to the ASG
- Grow the ASG gradually to 2x original size
- Shrink the ASG back to original size
- For more control, put old instances into Standby



Pattern: swap Launch Configurations

Deployment process:

- Start with **current** ASG & Launch Configuration behind the ELB
- **Attach** updated **green** Launch Configuration to the ASG
- Grow the ASG gradually to 2x original size
- Shrink the ASG back to original size
- For more control, put old instances into Standby



Swapping launch configurations

- **Launch configurations:**
  Blueprints for ASG instance provisioning, each ASG points to exactly one
- **Scale-out & replacement:**
  Events will use the attached (**green**) launch configuration to provision instances
- **Scale-in:**
  ASG scale-in events will terminate instances with oldest launch configuration first **while** trying to keep capacity in AZs balanced
- May need to address AZ imbalances separately
- **Temporarily remove instances from ASG**
  Place specific ASG instances (**blue**) into standby – stop receiving traffic

Pattern review: swap Launch Configurations

Risk Category	Mitigation Level	Reasoning
Application Issues	Fair	Detection of errors/issues in a heterogeneous fleet is complex
Application Performance	Fair	Less granular traffic split, initial traffic load
People/Process Errors	Good	Depends on automation framework
Infrastructure Failure	Great	Auto-Scaling
Rollback	Great	No DNS complexities
Cost	Good	Optimized via auto-scaling, but initial scale-out overprovisions

What if we're running containers?

EC2 Container Service

Environment boundary:

Docker container -> Task definition

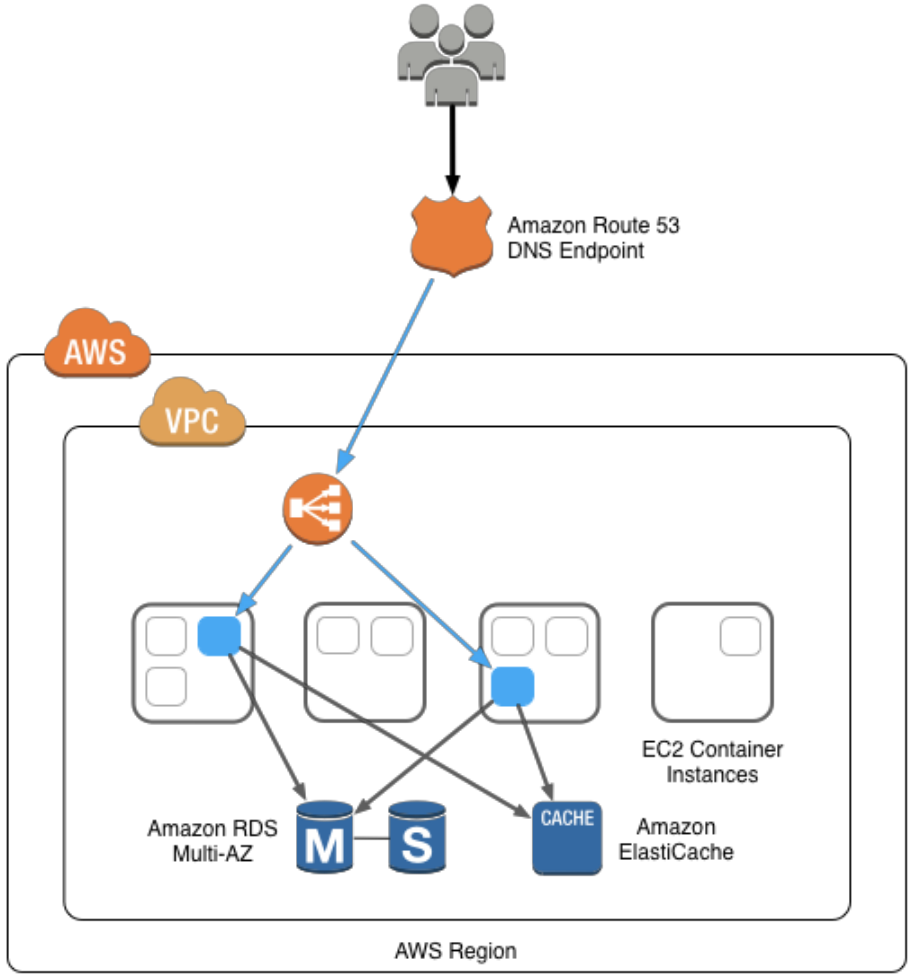
Deployment options:

1. Blue/green ECS services, DNS update
2. Blue/green ECS services, shared ELB
3. ECS update

Pattern: Swap ECS services with DNS

Deployment process:

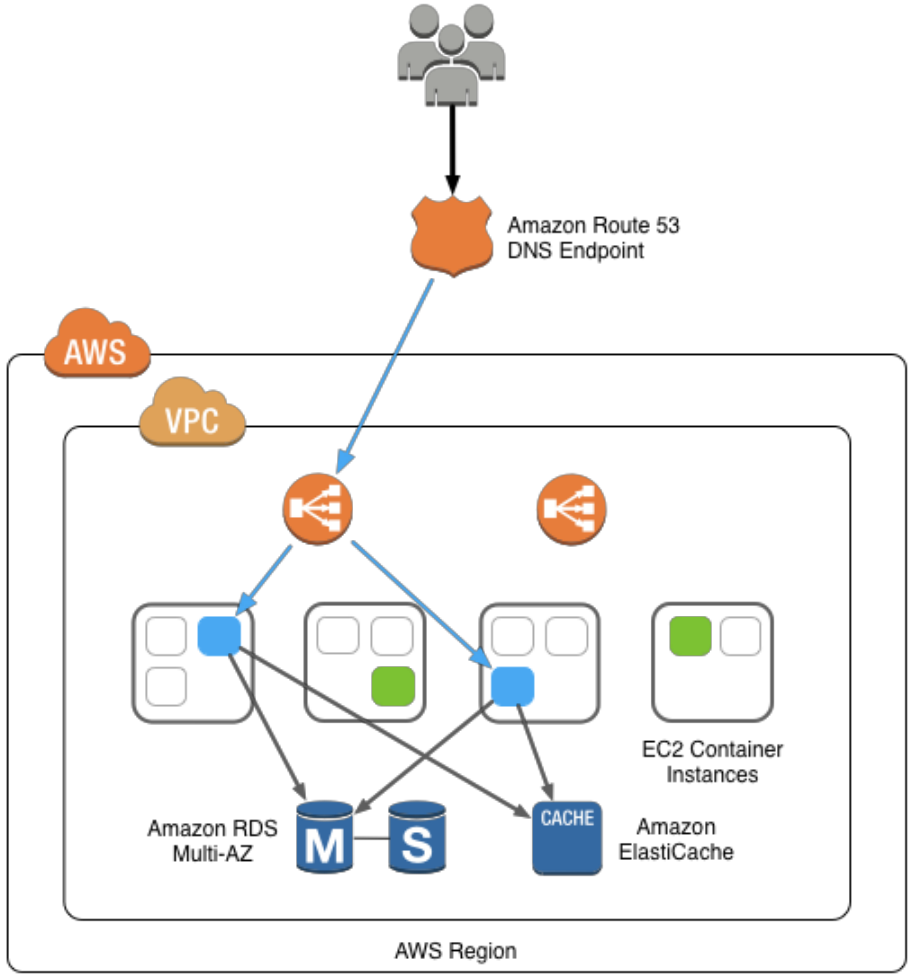
- Start with **blue** service composed of a task definition and ELB
- Create new task definition based on new version of Docker image and a new ELB
- Create **green** service with new task definition and ELB
- Update Route 53 alias record to direct traffic to new ELB endpoint
- Clean up blue service resources when no longer needed



Pattern: Swap ECS services with DNS

Deployment process:

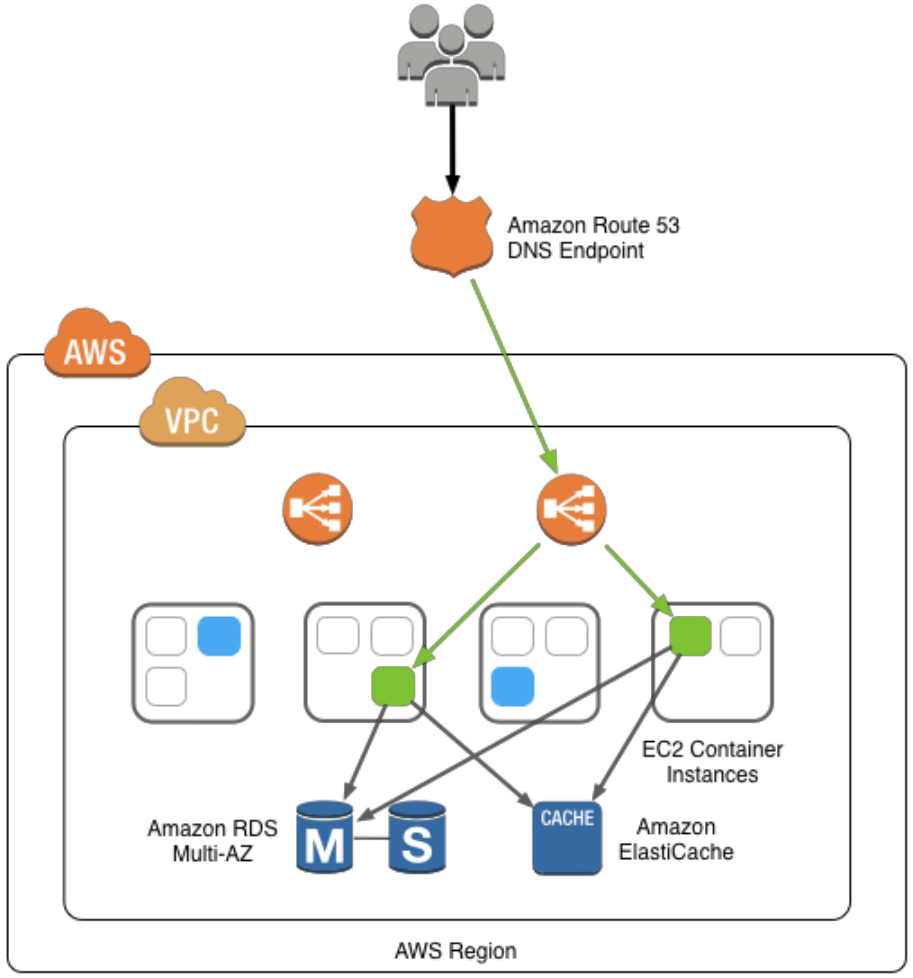
- Start with **blue** service composed of a task definition and ELB
- Create new task definition based on new version of Docker image and a new ELB
- Create **green** service with new task definition and ELB
- Update Route 53 alias record to direct traffic to new ELB endpoint
- Clean up blue service resources when no longer needed



Pattern: Swap ECS services with DNS

Deployment process:

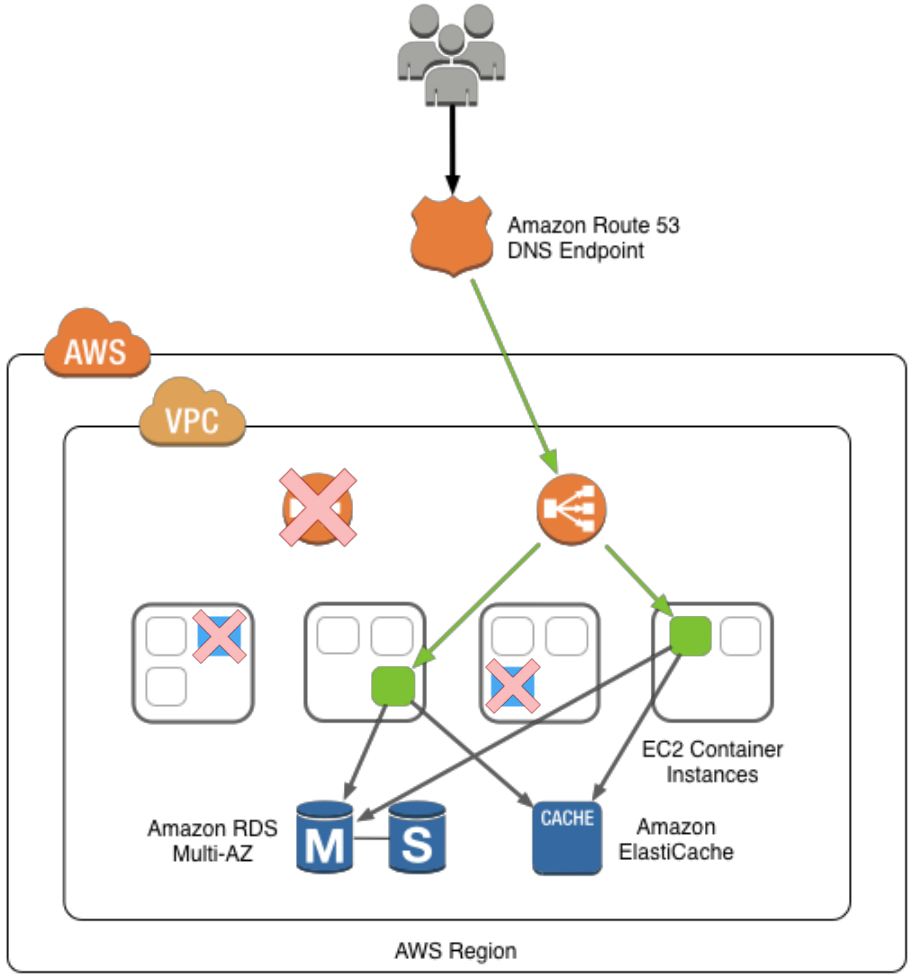
- Start with **blue** service composed of a task definition and ELB
- Create new task definition based on new version of Docker image and a new ELB
- Create **green** service with new task definition and ELB
- Update Route 53 alias record to direct traffic to new ELB endpoint
- Clean up blue Service resources when no longer needed



Pattern: Swap ECS services with DNS

Deployment process:

- Start with **blue** service composed of a task definition and ELB
- Create new task definition based on new version of Docker image and a new ELB
- Create **green** service with new task definition and ELB
- Update Route 53 alias record to direct traffic to new ELB endpoint
- Clean up blue service resources when no longer needed



Swap ECS services with DNS

1. Create task definition

```
$ aws ecs register-task-definition \
--cli-input-json file:///Code/web.json
```

2. Create service and ELB

```
$ aws ecs create-service \
--cluster dice-demo \
--service-name web-service-v2 \
--task-definition web-tier \
--load-balancers loadBalancerName=web-tier-
v2,containerName=web-tier,containerPort=8000 \
--desired-count 2 \
--role "ecsServiceRole"
```

web.json

```
{
  "containerDefinitions": [
    {
      "name": "web-tier",
      "image": "amui/web-test:v2",
      "cpu": 256,
      "memory": 256,
      "entryPoint": [],
      "environment": [],
      "command": [],
      "portMappings": [
        {
          "hostPort": 8000,
          "containerPort": 8000,
          "protocol": "tcp"
        }
      ],
      "essential": true
    }
  ],
  "family": "web-tier"
}
```

Swap ECS services with DNS

3. Route traffic to **green** ELB endpoint

```
$ aws route53 change-resource-record-sets \
--hosted-zone-id Zxxxxxxxxxxxx \
--change-batch file:///Code/dns.json
```

dns.json

```
{
  "Comment": "Update alias record to route
traffic from Blue to Green ELB for Dice Demo
ECS Service",
  "Changes": [
    {
      "Action": "UPSERT",
      "ResourceRecordSet": {
        "Name": "demo.example.com",
        "Type": "A",
        "AliasTarget": {
          "HostedZoneId": "Zxxxxxxxxxxx",
          "DNSName": "dice-demo-green-
1234567890.us-west-2.elb.amazonaws.com",
          "EvaluateTargetHealth": false
        }
 …
```

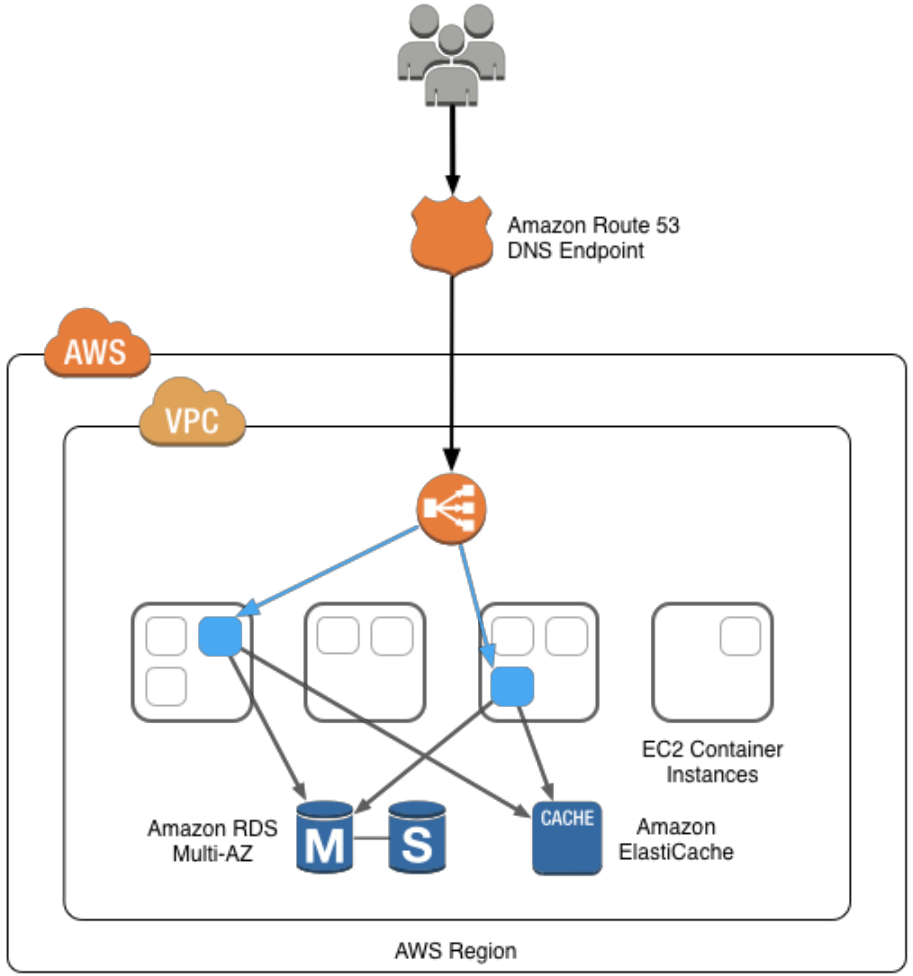
Pattern review: Swap ECS services with DNS

Risk category	Mitigation level	Reasoning
Application issues	Great	Full cutover or weighted canary analysis
Application performance	Good	ELB may require prewarm, CloudWatch, and automation to scale ECS
People/process errors	Great	Simple process
Infrastructure failure	Good	Leverage CloudWatch, Auto Scaling, ELB
Rollback	Fair	DNS TTL complexities (reaction time, flip/flop)
Cost	Fair	Require enough cluster resources to accommodate new service

Pattern: Swap ECS services with ELB

Deployment process:

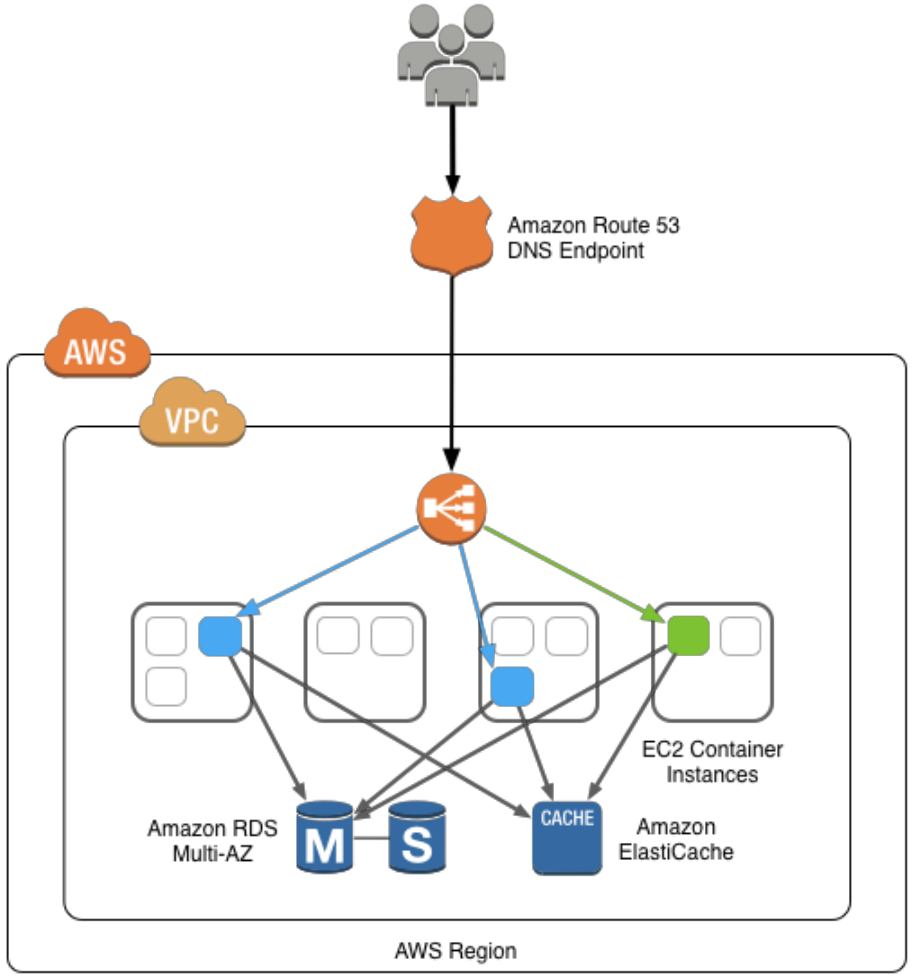
- Start with **blue** service composed of a task definition and ELB
- Create new task definition based on new version of Docker image
- Create **green** service with new task definition and map to existing ELB
- Scale up green service by incrementing number of tasks
- Decommission blue service by setting task count to 0



Pattern: Swap ECS services with ELB

Deployment process:

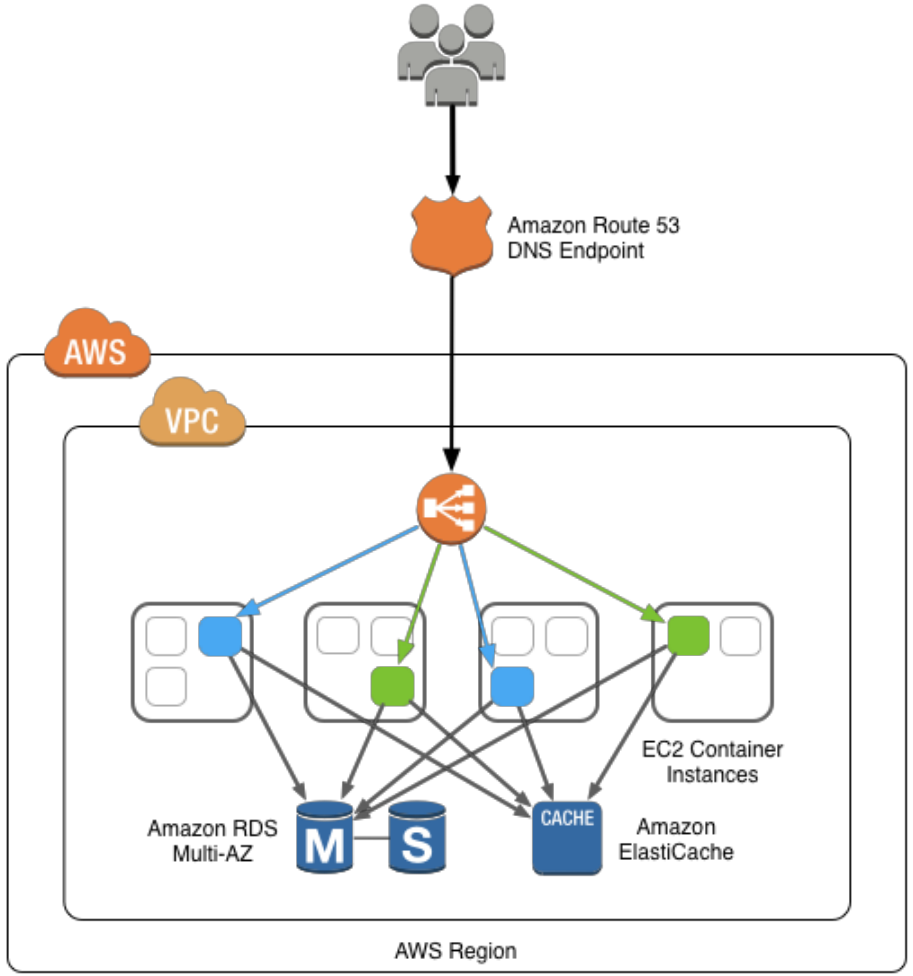
- Start with **blue** service composed of a task definition and ELB
- Create new task definition based on new version of Docker image
- Create **green** service with new task definition and map to existing ELB
- Scale up green service by incrementing number of tasks
- Decommission blue service by setting task count to 0



Pattern: Swap ECS services with ELB

Deployment process:

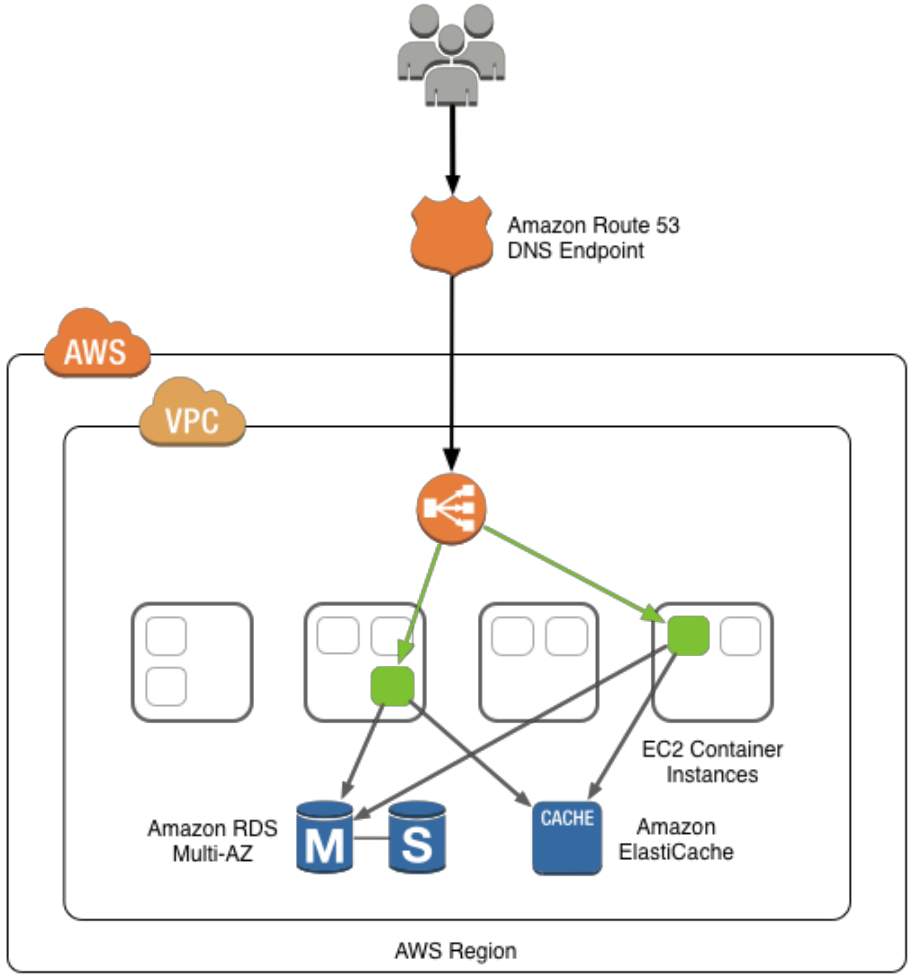
- Start with **blue** service composed of a task definition and ELB
- Create new task definition based on new version of Docker image
- Create **green** service with new task definition and map to existing ELB
- Scale up green service by incrementing number of tasks
- Decommission blue service by setting task count to 0



Pattern: Swap ECS services with ELB

Deployment process:

- Start with **blue** service composed of a task definition and ELB
- Create new task definition based on new version of Docker image
- Create **green** service with new task definition and map to existing ELB
- Scale up green service by incrementing number of tasks
- Decommission blue service by setting task count to 0



Swapping ECS services with ELB

```
$ aws ecs create-service \
--cluster dice-demo \
--service-name web-service-v2 \
--task-definition web-tier \
--load-balancers loadBalancerName=web-
tier,containerName=web-tier,containerPort=8000 \
--desired-count 1 \
--role "ecsServiceRole”
```

Swapping ECS services with ELB

```
$ aws ecs update-service \
--cluster dice-demo \
--service web-service-v2 \
--desired-count 2

$ aws ecs update-service
--cluster dice-demo \
--service web-service-v1 \
--desired-count 0
```

- Similar to swapping ASG behind ELB
- Application versions must co-exist
- Container resources may need to be cleaned up

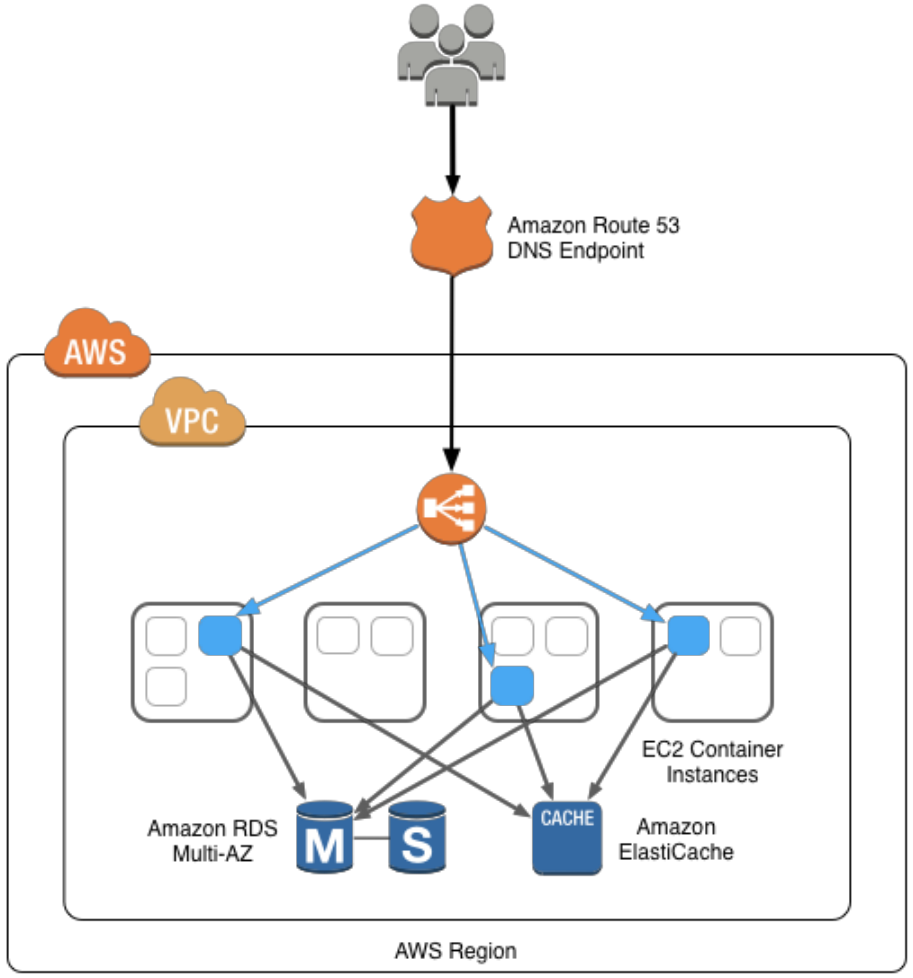
Pattern review: Swap ECS services with ELB

Risk Category	Mitigation Level	Reasoning
Application issues	Great	Canary analysis
Application performance	Great	ELB already warm, CloudWatch and automation to scale ECS
People/process errors	Good	Multi-step process to transition traffic between environments
Infrastructure failure	Good	Leverage CloudWatch, Auto Scaling, ELB
Rollback	Great	No DNS complexities
Cost	Good	Resource management handled by ECS

Pattern: ECS service update

Deployment process:

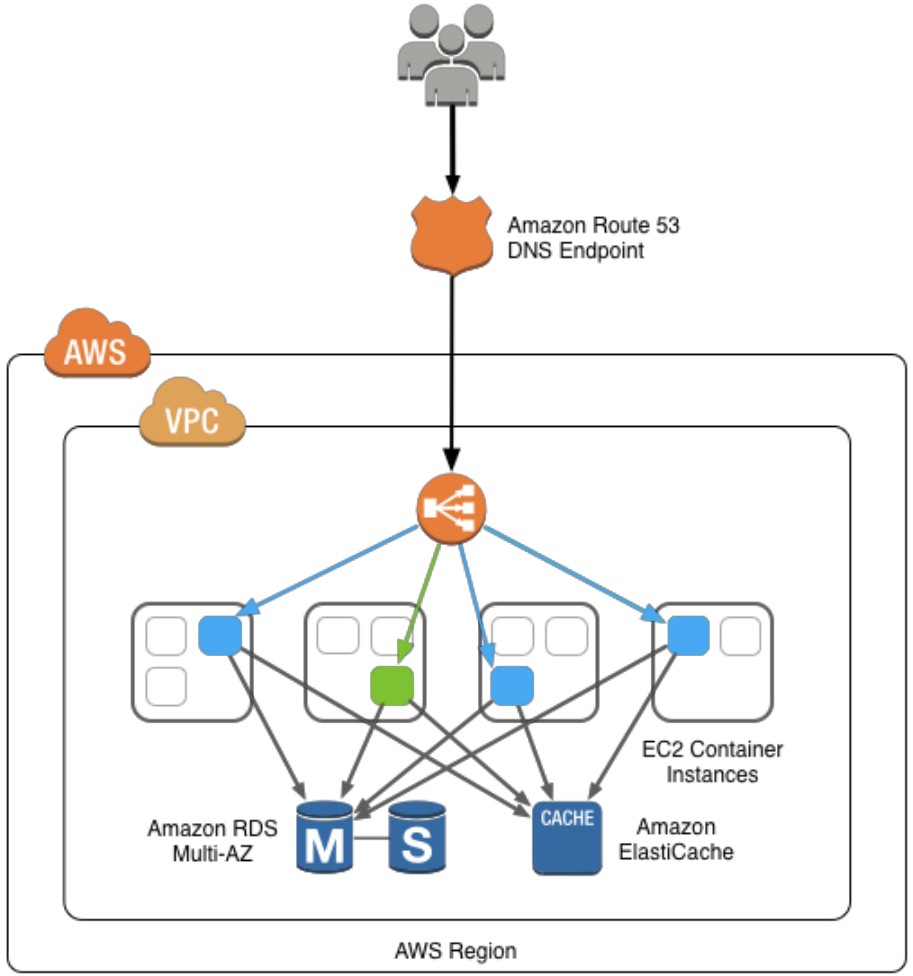
- Start with **blue** task definition referenced by an ECS service
- Create a **green** revision of the existing task definition
- Update existing ECS service to use the updated task definition
- ECS will deploy the new task definition to container instances in a rolling fashion



Pattern: ECS service update

Deployment process:

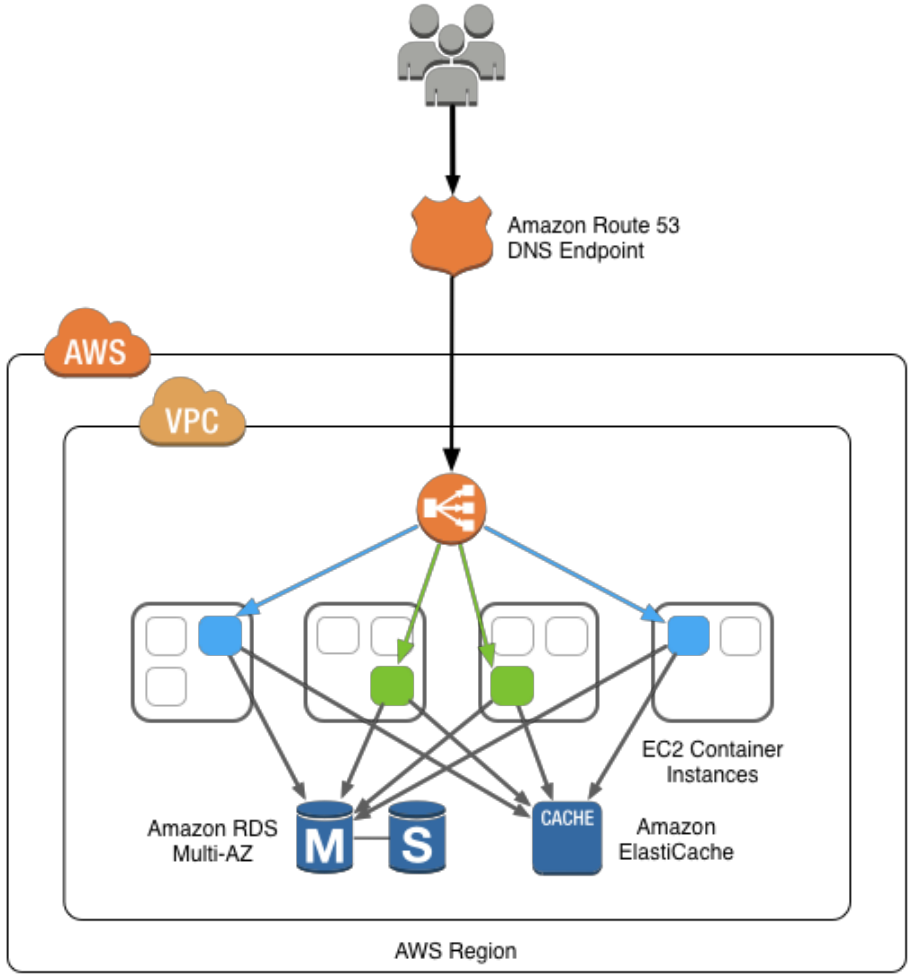
- Start with **blue** task definition referenced by an ECS service
- Create a **green** revision of the existing task definition
- Update existing ECS service to use the updated task definition
- ECS will deploy the new task definition to container instances in a rolling fashion



Pattern: ECS service update

Deployment process:

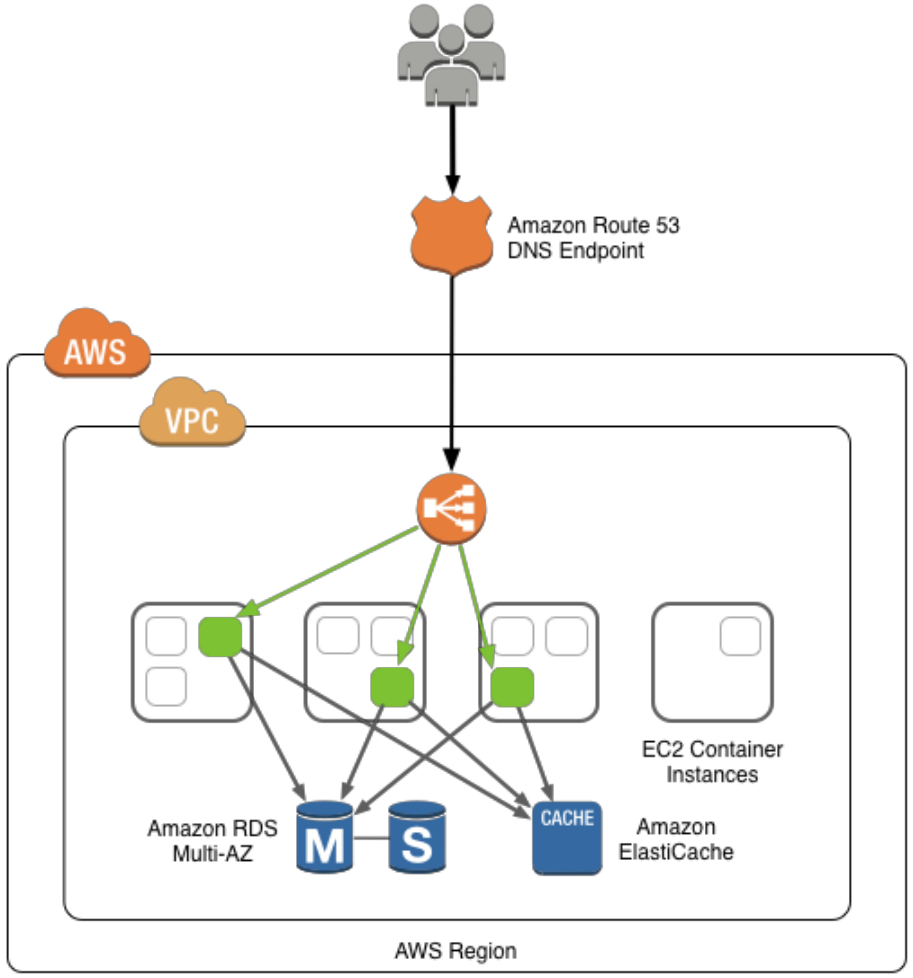
- Start with **blue** task definition referenced by an ECS service
- Create a **green** revision of the existing task definition
- Update existing ECS service to use the updated task definition
- ECS will deploy the new task definition to container instances in a rolling fashion



Pattern: ECS service update

Deployment process:

- Start with **blue** task definition referenced by an ECS service
- Create a **green** revision of the existing task definition
- Update existing ECS service to use the updated task definition
- ECS will deploy the new task definition to container instances in a rolling fashion



ECS service update

1. Create task definition and update service

```
$ aws ecs update-service --cluster dice-demo --service web-service --task-definition
web-tier
```

2. Monitor service update process

```
$ aws ecs describe-services --cluster dice-demo --services web-service

{
    "services": [
        {
            "status": "ACTIVE",
            "taskDefinition": "arn:aws:ecs:us-west-2:012345678901:task-definition/web-
tier:2",
```

ECS service update

“deployments” section of describe-services output displays progress

```
"deployments": [
    {
       "status": "PRIMARY",
       "desiredCount": 3,
       "taskDefinition": "arn:aws:ecs:us-west-2:012345678901:task-definition/web-tier:2",
       "runningCount": 0
    },
    {
       "status": "ACTIVE",
       "desiredCount": 3,
       "taskDefinition": "arn:aws:ecs:us-west-2:012345678901:task-definition/web-tier:1",
       "runningCount": 3
    }
```

ECS service update

“events” section of describe-services output will reveal any errors


```
"events": [
    {
        "message": "(service web-service) has reached a steady state.",
        "id": "34d17afa-89eb-454a-9311-f4de402222ca",
        "createdAt": 1442976067.928
    },
    {
        "message": "(service web-service) registered 1 instances in (elb web-tier)",
        "id": "b0020deb-8c4c-4128-bbd3-9b87487eba8f",
        "createdAt": 1442976061.644
    },
```

Pattern: ECS service update

Risk category	Mitigation level	Reasoning
Application Issues	Fair	No canary analysis
Application Performance	Good	No traffic management, ELB already warm
People/Process Errors	Great	Simple, automated process
Infrastructure Failure	Good	Autoscaling cluster instances and service
Rollback	Great	No DNS complexities
Cost	Great	Rolling deployment, cluster instance +1

What about schema changes?

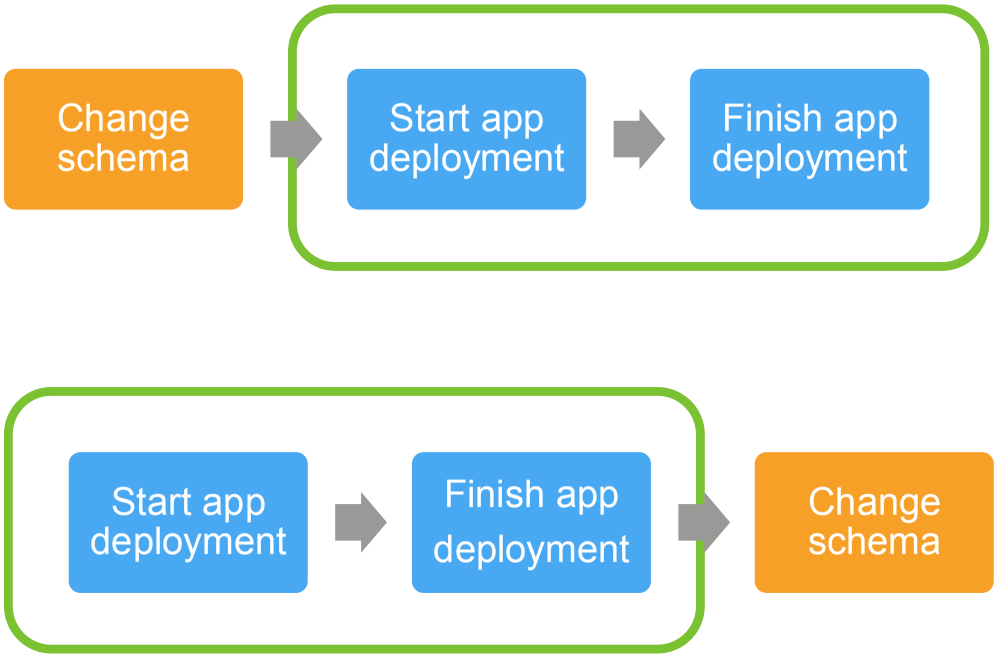
Decoupled schema changes & code changes

Two approaches:

- Database updates are backward-compatible (old code uses new schema)
- Code changes are backward-compatible with the old schema (new code uses old schema)

DB **outside** environment boundary

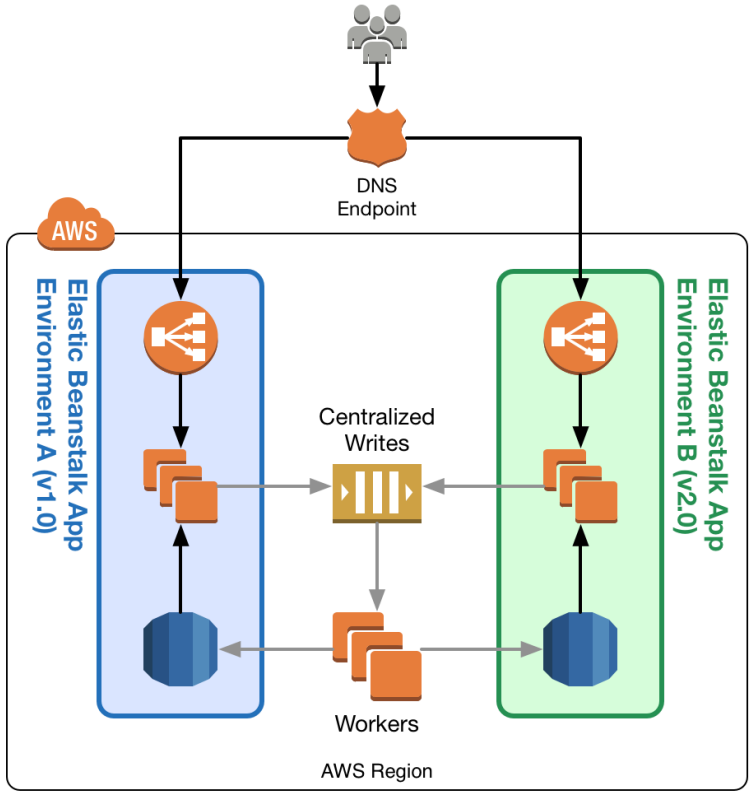
Tradeoff: **simplicity** vs. **risk**



What if schema changes can’t be decoupled, or you’re deploying across regions?

Isolated and synchronized data stores

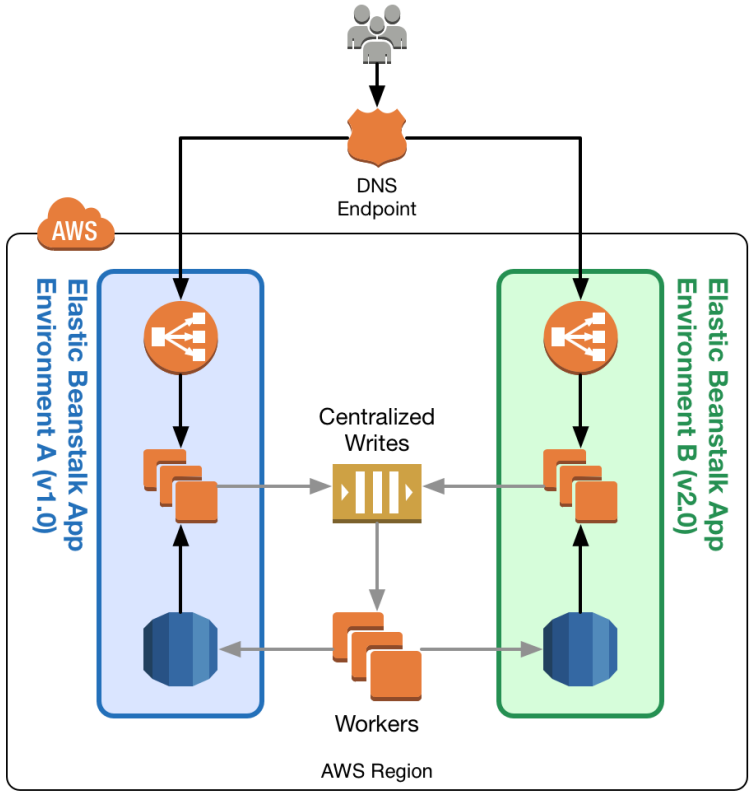
- DB **inside** environment boundary
- Environments have separate data stores
- Need to coordinate data changes across stacks:
  - Blue needs data for rollback
  - Green is new production stack
- What consistency model does the application use?



Centralized writes process

Deployment pattern:

- Deploy an aggregator of **centralized write operations**
- Common baseline of data: Use data store-specific replication
- Aggregator worker affects changes to data in both environments
- Account for lag/latency considerations



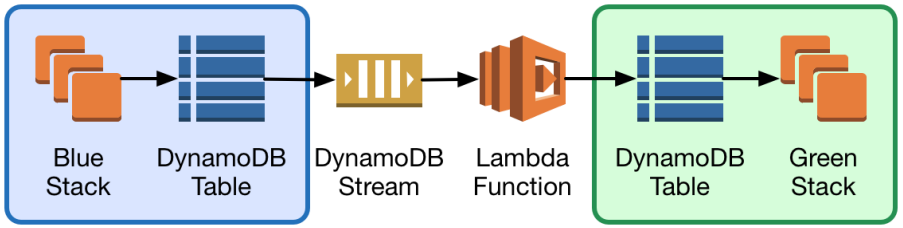
Simplifying the centralized writes process

Green app writes to both DBs

- Asynchronous process moves changes by **blue** app to **green** DB
- Great if there's a mismatch in consistency requirements (old app needs strong consistency)

Each app writes to its own DB

- Asynchronous process pushes changes to the other DB
- Fully decoupled architecture
- Example: DynamoDB tables + streams + triggers + Lambda functions



Closing thoughts

Blue/green deployment patterns at a glance

Pattern Mitigated risk	Classic DNS cutover	Swap Auto Scaling groups	Swap launch configs	Swap ECS services with DNS	Swap ECS services with ELB	ECS service update
Application issues	Canary analysis	Canary analysis	Mixed fleet	Canary analysis	Canary analysis	Mixed fleet
Application performance	Granular traffic switch	Instance-level granularity	Mixed fleet	ELB may require pre-warm	Container level granularity, warm ELB	No traffic management, warm ELB
People/process errors	Automation: Use CloudFormation with Elastic Beanstalk, OpsWork, third party			Simple process DNS swap	Multi-step process	ECS automated
Infrastructure failure	Automation framework	Auto Scaling, ELB	Auto Scaling, ELB	CloudWatch, Auto Scaling, ELB	CloudWatch, Auto Scaling, ELB	CloudWatch, Auto Scaling, ELB
Rollback capability	DNS	ELB	ELB	DNS	ECS automated	ECS automated
Cost management	Gradual scaling	Gradual scaling	Some over-provisioning	Require addl cluster instances	Resource reuse	Rolling deployment
Deployment complexity	Simple, DNS weights	Auto Scaling control	Scale-in adjustments	Simple, DNS weights	Multi-step process	Highly automated

Get comfortable with the deployment process

Deployments will always have risks associated with them

- Deployment & automation frameworks help mitigate process & human error risks
- Get comfortable with deployments, practice using production-like replicas
- AWS provides affordable, quickly provisioned resources – use the flexibility to experiment new approached

Thank you!

Remember to complete your evaluations!

Related Sessions

DVO202 - DevOps at Amazon: A Look at Our Tools and Processes

DVO305 - Turbocharge Your Continuous Deployment Pipeline with Containers