

CON402

**AWS**  
**re:Invent**

## Advanced Patterns in Microservices Implementation with Amazon ECS

**AWS**  
**re:Invent**

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Scaling a microservice-based infrastructure can be challenging in terms of both technical implementation and developer workflow. In this talk, AWS Solutions Architect Pierre Steckmeyer will be joined by Will McCutchen, Architect at BuzzFeed, to discuss Amazon ECS as a platform for building a robust infrastructure for microservices. We will look at the key attributes of microservice architectures and how Amazon ECS supports these requirements in production, from configuration to sophisticated workload scheduling to networking capabilities to resource optimization. We will also examine what it takes to build an end-to-end platform on top of the wider AWS ecosystem, and what it's like to migrate a large engineering organization from a monolithic approach to microservices.

### What to Expect from this Session

- Microservices Architecture
- Amazon ECS
- The Twelve-Factor App with Amazon ECS
- Task Placement
- How BuzzFeed built a platform on ECS

## Microservices Architecture

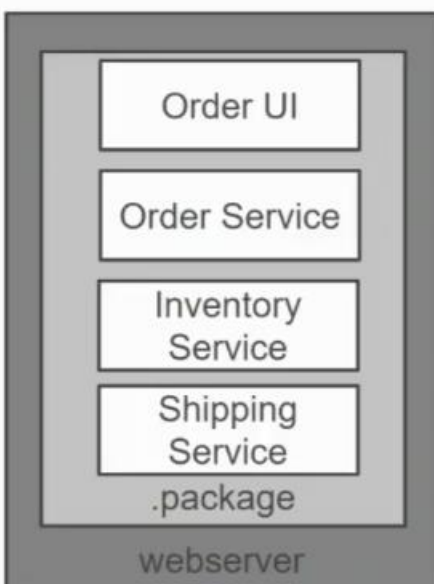
# What are Microservices?

Microservices are an **architectural and organizational** approach to software development in which software is composed of **small, independent services** that communicate over **well-defined APIs**. These services are owned by **small, self-contained** teams.

Watch the replay: CON208

Whitepaper: <http://bit.ly/2A0qGdt> - Running Containerized Microservices on AWS

## Monolithic vs. Microservices

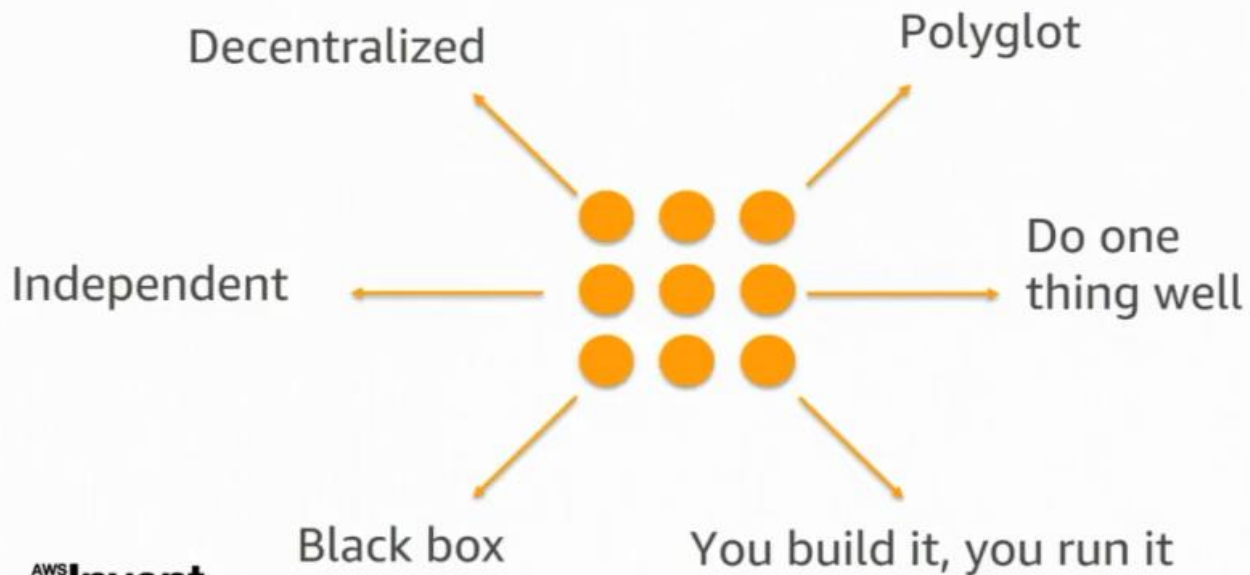


**AWS**  
**re:Invent**

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



# Characteristics

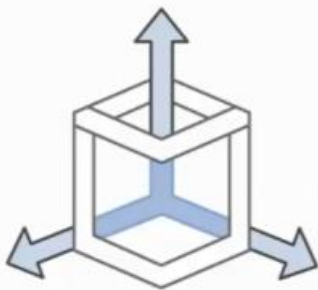


**AWS re:Invent**

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



## Why Amazon ECS



- Fully managed elastic service – You don't need to run anything, and the service scales as your microservices architecture grows
- Shared state optimistic scheduling
- Integration with Amazon CloudWatch service for monitoring and logging
- Integration with Code\* services for continuous integration and delivery (CI/CD)

## Deploying Containers on ECS – Choose a Scheduler

### Batch Jobs

ECS task scheduler

Run tasks once

Batch jobs

RunTask (random)

StartTask (placed)

### Long-Running Apps

ECS service scheduler

Health management

Scale-up and scale-down

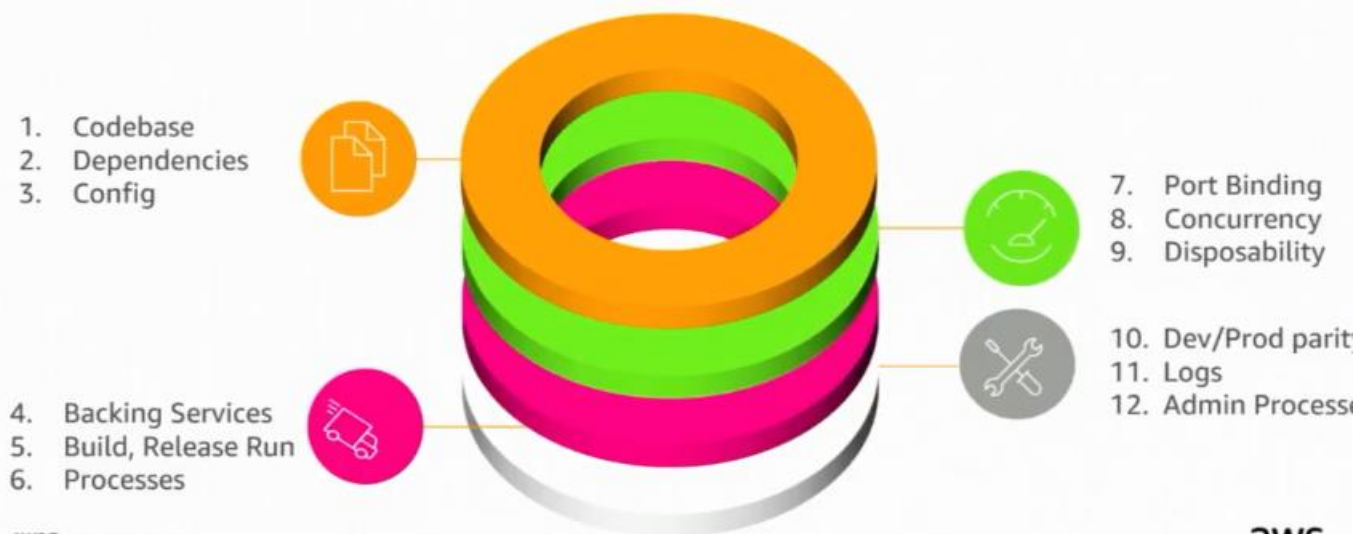
AZ aware

Grouped containers

Reference Architectures

## The Twelve-Factor App with Amazon ECS

## The Twelve-Factor App



**AWS re:Invent**

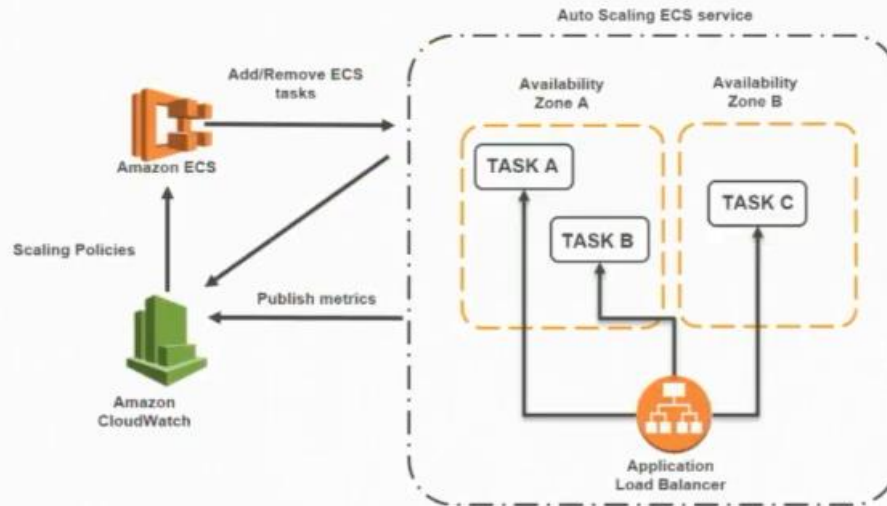
© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

<https://12factor.net/>





# Automatic Service Scaling



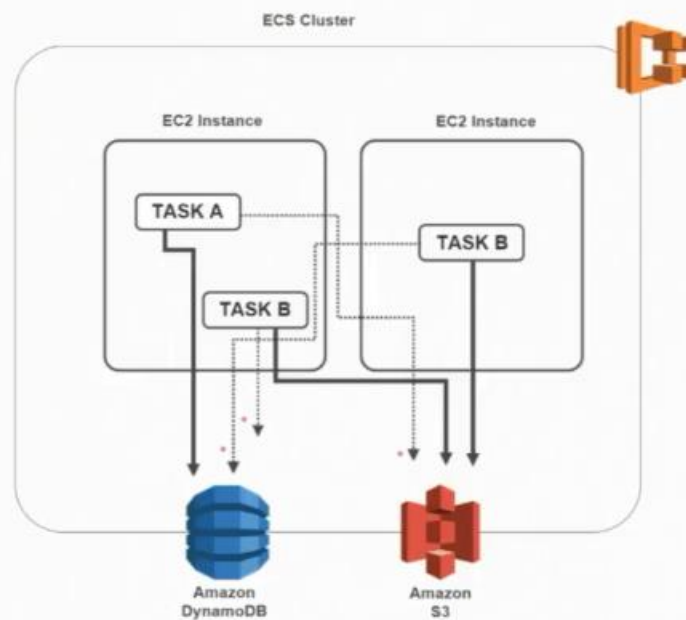
AWS  
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



ECS recommends that you scale across 2 dimensions. The 1<sup>st</sup> dimension is at the app level or how many tasks are you running? And at the cluster level, how many instances are you running within it.

# IAM Roles for Tasks



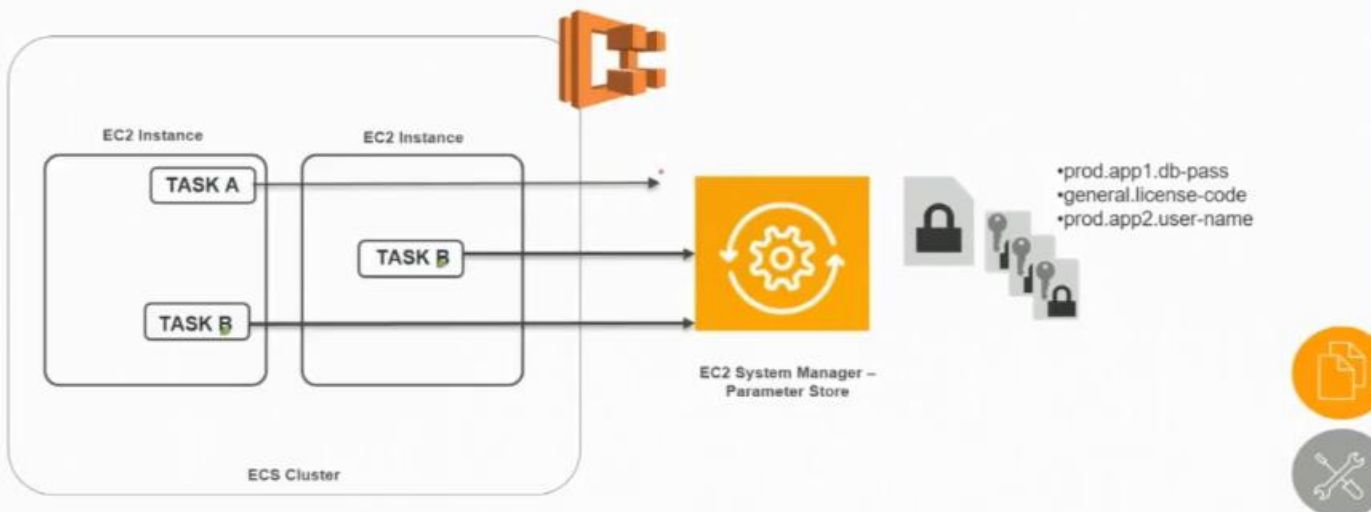
AWS  
re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



You can now give IAM roles to your tasks, this means that you can have a task that is allowed to read from S3 only and another task that is allowed to read and write to DynamoDB.

# Secrets Management



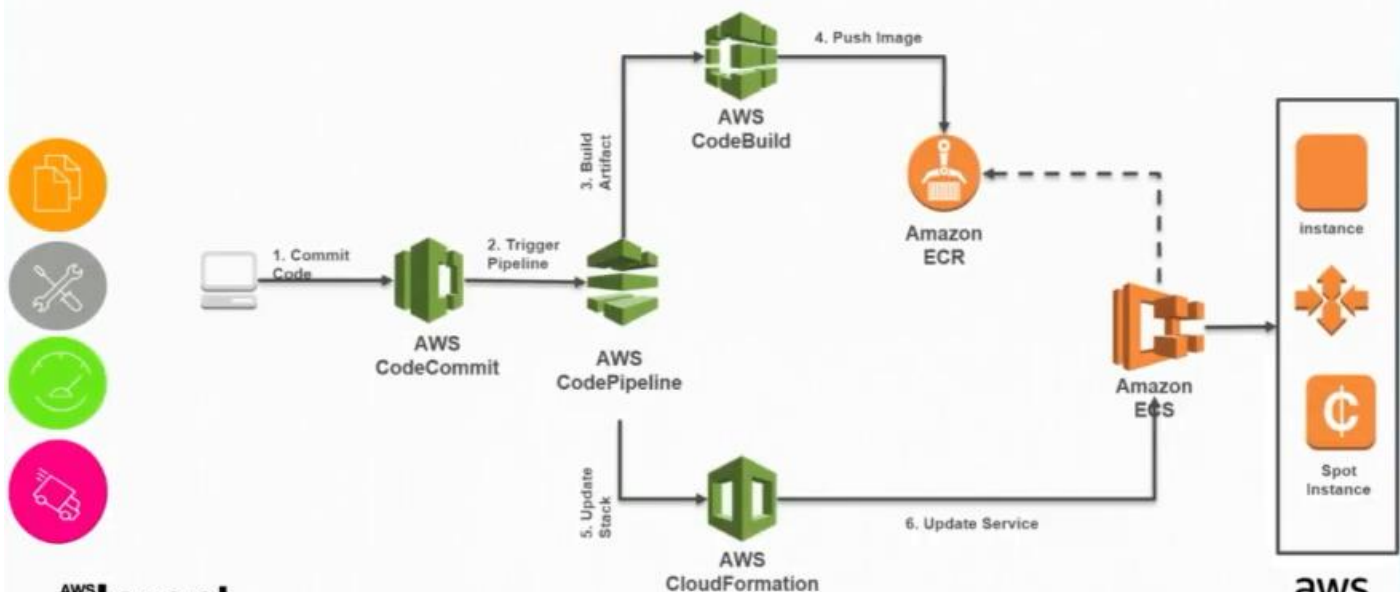
AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

You can encrypt the password you pass to your task and give the task an IAM role that has the key to decrypt the password using the system manager

# Continuous Deployment

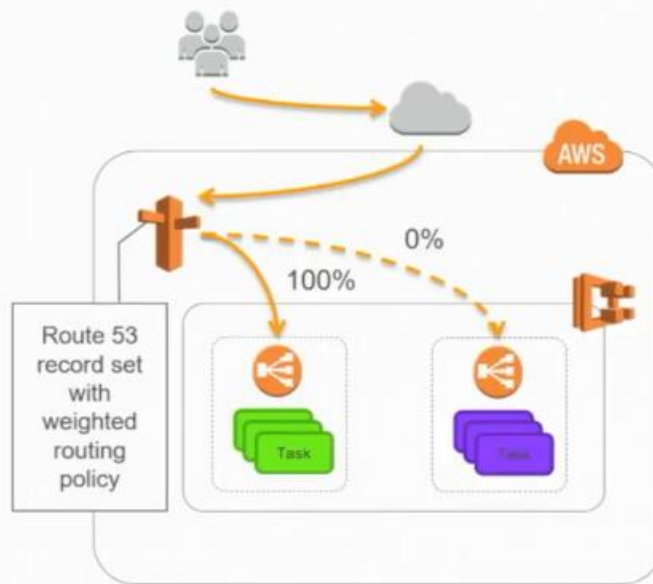


AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

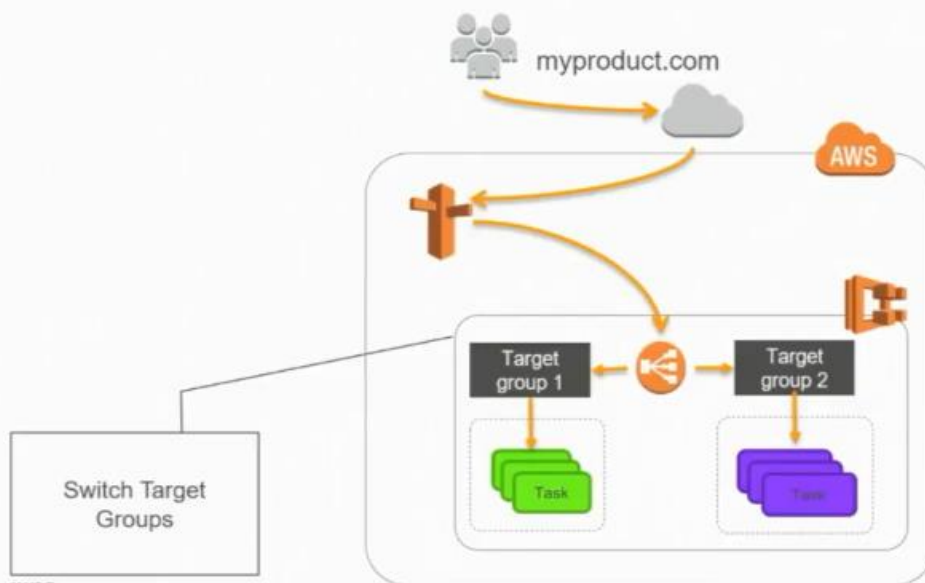
# Blue-Green Deployments (DNS-based)



**AWS re:Invent**

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

# Blue-Green Deployments (Target Group Switch)



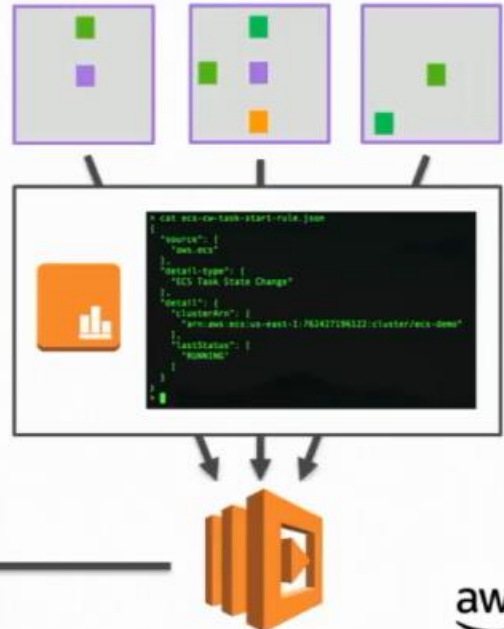
**AWS re:Invent**

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

# Consuming Events for Service Discovery

my-app → 10.1.0.20  
db-dev → 10.1.0.19  
websrv1 → 10.1.0.1  
websrv2 → 10.1.0.2  
websrv3 → 10.1.0.4  
app-dev1 → 10.1.0.9  
app-dev2 → 10.1.0.5  
app-dev3 → 10.1.0.8

```
def lambda_handler(event, context):  
    container_instance = ecs.describe_container_instances(  
        cluster=event['detail']['clusterArn'],  
        containerInstances=  
            event['detail']['containerInstanceArns'],  
    )['containerInstances'][0]  
  
    ec2_instance = ec2.describe_instances(  
        instanceIds=[container_instance['ec2InstanceId']],  
    )['Reservations'][0]['Instances'][0]  
  
    private_dns = ec2_instance['PrivateDnsName']  
    container_name = event['detail']['containers'][0]['name']  
  
    r53.change_resource_record_sets(  
        HostedZoneId=HostedZoneId,  
        ChangeBatch={  
            'Changes': [  
                {  
                    'Action': 'UPSERT',  
                    'ResourceRecordSet': {  
                        'Name': '{},{},'.format(container_name, HostedZoneName),  
                        'Type': 'SRV',  
                        'TTL': 60,  
                        'ResourceRecords': [  
                            {  
                                'Value': '1 {} {} {}'.format(0, private_dns,
```



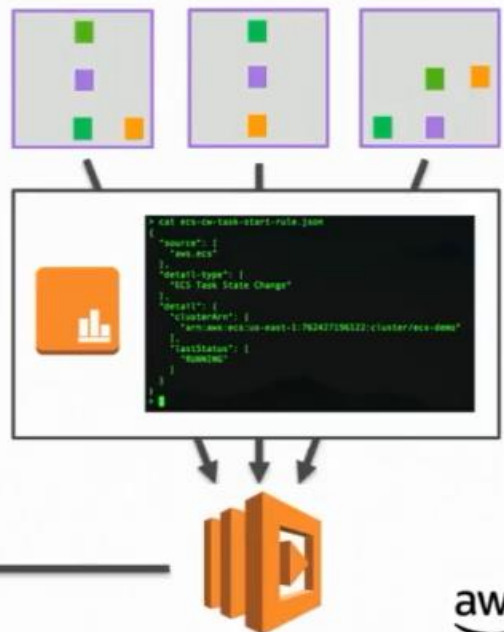
© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

For a DIY way to do service discovery, we can use the event stream that comes out of ECS

# Consuming Events for Service Discovery

app1-tst → 10.1.0.11  
db1-tst → 10.1.0.14  
app2 → 10.1.0.16  
db2 → 10.1.0.18  
  
db-dev → 10.1.0.19  
websrv1 → 10.1.0.1  
websrv2 → 10.1.0.2  
websrv3 → 10.1.0.4  
app-dev1 → 10.1.0.9  
app-dev2 → 10.1.0.5  
app-dev3 → 10.1.0.8

```
def lambda_handler(event, context):  
    container_instance = ecs.describe_container_instances(  
        cluster=event['detail']['clusterArn'],  
        containerInstances=  
            event['detail']['containerInstanceArns'],  
    )['containerInstances'][0]  
  
    ec2_instance = ec2.describe_instances(  
        instanceIds=[container_instance['ec2InstanceId']],  
    )['Reservations'][0]['Instances'][0]  
  
    private_dns = ec2_instance['PrivateDnsName']  
    container_name = event['detail']['containers'][0]['name']  
  
    r53.change_resource_record_sets(  
        HostedZoneId=HostedZoneId,  
        ChangeBatch={  
            'Changes': [  
                {  
                    'Action': 'UPSERT',  
                    'ResourceRecordSet': {  
                        'Name': '{},{},'.format(container_name, HostedZoneName),  
                        'Type': 'SRV',  
                        'TTL': 60,  
                        'ResourceRecords': [  
                            {  
                                'Value': '1 {} {} {}'.format(0, private_dns,
```

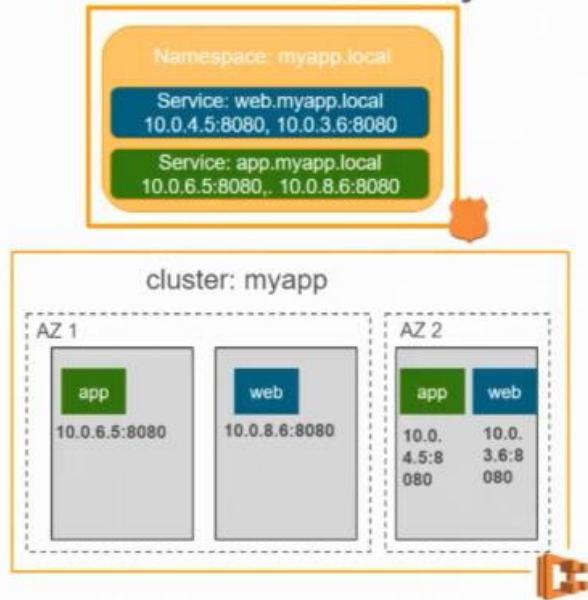


© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

We can use lambda and route53 hosted zones as things happen



# Service Discovery with Route 53



- Client side Service Discovery
- Service Registry = Route53 DNS Server
- Registry update done by ECS scheduler
- Managed, high availability, high scale, Extensible

More at breakout session  
CON403 Friday 10 AM



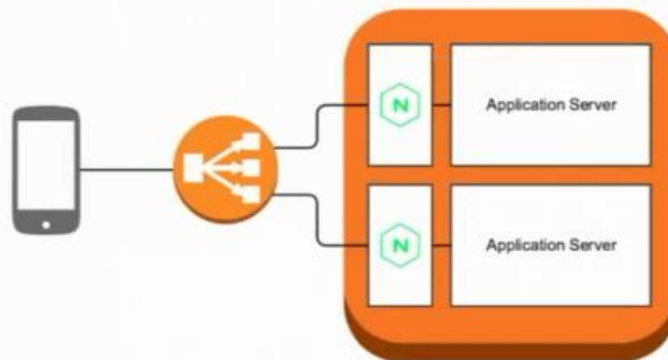
**AWS re:Invent**

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Route53 is now going to start supporting service discovery, you just have to tick a box when starting up ECS and Route53 will start taking note of the instances coming up and going

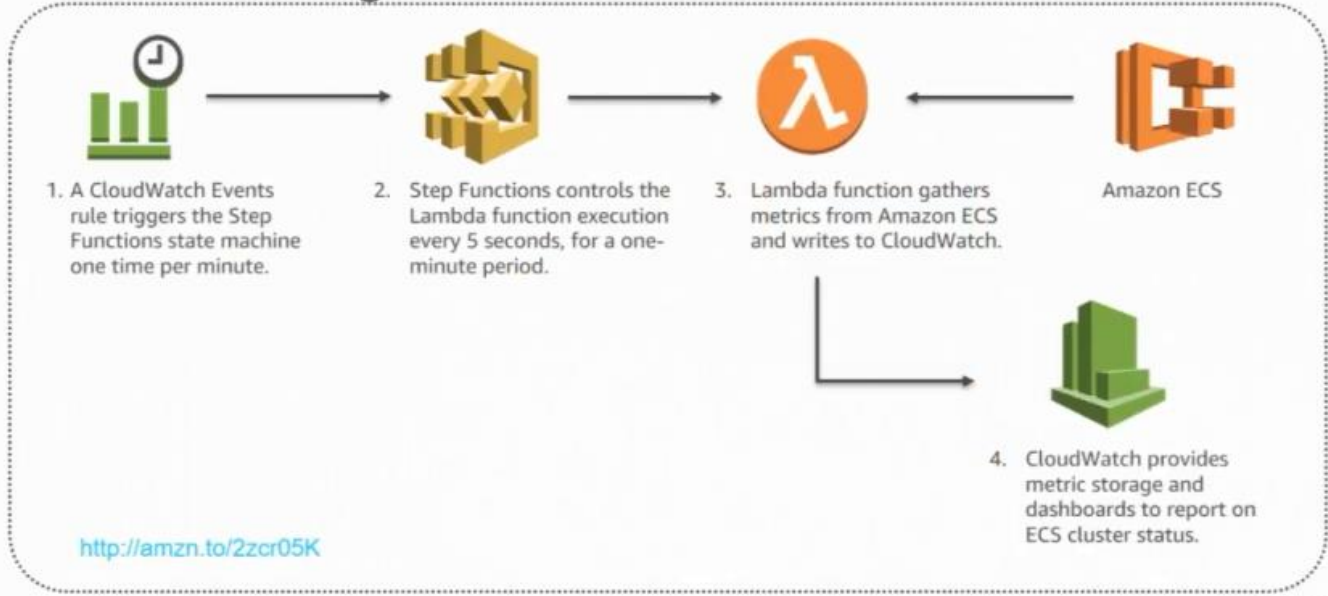
# Nginx reverse proxy



<http://amzn.to/2neahgV>

Nginx can provide security by filtering requests that you want to allow and reject all other requests, you can also use Nginx for gzip compression

# Custom, High-Resolution Metrics



**aws re:Invent**

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



## Task Placement Examples

### Placement: Targeting Instance Type

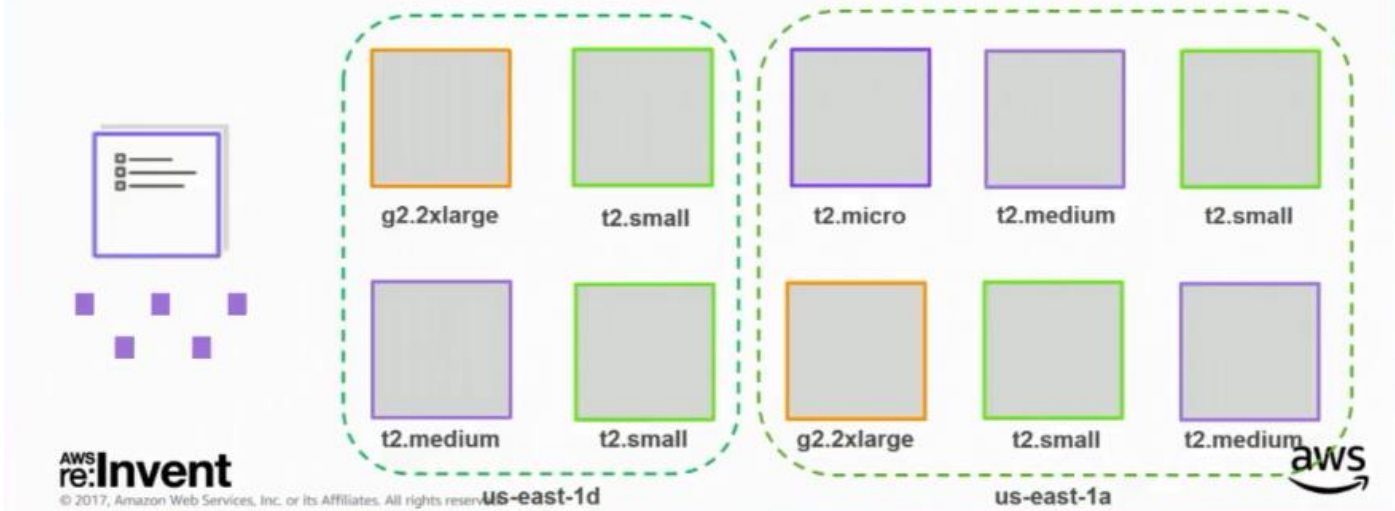
```
aws ecs run-task --cluster ecs-demo --task-definition myapp --count 5 --placement-constraints type="memberOf",expression="attribute:ecs.instance-type == g2.2xlarge"
```



You can indicate that you want your task to land on some specific instance type

## Placement: Targeting Instance Type & Zone

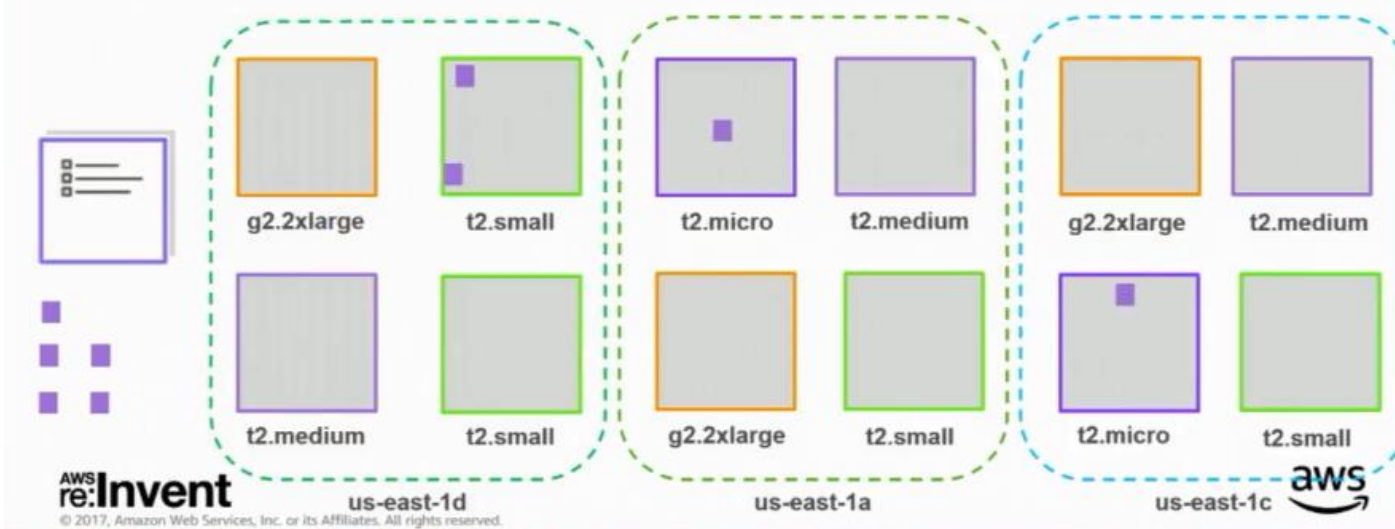
```
aws ecs run-task --cluster ecs-demo --task-definition myapp --count 5 --placement-constraints
type="memberOf",expression="(attribute:ecs.instance-type == t2.small or
attribute:ecs.instance-type == t2.medium) and attribute:ecs.availability-zone != us-east-1d"
```



You can also specify you want your container to land on certain AZ

## Placement: Spread across Zone and Binpack

```
aws ecs run-task --cluster ecs-demo --task-definition myapp --count 9 --placement-strategy
type="spread",field="attribute:ecs.availability-zone" type="binpack",field="memory"
```

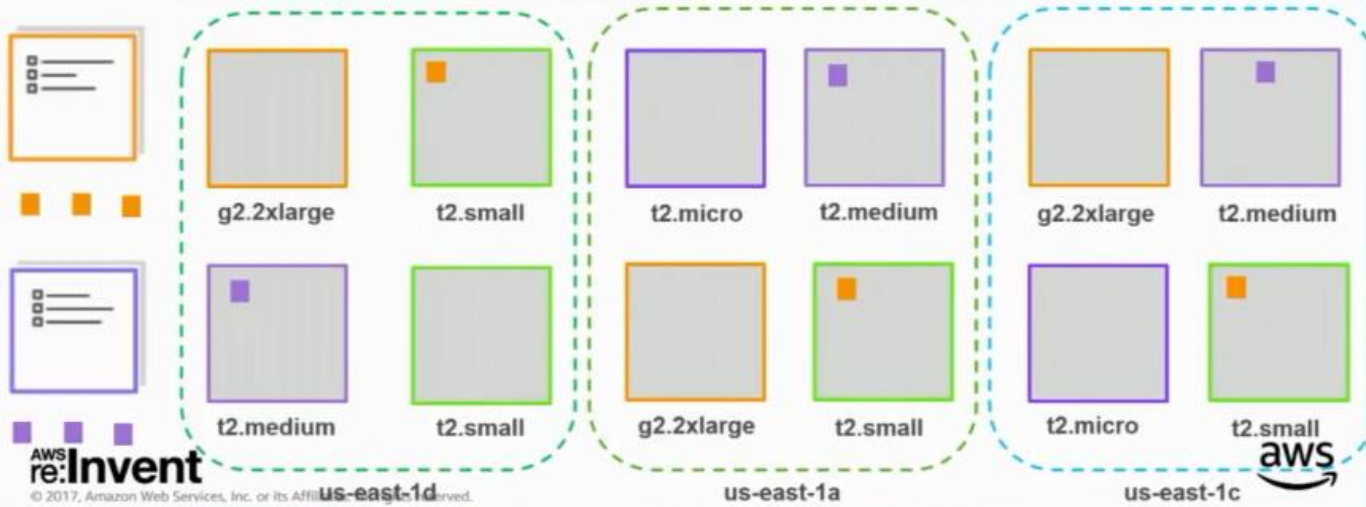


You can also spread your workload on different AZ and also do bin packing

# Placement: Affinity and Anti-Affinity

```
aws ecs run-task --count 3 --cluster ecs-demo --task-definition myapp --group webserver --placement-constraints type=memberOf,expression="task:group == webserver"
```

```
aws ecs run-task --count 3 --cluster ecs-demo --task-definition mydb --group webserver --placement-constraints type=memberOf,expression="not=(task:group == webserver)"
```



You can also specify if you want tasks to be close to one another or not

## Running a Service

```
{
  "cluster": "ecs-demo",
  "serviceName": "my-service",
  "taskDefinition": "my-app",
  "desiredCount": 10,
  "placementConstraints": [
    {
      "type": "memberOf",
      "expression": "attribute:ecs.instance-type matches t2.*"
    }
  ],
  "placementStrategy": [
    {
      "type": "spread",
      "field": "attribute:ecs.availability-zone"
    },
    {
      "type": "binpack",
      "field": "MEMORY"
    }
  ]
}
```

AWS re:Invent

© 2017, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

aws

You can do this at runtime or just embed it within your service definition file as above



**BuzzFeed**

**Building a  
platform  
on ECS**

**BuzzFeed**

**Will McCutchen**

Platform Infrastructure

[will.mccutchen@buzzfeed.com](mailto:will.mccutchen@buzzfeed.com)  
[twitter.com/mccutchen](https://twitter.com/mccutchen)

 **README.md**

# rig

v. Set up (equipment or a device or structure), typically hastily or in makeshift fashion

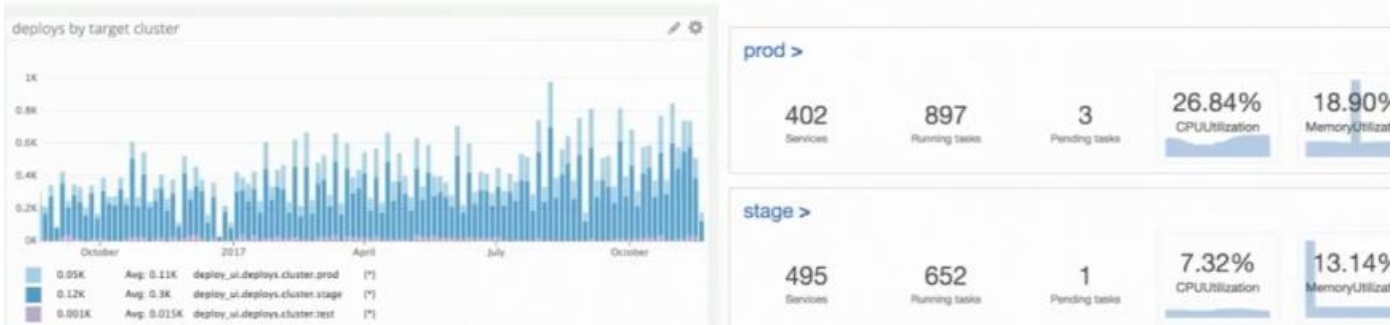
`rig` is a complete platform for containerized services on Amazon ECS.

[View all of README.md](#)



## Where are we now?

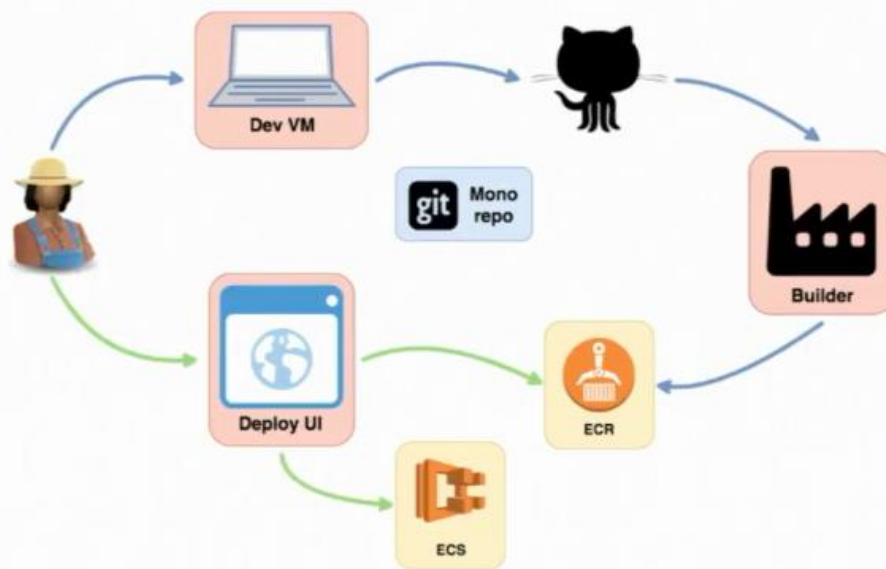
- ~20 months in production
- ~200 users
- > 50,000 deploys
- ~400 services currently deployed
- ~80 container instances
- 6 clusters
- 2 regions



## How did we get here?

## What we learned

## Make your development & deployment workflow as frictionless as possible



All rig services are in a single monolithic Git repo, builder gets the code and builds the Docker container and pushes it to ECR, then the user gets to provide 3 parameters, the service name, version, and the target cluster name, the system then goes ahead and deploys the code to ECS so that the user can see and use the app in the target cluster environment.

## Target abstractions, force consistency

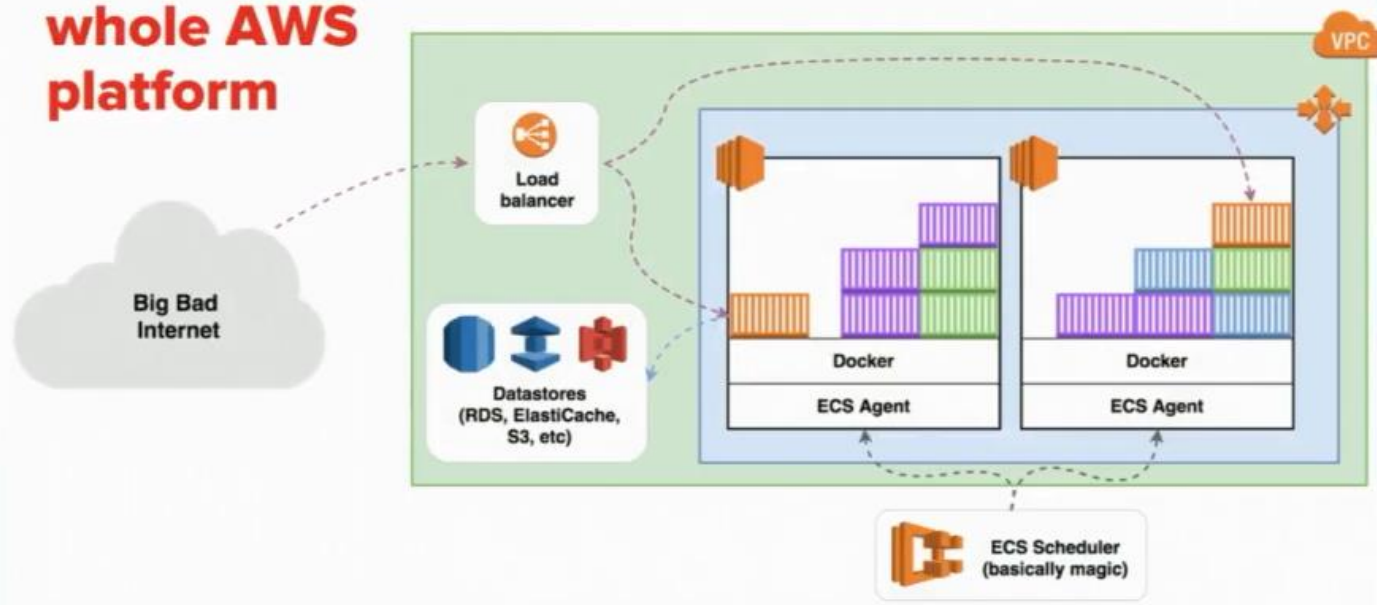
```
service.yml
1 ---
2 instances:
3   test: 1
4   stage: 2
5   prod: 3
6   cpu: 256
7   mem: 512
8   listeners:
9     - subdomain: deploy-internal
10       port: 40000
11       protocol: http-internal
12   monitoring:
13     team: platform-infra
14     slack_channel:
15     alerts: platform-infra-alerts
16     deploys: deploys
17
```

```
config.yml
1 ---
2 default:
3   debug: false
4   working_dir: /tmp/git
5   canary_services: []
6
7 dev:
8   debug: true
9
10 prod:
11   canary_services:
12     - api_gateway
13     - httpbin
14
```

```
$ papertrail -f 'program:prod/httpbin/'
Aug 09 01:27:49 rig-prod-ecs-diagnostics-abrasion prod/httpbin/67c9cbb: 2017/0
Aug 09 01:27:49 rig-prod-ecs-exchange-manufacturer prod/httpbin/67c9cbb: 2017/0
Aug 09 01:27:57 rig-prod-ecs-exchange-manufacturer prod/httpbin/67c9cbb: 2017/0
```

We have developed abstractions that are being enforced generally, like a rig service is a collection of the service definition that defines the structure of the service in a service.yml file, a configuration can also be created using the base config.yml file shown above

## Leverage the whole AWS platform



This is what a rig cluster looks like, we build immutable AMIs for all the ASGs that we create for the ECS services we run based on CloudWatch metrics. Every service exposes a network interface that uses a LB and a deterministic Route53 route

## Make everything self-service

The screenshot shows a self-service UI for rig deployment. It consists of three overlapping panels. The leftmost panel is titled "rig credentials" and contains a section for "account info" with a "username" field showing "will.mccutchen@buzzf". The middle panel is titled "rig data store" and contains a "service" dropdown menu with "httpbin" selected. The rightmost panel is titled "rig deploy" and contains a "service" dropdown menu with "httpbin" selected, a "version" dropdown menu with "master / 858a" selected, and a checkbox labeled "deploy a canary". Below the "service" dropdown is a link "view current deployments". Below the "version" dropdown is a link "commit 858a802 (show di)" and a timestamp "5 days ago".

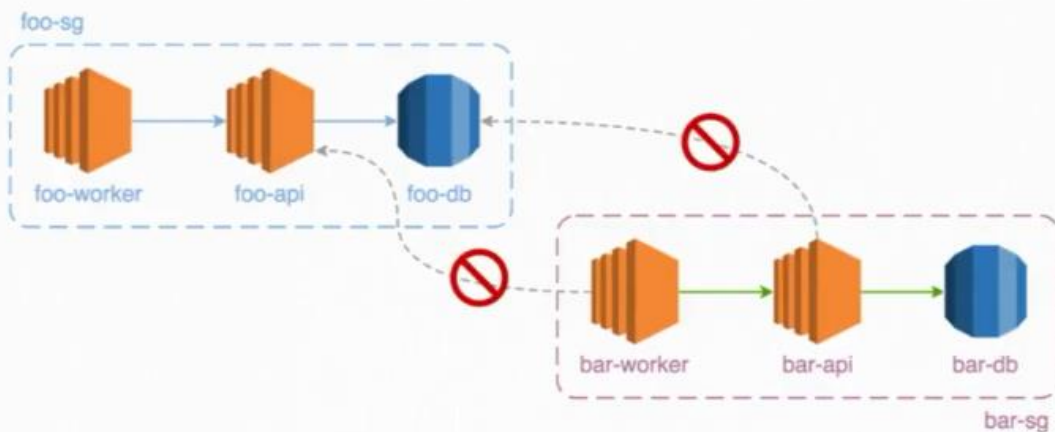
We also created self service UI apps that allows engineers to deploy their apps, configure datastores, credentials for user onboarding.

# Some challenges

## Network-level access control and isolation

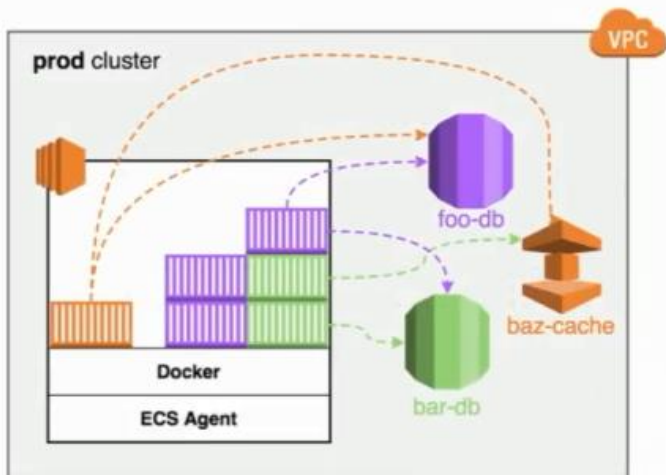
SECURITY

## The good old days

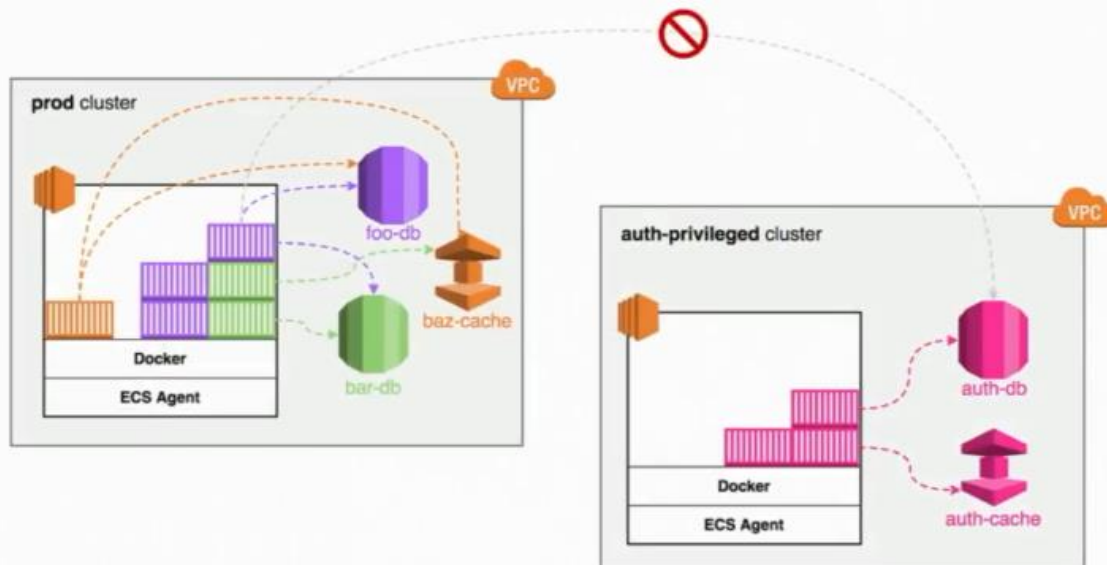




## On ECS, everything runs everywhere

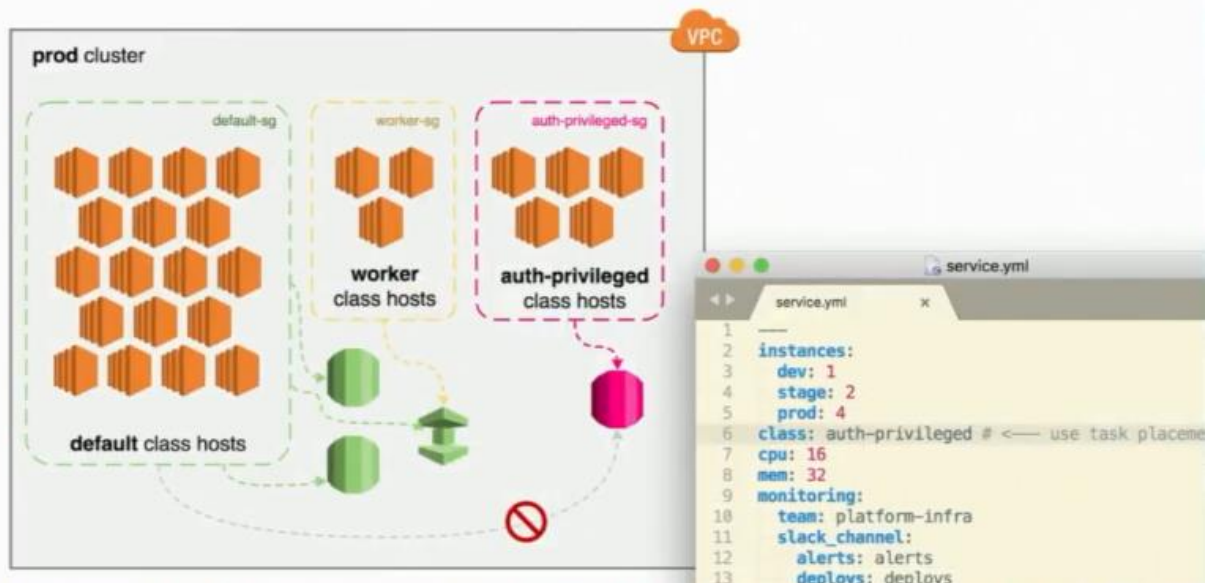


## Quick fix: Isolated ECS clusters & VPCs





## Better fix: task placement & class host groups

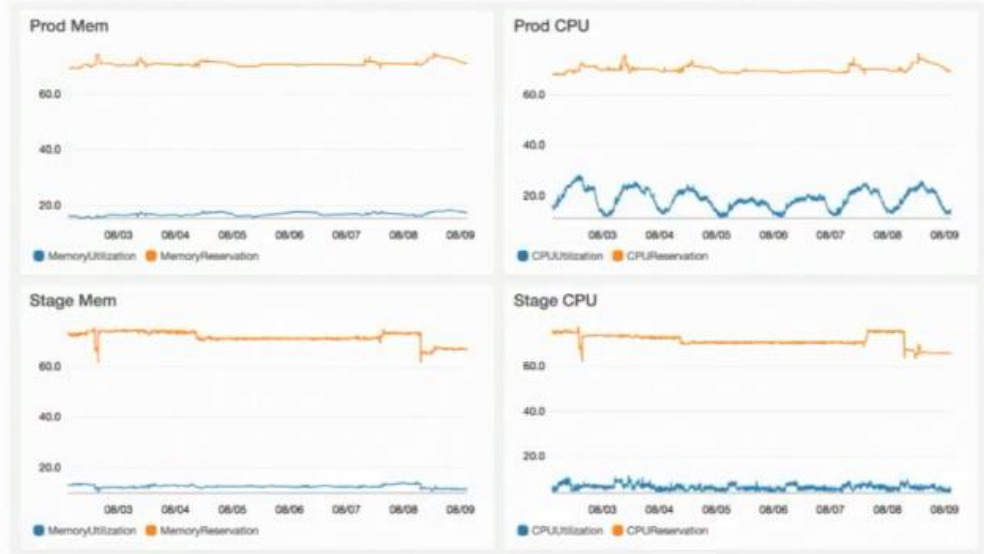


## Quickly & safely rolling clusters

We have a lambda function that listens to the ECS cluster for things going on

## Sharing ECR registries

# Efficiency



## What's next?