OLIVEYOUNG

ECS Auto Scaling/배포 전략

안성진 SA

Amazon Web Services

Agenda

ECS Auto Scaling

- Service Auto Scaling
- Cluster Auto Scaling

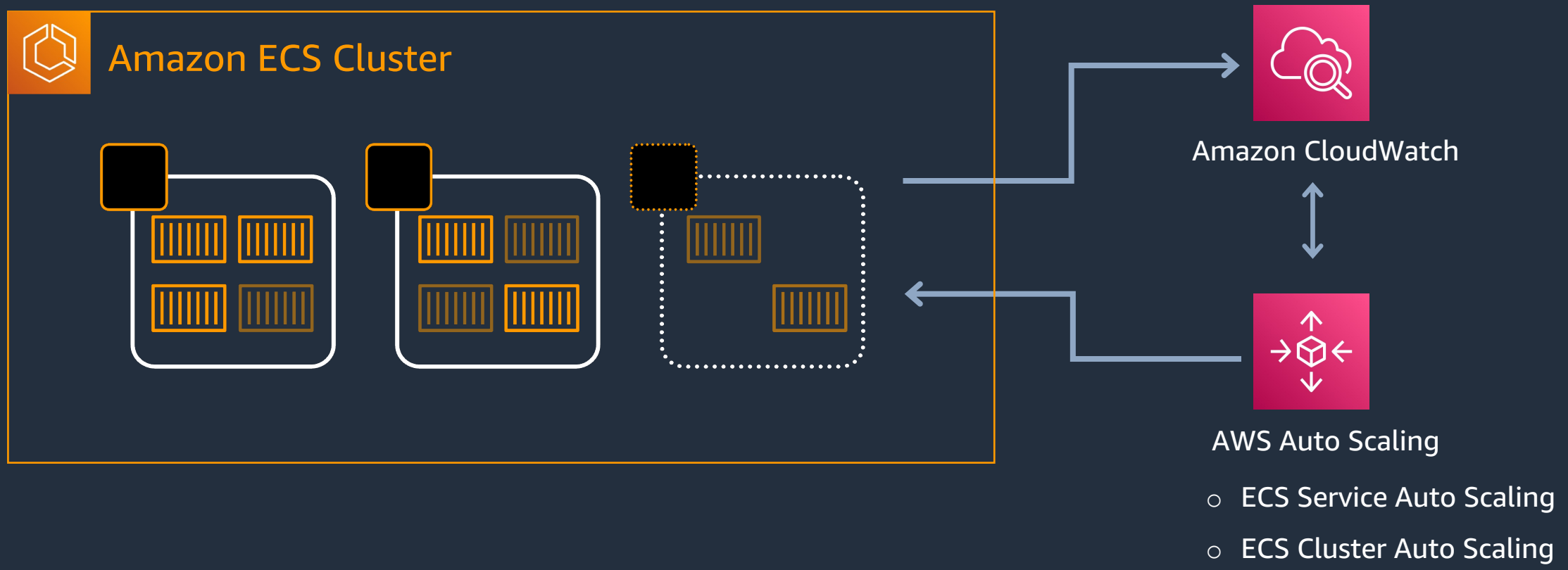
Capacity Provider Demo

+ Spot Draining tip

Deployment

Amazon ECS Auto Scaling

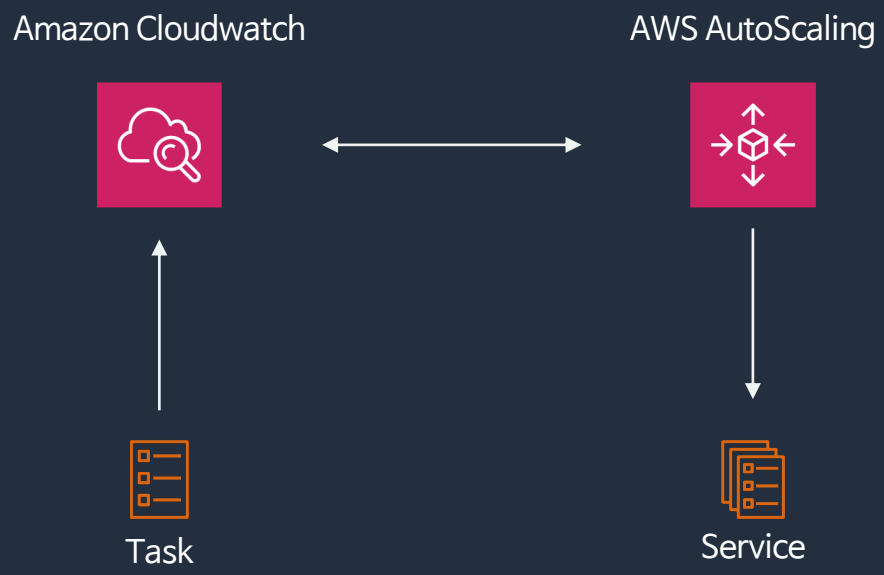
Auto Scaling: 성능, 안정성 및 비용 최적화



Auto Scaling

서비스 내의 작업 수를 자동으로 늘리거나 줄이는 역할을 함

ECS는 CPU 및 메모리 통계를 CloudWatch에 게시하고,
Target Tracking, Step Scaling, Scheduled Scaling 지원

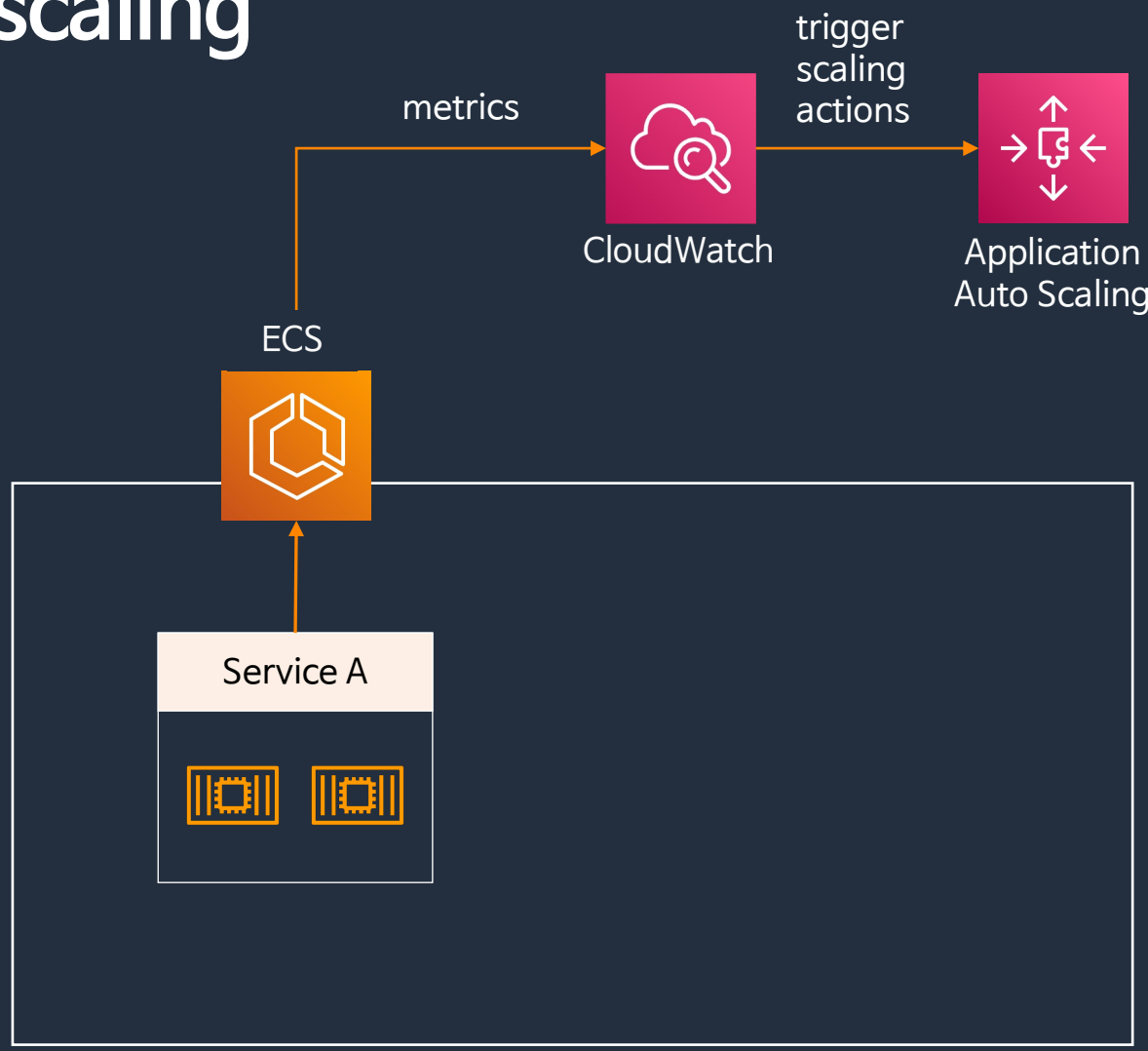


AWS에서는 CPU 메트릭 기반의 Target Tracking 사용을 권고

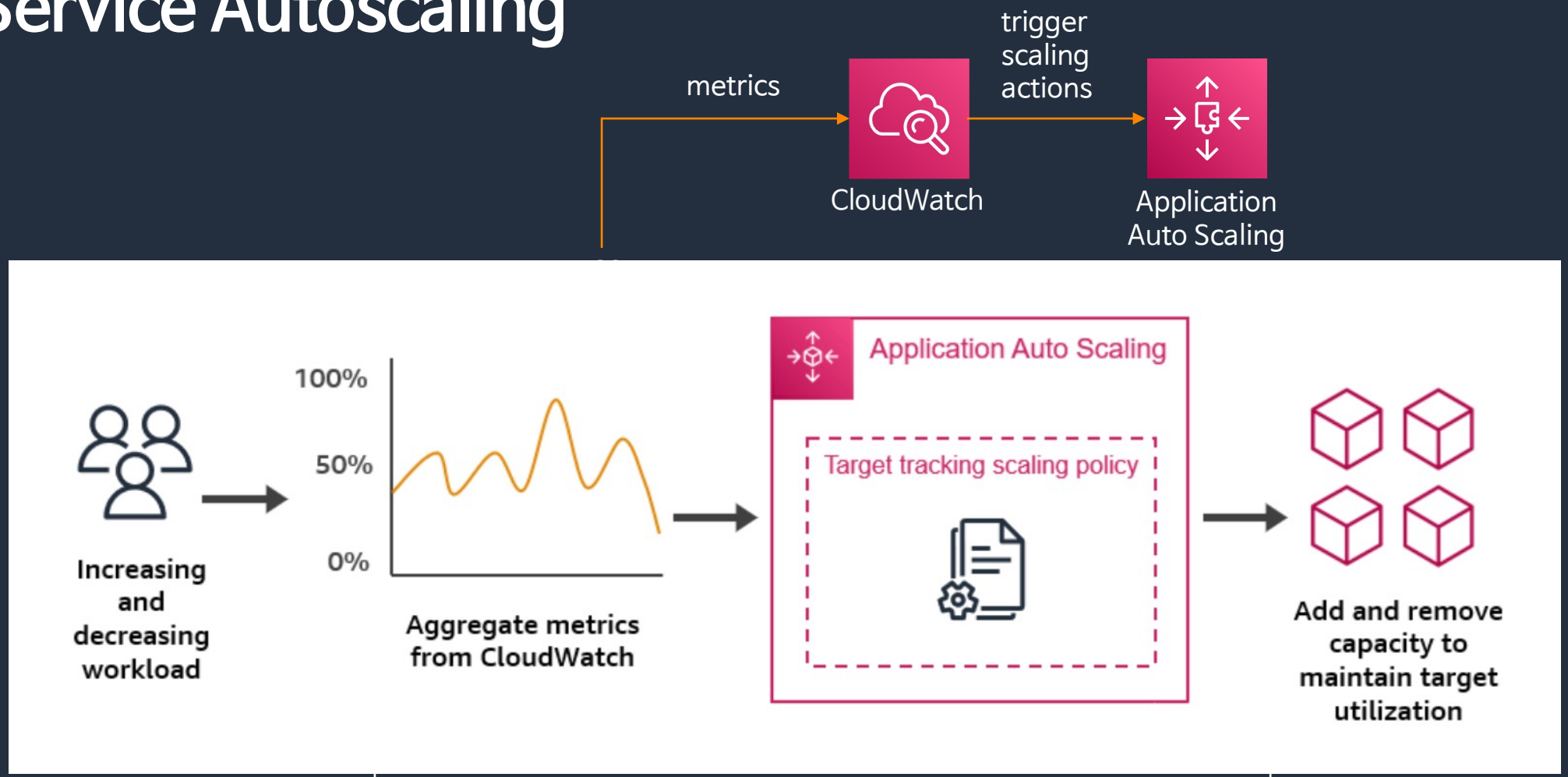
Target Tracking은 메트릭을 반올림하여 확장을 원활하게 수행.

Task는 CloudWatch로 데이터를 전송하고, CloudWatch는 서비스 작업 수를 늘리거나 줄이기 위해 AutoScaling을 트리거시킴

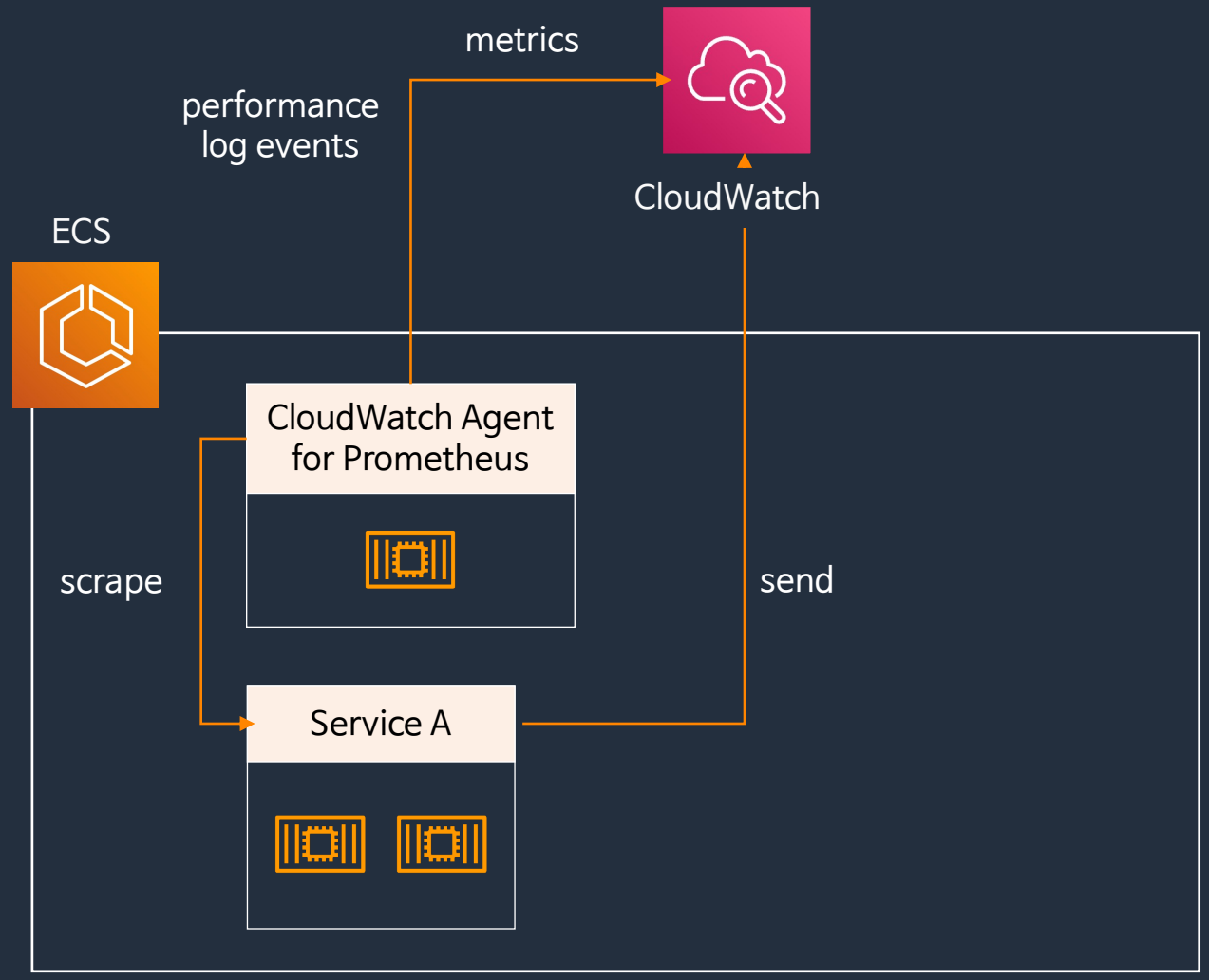
Service Autoscaling



Service Autoscaling

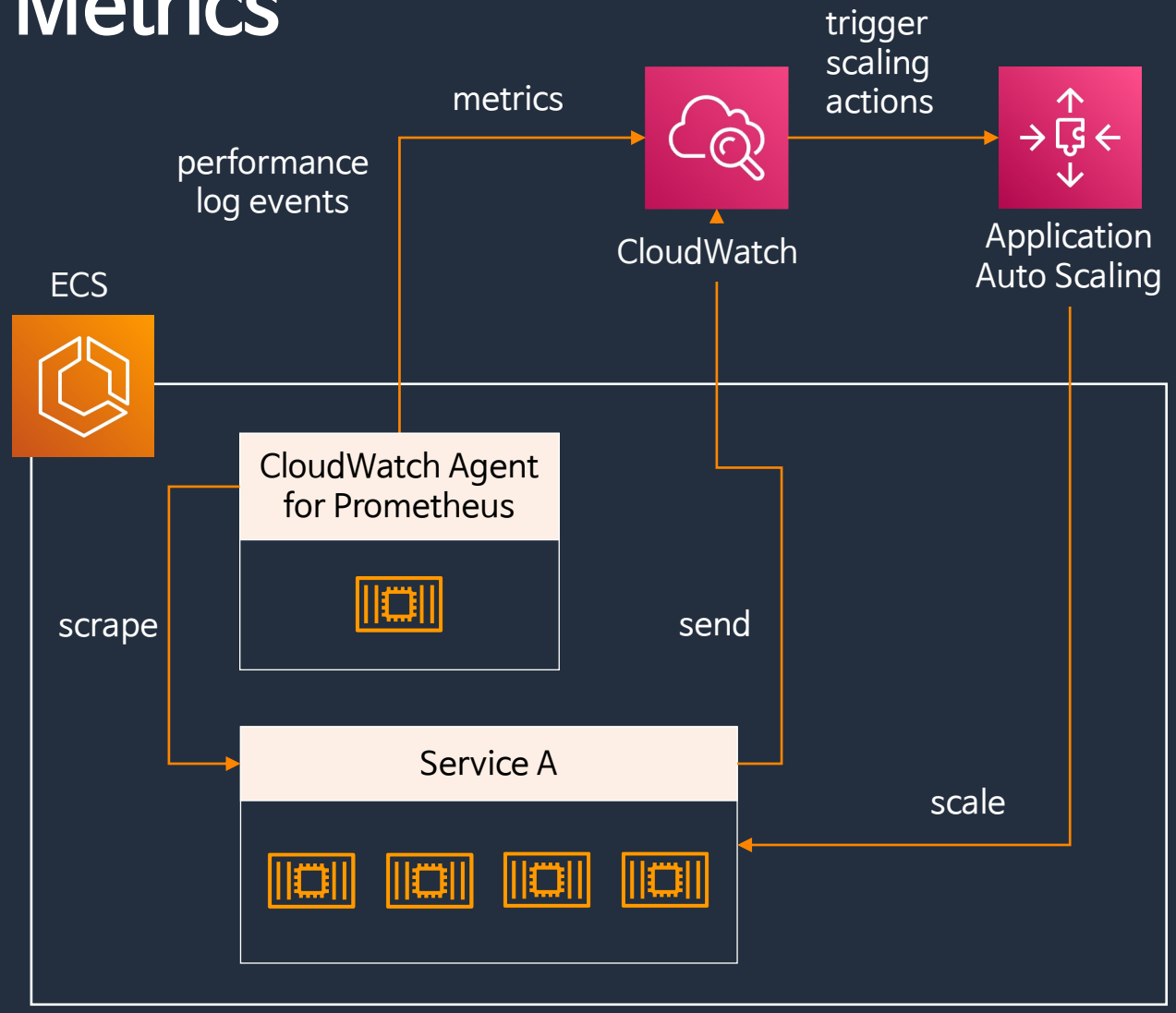


with Service Metrics



https://aws.amazon.com/ko/blogs/containers/autoscaling-amazon-ecs-services-based-on-custom-cloudwatch-and-prometheus-metrics/

with Service Metrics



https://aws.amazon.com/ko/blogs/containers/autoscaling-amazon-ecs-services-based-on-custom-cloudwatch-and-prometheus-metrics/

with Service Metrics

AWS 기술 블로그

애플리케이션 오토 스케일링의 사용자 정의 지표 기반으로 Amazon ECS 오토 스케일링 하기

by Jungseob Shin | on 21 4월 2023 | in Amazon CloudWatch, Containers | Permalink | Share

이 글은 AWS Containers 블로그의 Autoscaling Amazon ECS services based on custom metrics with Application Auto Scaling by Viji Sarathy and Anoop Singh의 한국어 번역입니다.

소개

애플리케이션 오토 스케일링은 확장 가능한 리소스를 자동으로 확장할 수 있는 솔루션이 필요한 개발자 및 시스템 관리자를 위한 웹 서비스입니다. Amazon Elastic Container Service (Amazon ECS) 서비스, Amazon DynamoDB 테이블, AWS Lambda Provisioned Concurrency 등과 같은 AWS 서비스를 위한 리소스를 자동으로 스케일링할 수 있습니다. 이제 애플리케이션 오토 스케일링은 Amazon CloudWatch 사용자 정의 지표를 기반으로 하는 스케일링 정책을 사용하여 이러한 리소스를 스케일링하는 기능을 제공합니다. 이 글에서는 이 기능을 활용해서 Amazon ECS 서비스에서 HTTP 요청의 평균 처리 속도를 기반으로 오토 스케일링하는 예시를 통해 어떻게 작동하는지 보여드리겠습니다.

배경설명

수평 확장성은 클라우드 네이티브 애플리케이션에서 매우 중요한 부분입니다. 애플리케이션 오토 스케일링은 여러 AWS 서비스와 통합되어 애플리케이션의 요구에 맞게 스케일링 기능을 추가할 수 있습니다. Amazon CloudWatch에서 사용 가능한 관련 사전 정의된 지표 중 하나 이상을 대상 추적(target tracking) 또는 단계별(step) 스케일링 정책과 함께 사용하여 주어진 서비스의 리소스를 비례적으로 확장할 수 있습니다. 하지만 사전 정의된 지표만으로는 스케일링 작업을 언제 실행하고 얼마나 실행할지에 대해 신뢰성 측면에서 다소 부족한 경우가 있습니다. 특정 시나리오에서는 HTTP 요청 수, 큐/토픽에서 검색된 메시지 수, 실행된 데이터베이스 트랜잭션 수 등의 애플리케이션 측면을 추적하는 사용자 지정 지표가 스케일링 작업을 작동시키는 데 더 적합할 수 있습니다.

애플리케이션 오토 스케일링과 함께 대상 추적 정책을 사용할 때 중요한 고려 사항은 지정된 지표가 확장 가능한 대상이 얼마나 바쁜지를 설명할 수 있게 평균 사용률을 나타내야 한다는 것입니다. 애플리케이션이 수신한 HTTP 요청 수, 메시지 브로커의 큐/토픽에서 검색된 메시지 수와 같은 지표는 본질적으로 누적되어 단조롭게 증가하기 때문에 이런 고려 사항을 충족하지 못합니다. 따라서 확장 가능한 대상의 용량에 비례하여 증가하거나 감소하는 사용률 지표로 변환해야 합니다. 이전에는 고객이 이 변환을 수행하는 전용 코드를 작성해야 했습니다. 이런 사례의 대표적인 예는 이 게시글에서 제시되고, Amazon Managed Streaming for Apache Kafka(Amazon MSK)의 토픽에 게시된 메시지 속도를 기반으로 Amazon ECS 서비스의 작업(Task) 수를 확장하는 방법을 자세히 설명합니다. 이 방법은 AWS Lambda 함수를 사용하여 주기적으로 Amazon CloudWatch에서 사용자 지정 지표 데이터를 가져온 다음 평균 사용률 지표를 계산하고 이를 Amazon CloudWatch에 새로운 사용자 지정 지표로 게시합니다. 그런 다음 미리 계산된 이 새로운 사용률 지표를 참조하는 사용자 지정 지표 사양을 사용하여 대상 추적 정책을 정의합

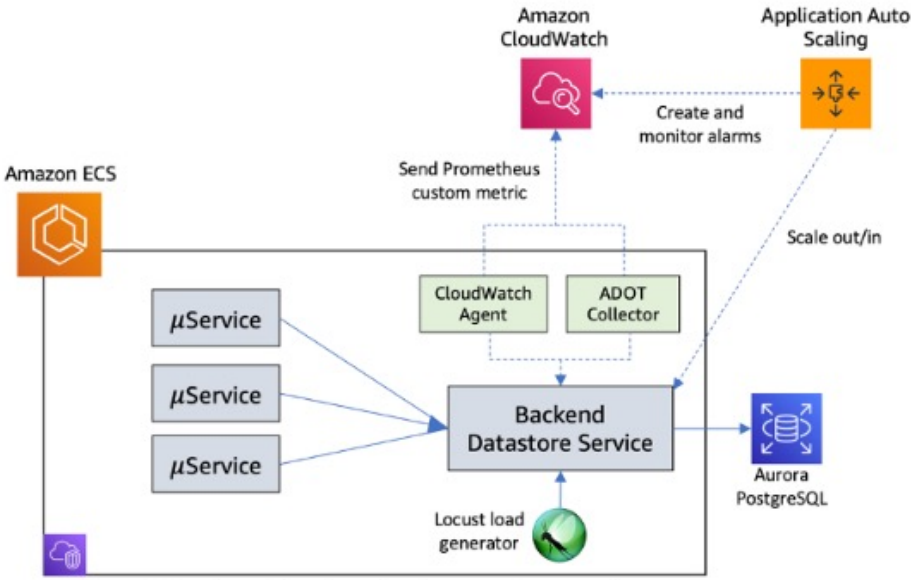
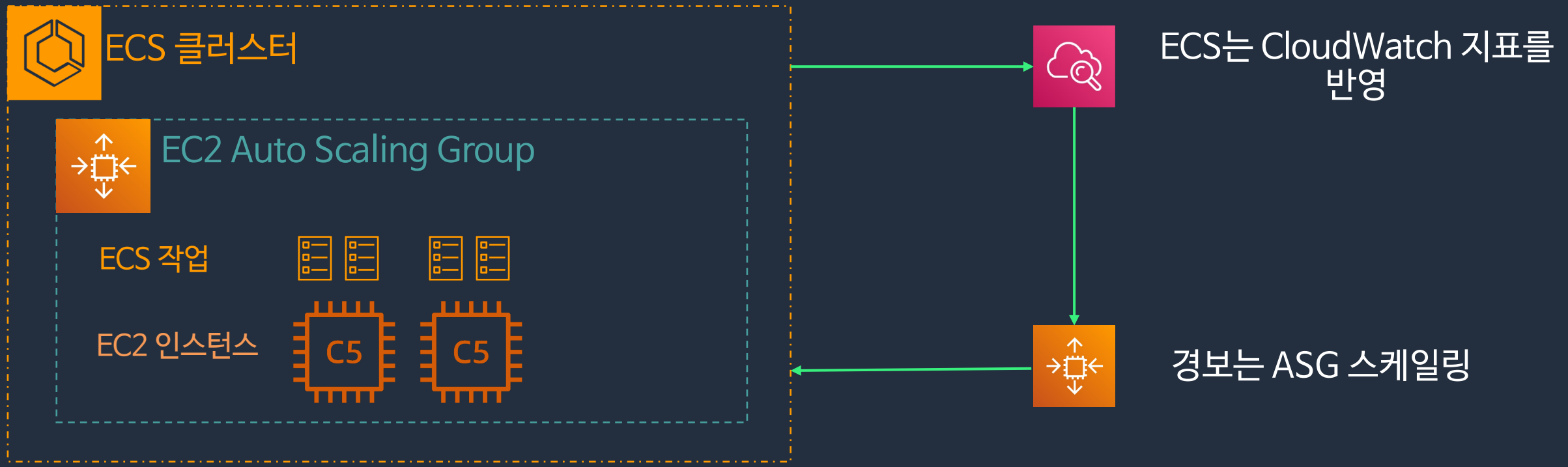


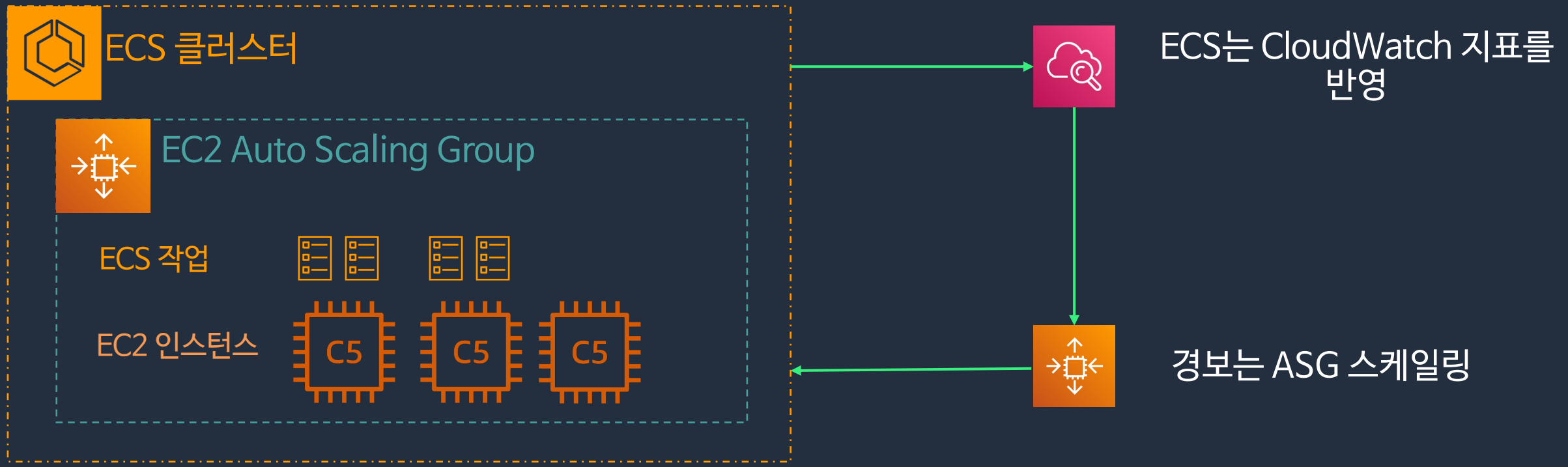
Figure 1. Sample workload used to demonstrate autoscaling

https://aws.amazon.com/ko/blogs/tech/autoscaling-amazon-ecs-services-based-on-custom-metrics-with-application-auto-scaling/

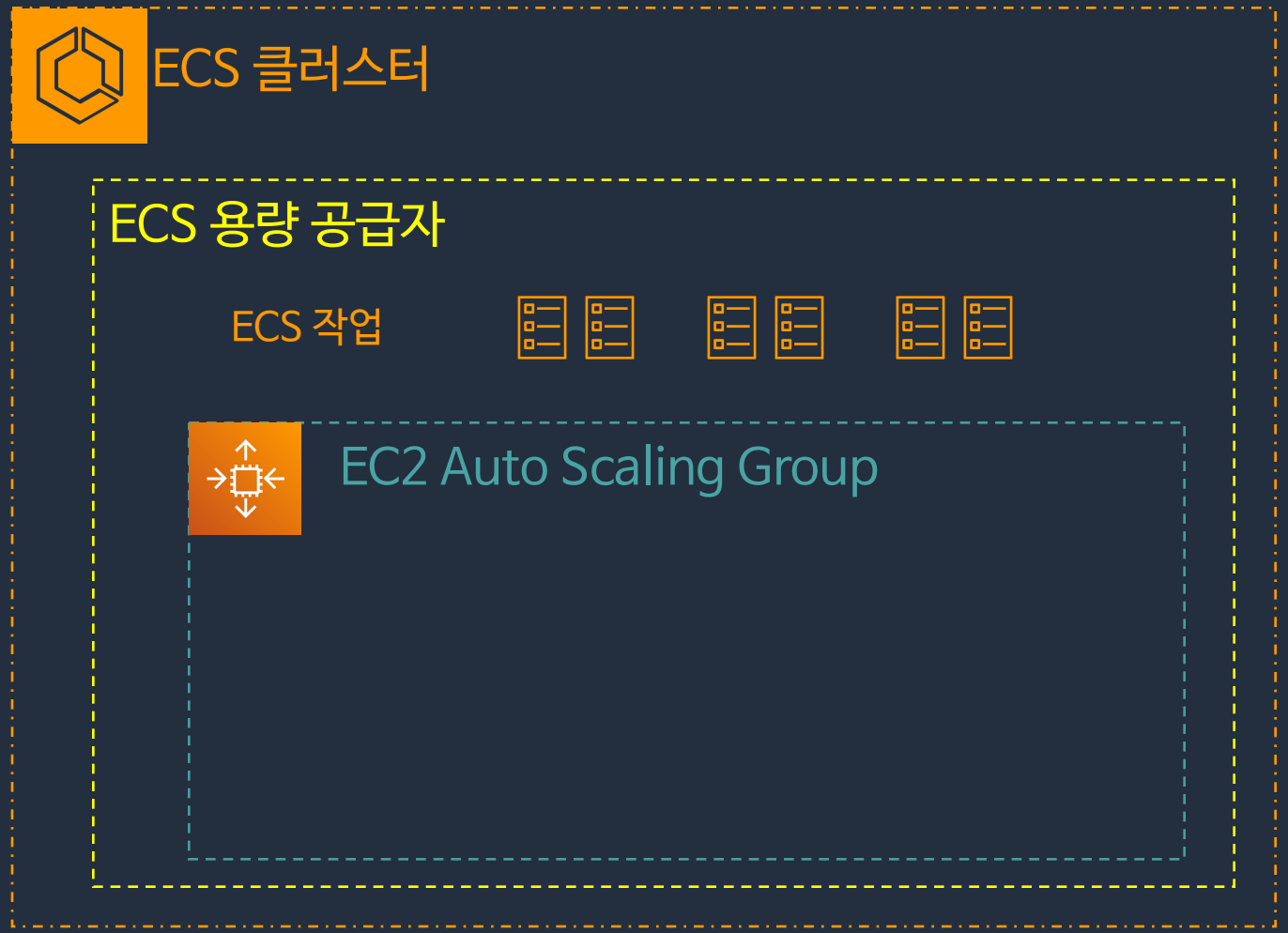
클러스터 스케일링



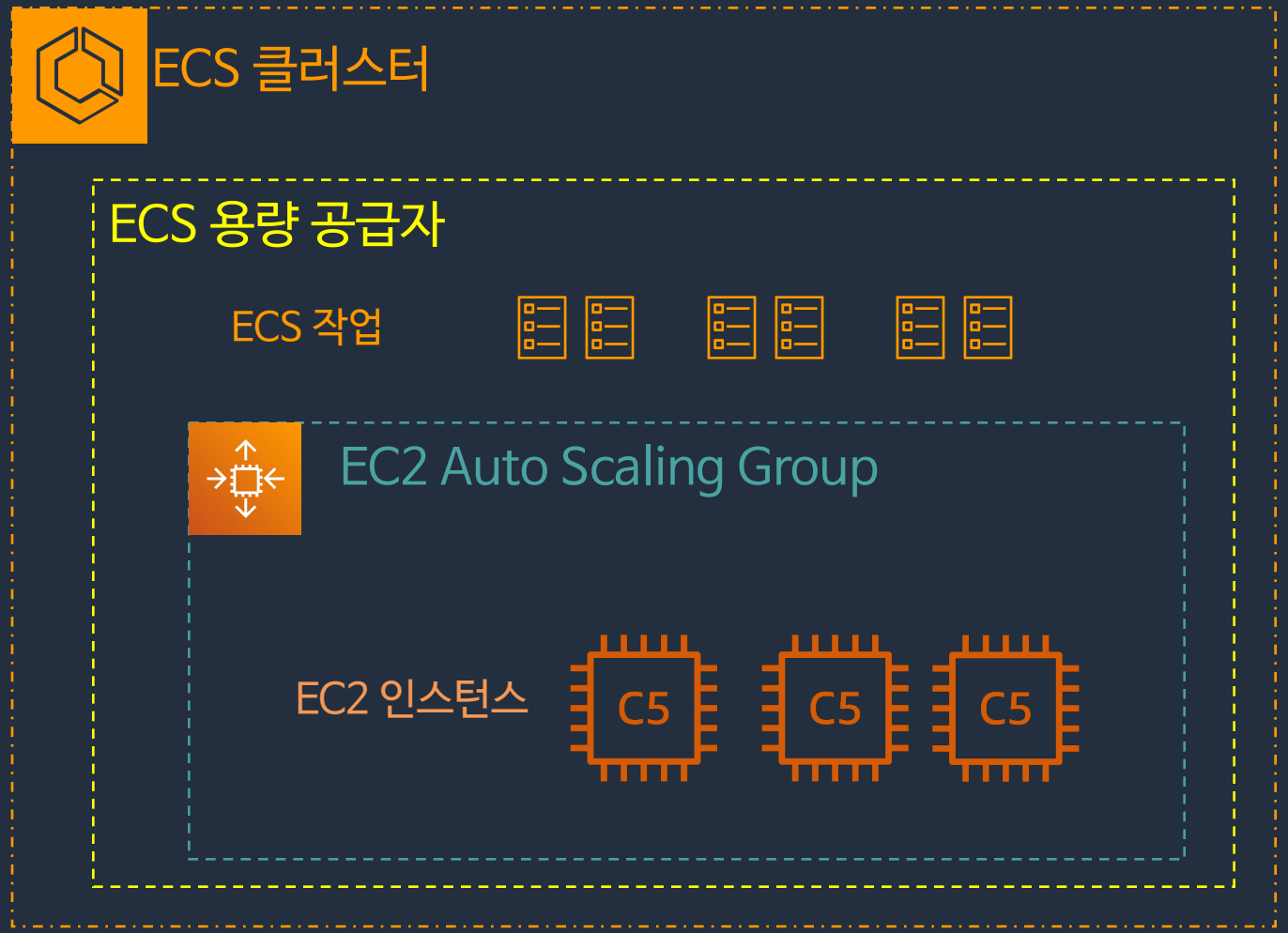
클러스터 스케일링



Amazon ECS 용량 공급자



Amazon ECS 용량 공급자



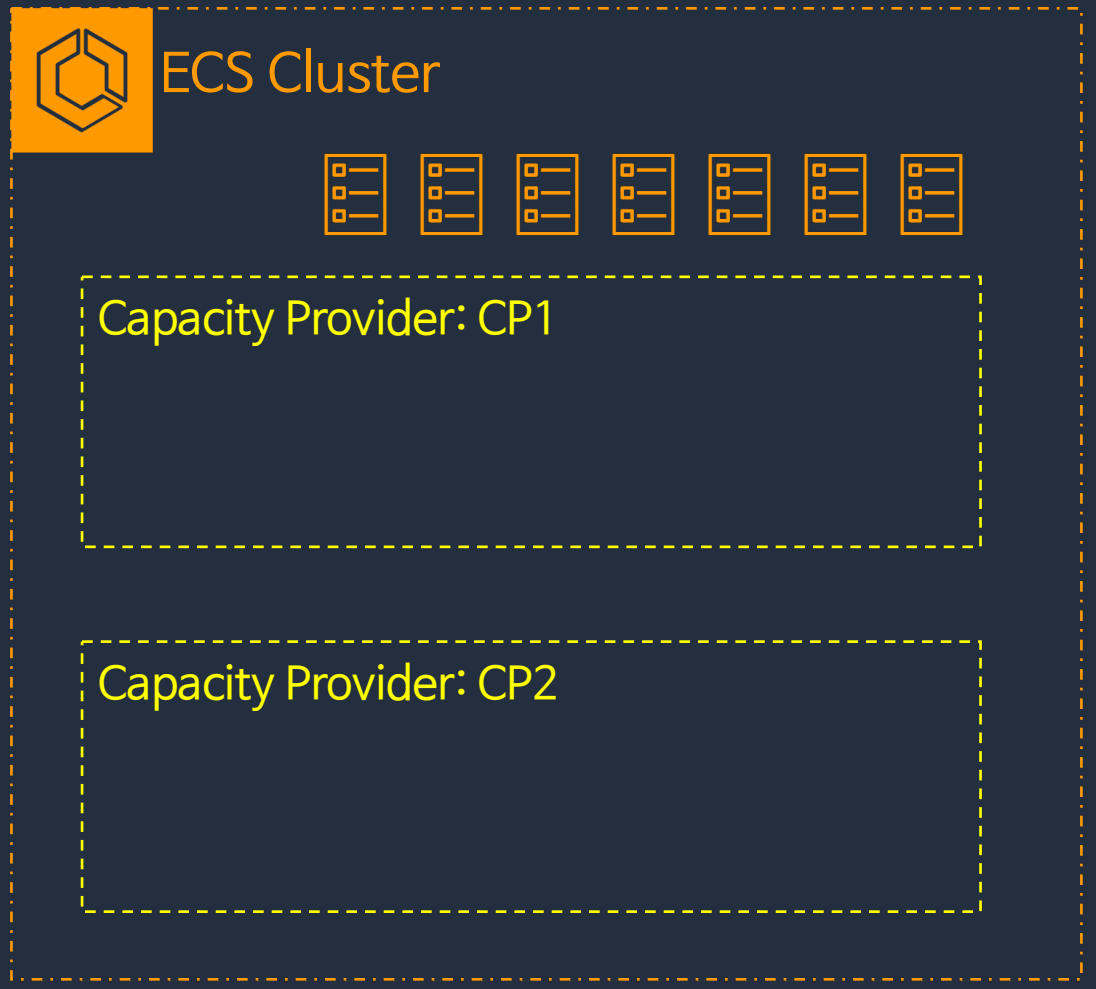
용량 공급자 전략

Capacity Provider Strategy

CP1: base = 3, weight = 1

CP2: base = 0, weight = 3

7 Tasks, CP1 = 3, CP2 = 3



https://aws.amazon.com/blogs/containers/deep-dive-on-amazon-ecs-cluster-auto-scaling/

How does CAS Work

- CAS는 CloudWatch metric 중 CapacityProviderReservation 을 추적하도록 동작합니다.
- CapacityProviderReservation = M/N * 100
  - M = ASG에서 필요한 인스턴스 개수 (to be computed)
  - N = ASG에서 현재 실행중인 인스턴스 개수(known)
- CAS는 아래 내용을 고려하는 알고리즘을 기반으로 M을
  - 단일 인스턴스 유형 또는 혼합 인스턴스 유형을 사용하도
  - 프로비저닝 상태에서 대기 중인 작업의 리소스 요구 사항(
  - 작업에 대한 배치 전략(있는 경우)

For ASGs configured to use a single instance type

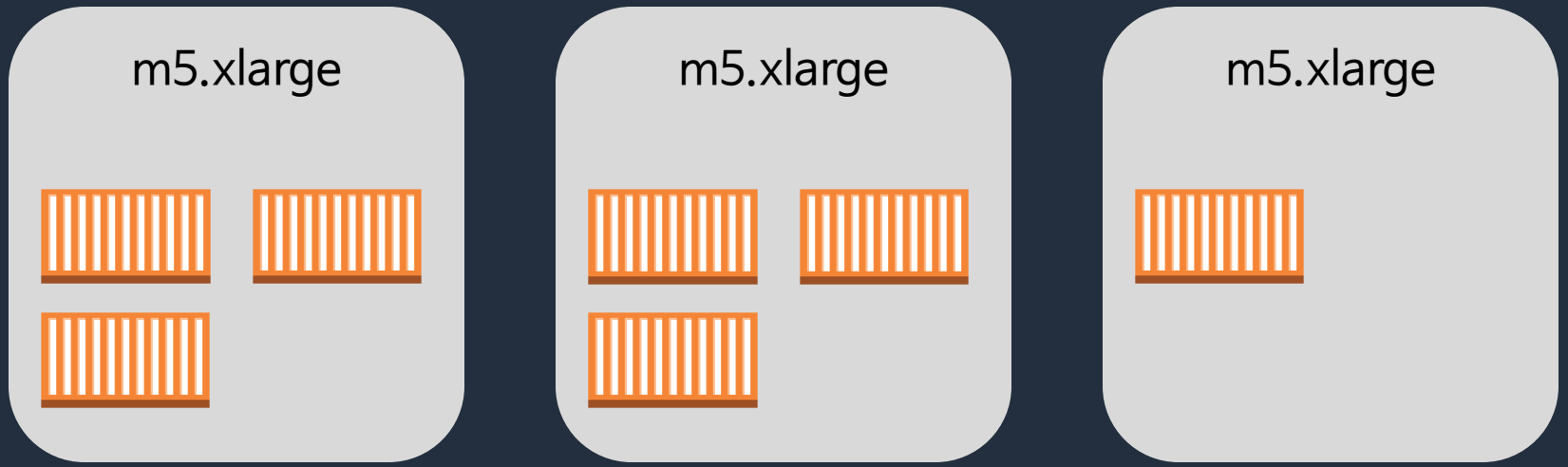
1. Group all of the provisioning tasks so that each group has the exact same resource requirements.
2. Fetch the instance type and its attributes that the ASG is configured to use.
3. For each group with identical resource requirements, calculate the number of instances required if a `binpack` placement strategy were used (placement strategies can't change the lower bound of the number of instances required, only the distribution of tasks on those instances). This calculation accounts for vCPU, memory, ENI, ports, and GPUs of the tasks and the instances. Task placement constraints *are* considered. However, any placement constraint other than `distinctInstance` is not recommended.
4. Calculate M as the maximum value of step 3 across all task groups.
5. Finally, require that N + `minimumScalingStepSize` =< M <= N + `maximumScalingStepSize` . (These two parameters are defined in the capacity provider configuration).

For ASGs configured to use multiple instance types

1. Group all of the provisioning tasks so that each group has the exact same resource requirements.
2. Fetch the instance types and their attributes that the ASG is configured to use.
   - A. Sort these instance types by each attribute i.e. vCPU, memory, ENI, ports, and GPU.
   - B. The largest instance types across each attribute are selected.
3. For each group with identical resource requirements, calculate the number of instances required based on each of the largest instance types identified in step 2 if a `binpack` placement strategy were used (placement strategies can't change the lower bound of the number of instances required, only the distribution of tasks on those instances). This calculation accounts for vCPU, memory, ENI, ports, and GPUs of the tasks and the instances. Task placement constraints *are* considered. However, any placement constraint other than `distinctInstance` is not recommended.
4. Calculate M as the minimum value within each task group and maximum value of those across all task groups.
5. Finally, require that N + `minimumScalingStepSize` =< M <= N + `maximumScalingStepSize` . (These two parameters are defined in the capacity provider configuration).

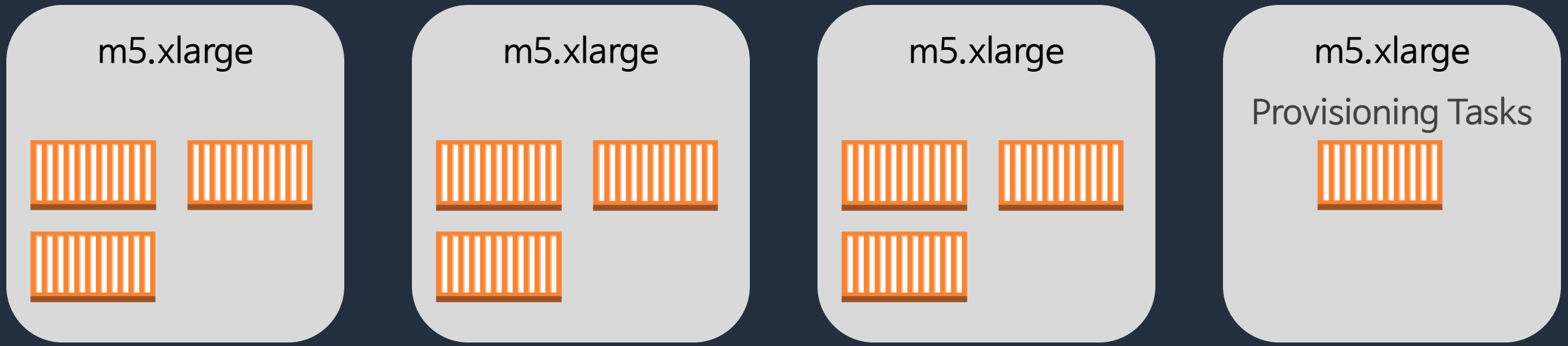
How does Cluster Scale-out Work?

- 클러스터에는 3개의 인스턴스가 있는 ASG와 함께 하나의 용량 공급자가 있습니다.
  - 서비스에 7가지 작업이 있다고 가정
- 각 인스턴스에는 세 가지 작업에 충분한 용량이 있습니다.
- 프로비저닝 상태의 작업이 없습니다
- M = 3, N = 3, CapacityProviderReservation = 100



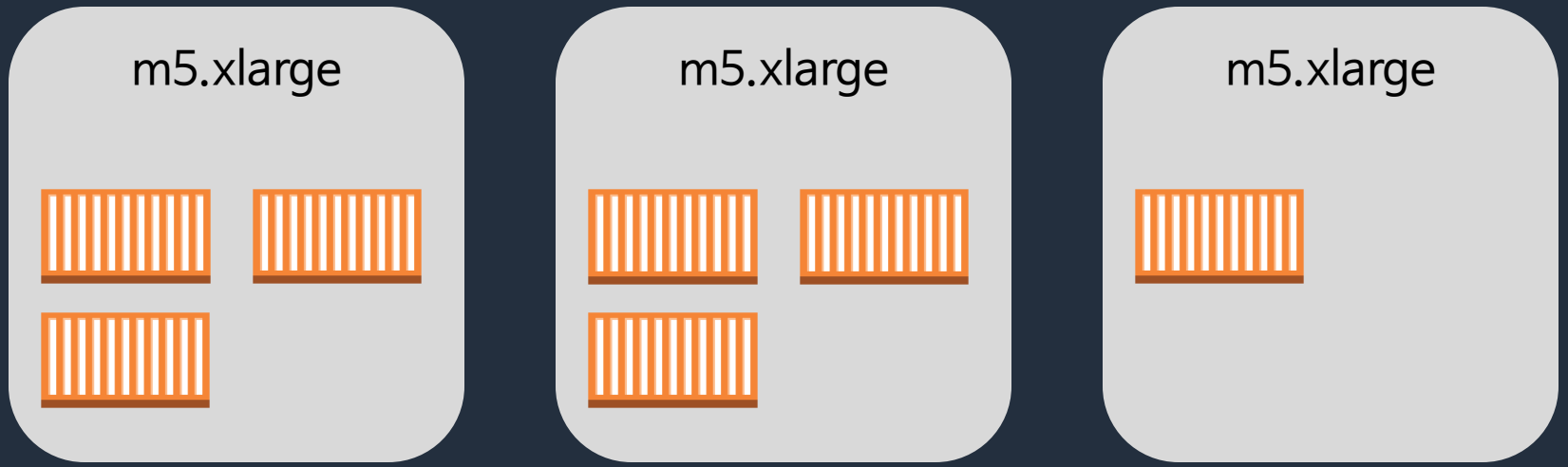
How does Cluster Scale-out Work?

- ECS 서비스 Auto Scaling은 서비스의 작업 수를 10개로 확장합니다.
- 하나의 작업이 사용 가능한 용량 없이 프로비저닝 상태입니다.
- CAS estimates M = 4, CapacityProviderReservation = 133
- ASG의 원하는 카운트를 N = 3에서 N = 4로 비례적으로 조정하기 위해 스케일링이 시작됩니다.



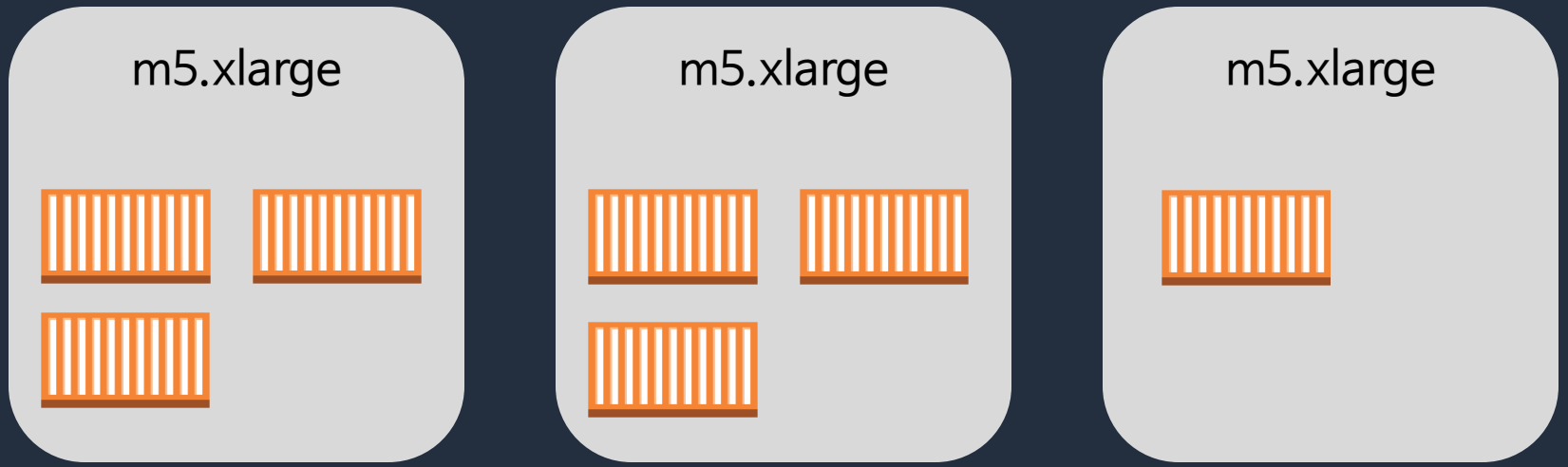
How does Cluster Scale-in Work?

- 이전과 동일한 설정
- Managed termination protection is enabled
- 프로비저닝 상태의 작업이 없는 경우, M = 3, N = 3, CapacityProviderReservation = 100



How does Cluster Scale-in Work?

- ECS Service Auto Scaling은 서비스의 작업 수를 **4개**로 축소합니다.
- CAS estimates M = 2, CapacityProviderReservation = 66
- ASG의 N = 3에서 N = 2로 조정하기 위해 스케일링이 시작됩니다.

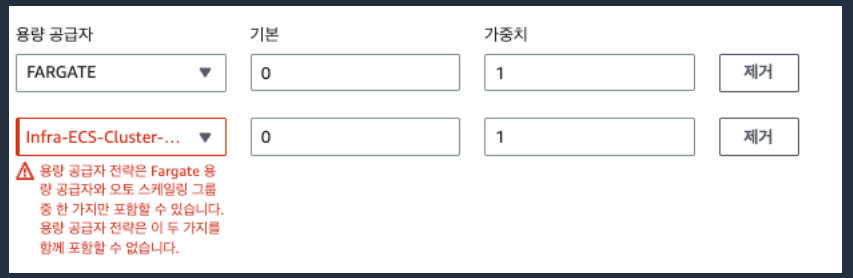


용량 공급자 고려 사항 - EC2

- 하나 이상 작업을 시작하면 용량 사용 가능 여부에 따라 작업이 실행되거나 실행되지 않음
  - Cluster AutoScaling이 활성화되면 충분한 용량이 없을 때 PROVISIONING 상태로 전환
- Enable managed scaling
- Enable instance scale-in Protection
- 용량 공급자가 관리하는 ASG을 수정하지 않음
- Auto Scaling에 Warm Pool을 추가 할 수 있음
  - **ECS_WARM_POOLS_CHECK=true**

용량 공급자 고려 사항 – Fargate

- 새 콘솔에서 Fargate 및 Fargate Spot은 용량 공급자를 생성할 필요가 없음
- Fargate와 EC2 **용량 공급자 전략**을 함께 쓸 수 없음
- Fargate Spot 용량 공급자 사용 시 Windows 컨테이너는 지원하지 않음
- ARM 64 아키텍처를 사용하는 Linux Task는 Fargate Spot을 지원하지 않음



Deep Dive on ECS Cluster Auto Scaling

Containers

Deep Dive on Amazon ECS Cluster Auto Scaling

by Nick Coult | on 03 JAN 2020 | in Amazon EC2, Amazon EC2 Container Service, Amazon Elastic Container Service, Auto Scaling, Containers, Top Posts | Permalink | Comments | Share

Introduction

Up until recently, ensuring that the number of EC2 instances in your ECS cluster would scale as needed to accommodate your tasks and services could be challenging. ECS clusters could not always scale out when needed, and scaling in could impact availability unless handled carefully. Sometimes, customers would resort to custom tooling such as Lambda functions, custom metrics, and other heavy lifting to address the challenges, but there was no single approach that could work in all situations. Of course, running your tasks on Fargate instead of EC2 instances eliminates the need for scaling clusters entirely, but not every customer is ready or able to adopt Fargate for all of their workloads.

ECS Cluster Auto Scaling (CAS) is a new capability for ECS to manage the scaling of EC2 Auto Scaling Groups (ASG). With CAS, you can configure ECS to scale your ASG automatically, and just focus on running your tasks. ECS will ensure the ASG scales in and out as needed with no further intervention required. CAS relies on ECS capacity providers, which provide the link between your ECS cluster and the ASGs you want to use. Each ASG is associated with a capacity provider, and each such capacity provider has only one ASG, but many capacity providers can be associated with one ECS cluster. In order to scale the entire cluster automatically, each capacity provider manages the scaling of its associated ASG.

One of our goals in launching CAS is that scaling ECS clusters "just works" and you don't have to think about it. However, you might still want to know what is happening behind the scenes. In this blog post, I'm going to deep dive on exactly how CAS works.

The core scaling logic

The core responsibility of CAS to ensure that the "right" number of instances are running in an ASG to meet the needs of the tasks assigned to that ASG, including tasks already running as well as tasks the customer is trying to run that don't fit on the existing instances. Let's call this number M. Let's also call the current number of instances in the ASG that are already running N. We'll make extensive use of M and N throughout the rest of the blog post, so it's important to have a completely clear understanding of how to think about them. For now, we haven't explained how we know what M should be, but for the purposes of discussion, let's assume that M is what you need. Given this assumption, if N = M, scaling out is not required, and scaling in isn't possible. On the other hand, if N < M, scale out is required because you don't have enough instances. Lastly, if N > M, scale in is possible (but not necessarily required) because you have more instances than you need to run all of your ECS tasks. As we will see later, we also define a new CloudWatch metric based on N and M, called the `CapacityProviderReservation`. Given N and M, this metric has a very simple definition:

CapacityProviderReservation = M / N X 100

To put it in plain language, the metric is the ratio of how big the ASG needs to be relative to how big it actually is, expressed as a percentage. As explained later in this blog, this metric is used by CAS to control the scaling of the ASG. In the formula above, the number M is the part that CAS controls; in turn, M is driven by the customer's tasks (both already running and waiting to run). How M is calculated is key to how CAS actually does the scaling.

In order to determine M, we need to have a concept of tasks that the customer is trying to run that don't fit on existing instances. To achieve this, we adapted the existing ECS task lifecycle. Previously, tasks would either run or not, depending on whether capacity was available. Now, tasks in the provisioning state include tasks that could not find sufficient resources on the existing instances. This means, for example, if you call the RunTask API and the tasks don't get placed on an instance because of insufficient resources (meaning no active instances had sufficient memory, vCPUs, ports, ENIs, and/or GPUs to run the tasks), instead of failing immediately, the task will go into the provisioning state (note, however, that the transition to provisioning only happens if you have enabled managed scaling for the capacity provider; otherwise, tasks that can't find capacity will fail immediately, as they did previously). As more instances become available, tasks in the provisioning state will get placed onto those instances, reducing the number of tasks in provisioning. In some sense, you can think of the provisioning tasks as a queue; task that can be placed due to resources get added to the queue, and as more resources become available, tasks get removed from the queue.

https://aws.amazon.com/blogs/containers/deep-dive-on-amazon-ecs-cluster-auto-scaling/

Capacity Provider Demo

EC2 Spot and Fargate Spot

Amazon EC2 Spot

Unused Capacity

Price up to 90% less than On-Demand

Can be *reclaimed* by Amazon EC2 (with two minute warning)

You choose instance pools

AWS Fargate Spot

Unused Capacity

Price up to 70% less than standard Fargate

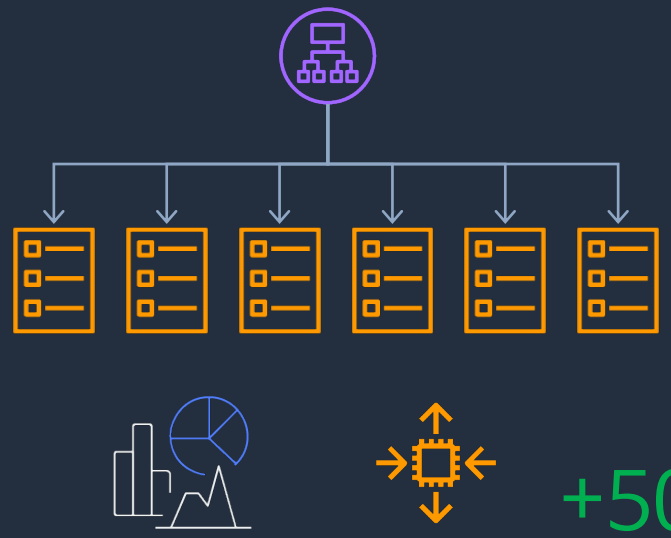
Can be *reclaimed* (with two minute warning)

Automatic diversification

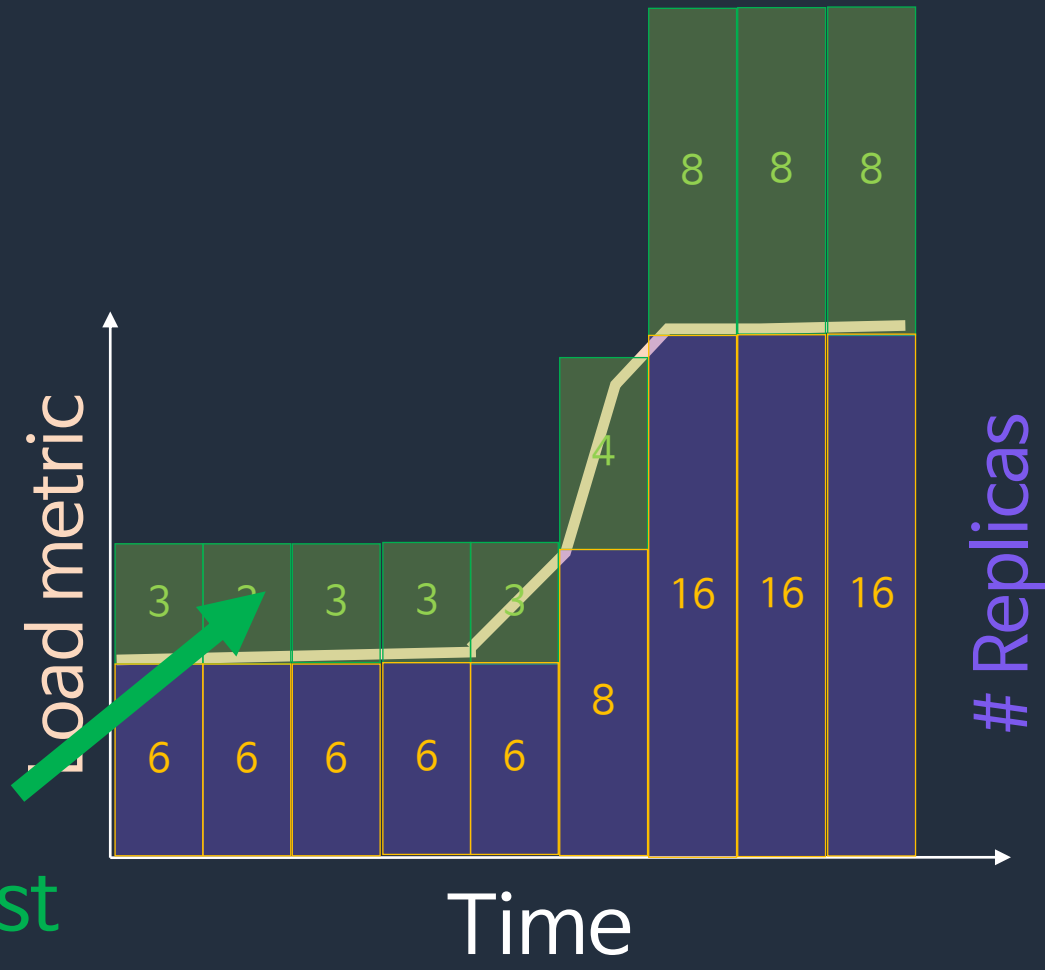
Splitting Across Capacity Providers: OD and Spot

Overprovision by 50%:
Reduce metric target value by 1/3

Run 2/3 On-Demand, 1/3 on Spot.



+50% capacity
for +5-10% cost



ECS Spot 중단 처리

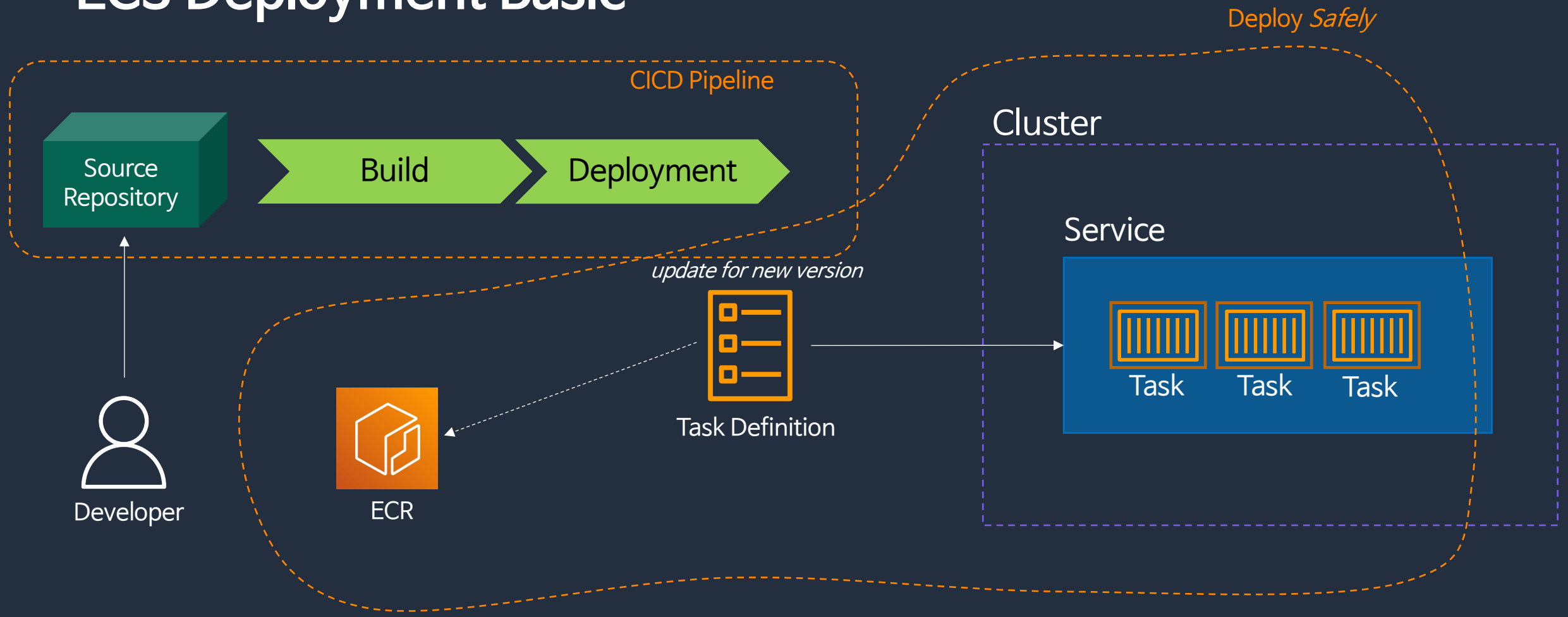
- Amazon ECS는 Spot 중단을 자동으로 처리합니다.
  - set ECS_ENABLE_SPOT_INSTANCE_DRAINING=true on EC2 instance – User Data
- ECS는 Task termination을 조정합니다.
- 로드 밸런서 연결을 자동으로 정상 종료시킵니다.

Type	EC2 Spot	Fargate Spot
Interruption Behavior	Terminate/Stop/ Hibernate	Stop-Task
2min Spot Interruption Warning at T0	T0 – Instance Metadata T0 – Eventbridge T0 - containers receive SIGTERM signal T0+30 sec - containers get SIGKILL signal	T0 – task state change event to EventBridge T0 - containers receive SIGTERM signal T0+30 sec - containers get SIGKILL signal
Change default 30sec via config	ECS_CONTAINER_STOP_TIMEOUT in user-data section	stopTimeout in Task Definition

https://aws.amazon.com/ko/premiumsupport/knowledge-center/fargate-spot-termination-notice/?nc1=h_ls

Deployment

ECS Deployment Basic



Deployment Types

1. Rolling
2. Blue/Green Deployment
3. External Tools (3rd Party)

Rolling Deployment

ECS Service Scheduler는 배포 구성에 따라 기존 작업을 새 작업으로 바꿉니다.

배포 구성

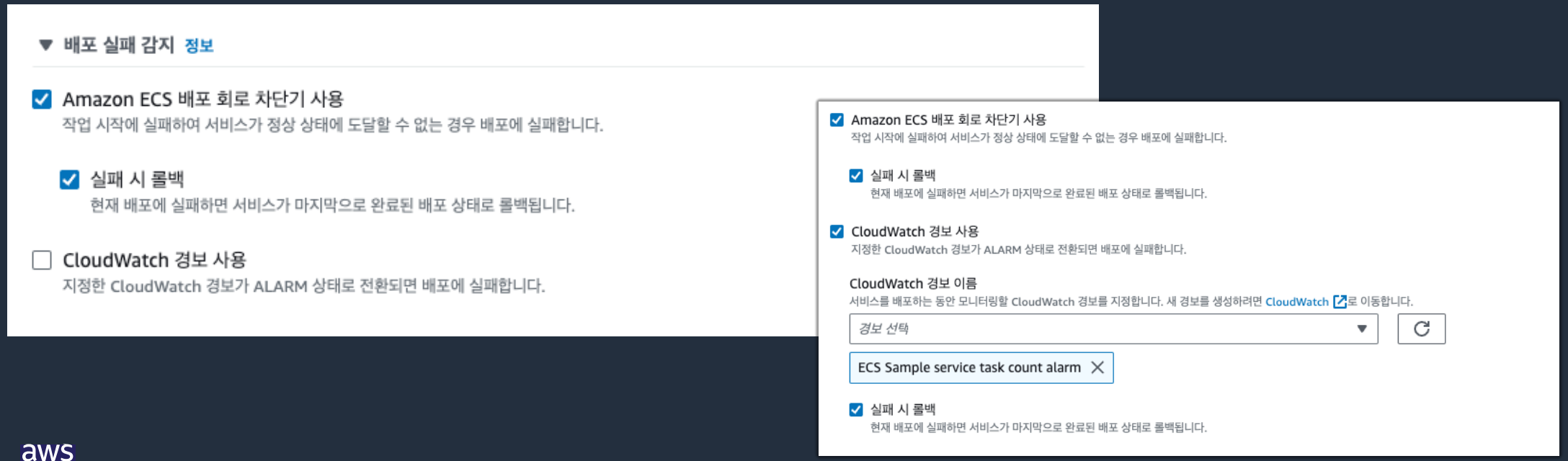
- 최소 실행 작업 비율
- 최대 실행 작업 비율

Rolling Deployment - Failure detection

배포가 실패한 시기를 식별하려면…

1. 배포 회로 차단기
2. CloudWatch 경보

이후 선택적으로 마지막 작업 배포로 롤백할 수 있습니다.



만약 배포에 실패했거나 Task가 중지됐다면?

서비스 배포 상태 변경 이벤트

Amazon ECS는 세부 정보 유형 **ECS 배포 상태 변경**과
EventBridge 규칙을 만드는 데 사용되는 이벤트 패턴입

```
{
    "source": [
        "aws.ecs"
    ],
    "detail-type": [
        "ECS Deployment State Char
    ]
}
```

Amazon ECS는 `INFO` 및 `ERROR` 이벤트 유형과 함

`SERVICE_DEPLOYMENT_IN_PROGRESS`
서비스 배포가 진행 중입니다. 이 이벤트는 초기 배포

`SERVICE_DEPLOYMENT_COMPLETED`
서비스 배포가 완료되었습니다. 배포 후 서비스가 안

`SERVICE_DEPLOYMENT_FAILED`
서비스 배포가 실패했습니다. 이 이벤트는 배포 회로

예 서비스 배포 실패 이벤트

서비스 배포 실패 상태 이벤트는 다음과 같은 형식으로 제공됩니다. 서비스 배포 실패 상태 이벤트는 배포 회로 차단기 논리가 켜진 서비스에 대해서만 전송됩니다. 자세한 정보는 롤링 업데이트을 참조하세요.

```
{
   "version": "0",
   "id": "ddca6449-b258-46c0-8653-e0e3aEXAMPLE",
   "detail-type": "ECS Deployment State Change",
   "source": "aws.ecs",
   "account": "111122223333",
   "time": "2020-05-23T12:31:14Z",
   "region": "us-west-2",
   "resources": [
        "arn:aws:ecs:us-west-2:111122223333:service/default/servicetest"
   ],
   "detail": {
        "eventType": "ERROR",
        "eventName": "SERVICE_DEPLOYMENT_FAILED",
        "deploymentId": "ecs-svc/123",
        "updatedAt": "2020-05-23T11:11:11Z",
        "reason": "ECS deployment circuit breaker: task failed to start."
   }
}
```

트러블 슈팅 가이드
https://docs.aws.amazon.com/ko_kr/AmazonECS/latest/developerguide/ecs_cwe_events.html#ecs_service_deployment_events
https://repost.aws/ko/knowledge-center/ecs-task-stopped

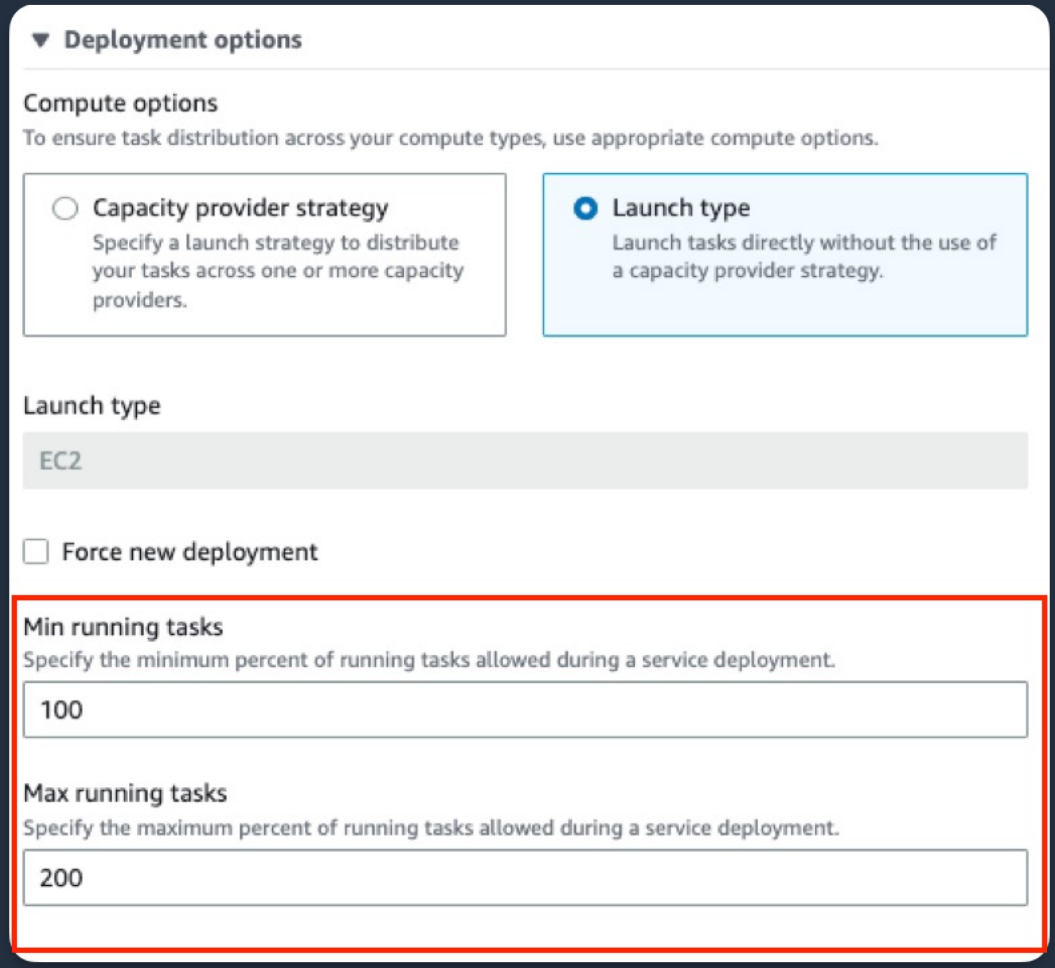
RollingUpdate Deploy Tip

ECS DEPLOYMENT 설정

- ECS Deployment 옵션 조정하기
  - Min running tasks
  - Max running tasks

Availability – Min : 100, Max : 200

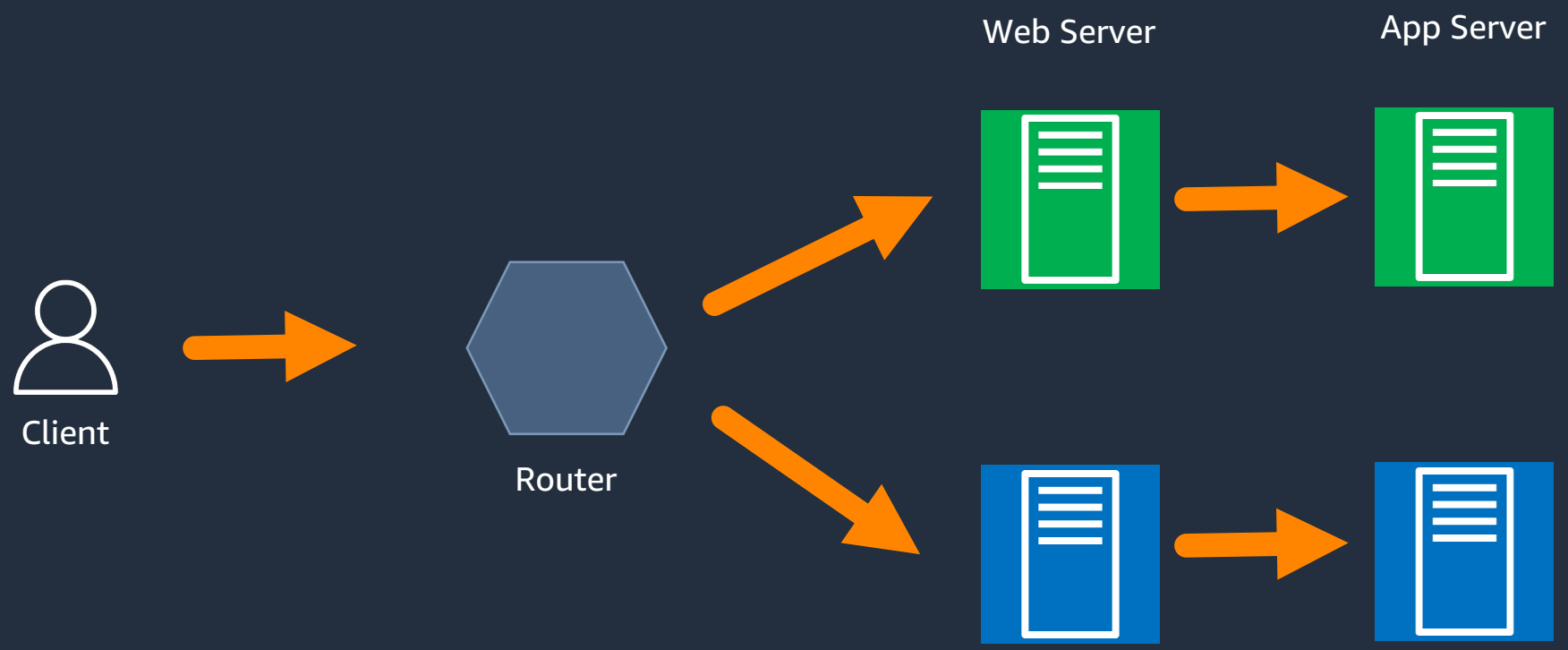
Speed – Min : 50, Max : 200



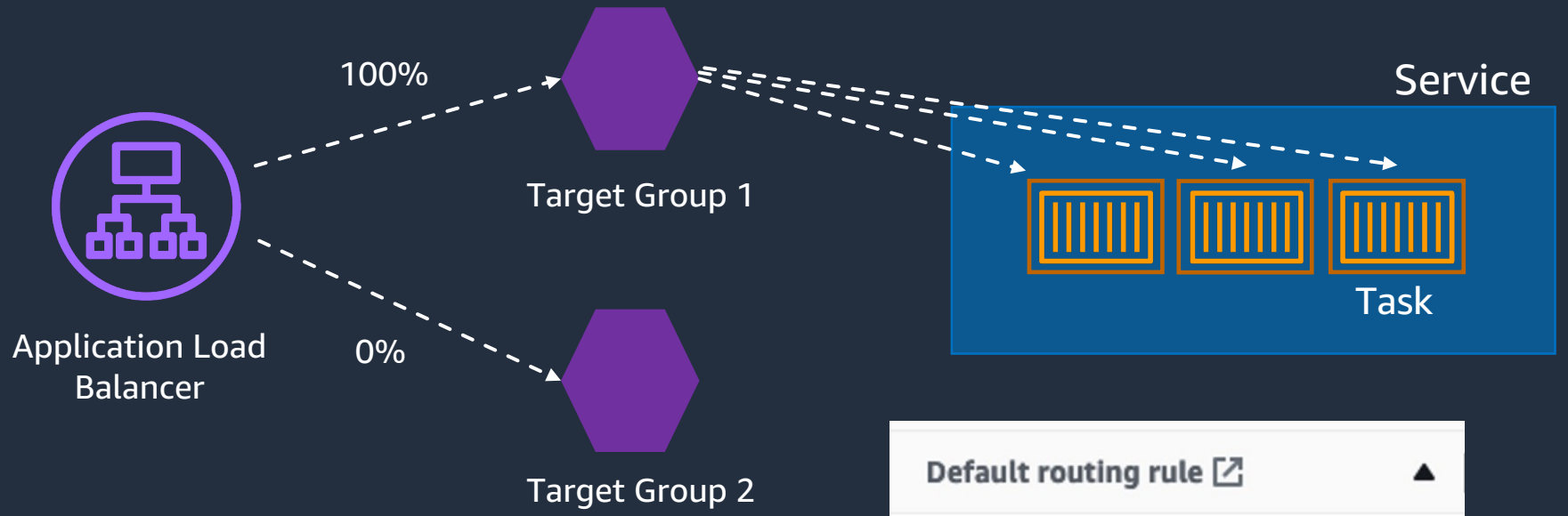
BlueGreen Deployment

애플리케이션 가용성 향상

롤백 프로세스를 단순화하여 배포 위험 감소



ECS Configuration for BlueGreen Deployment



Default routing rule

1. **Forward to**
   - bg-ecs-tg-2: 100 (100%)
   - bg-ecs-tg-1: 0 (0%)
   - *Group-level stickiness*: Off

BlueGreen Deployment - Traffic Shifting Ways

- Canary
- Linear
- All-at-once

배포 구성	설명
CodeDeployDefault.ECSLinear10PercentEvery1Minutes	모든 트래픽이 전환될 때까지 트래픽의 10%가 매분마다 전환됩니다.
CodeDeployDefault.ECSLinear10PercentEvery3Minutes	모든 트래픽이 이동될 때까지 트래픽의 10%가 3분마다 이동됩니다.
CodeDeployDefault.ECSCanary10Percent5Minutes	첫 번째 증분에 트래픽의 10%가 전환됩니다. 나머지 90%는 5분 이후 배포됩니다.
CodeDeployDefault.ECSCanary10Percent15Minutes	첫 번째 증분에 트래픽의 10%가 전환됩니다. 나머지 90%는 15분 이후 배포됩니다.
CodeDeployDefault.ECSAllAtOnce	모든 트래픽을 업데이트된 Amazon ECS 컨테이너로 한 번에 이동합니다.

Deployment Configuration

Deployments

Choose a deployment option for the service.

Deployment type* Rolling update

Blue/green deployment (powered by AWS CodeDeploy)

This sets AWS CodeDeploy as the deployment controller for the service. A CodeDeploy application and deployment group are created automatically with default settings for the service. To change to the rolling update deployment type after the service has been created, you must re-create the service and select the "rolling update" deployment type.

Deployment configuration* CodeDeployDefault.ECSCanary10Percent5Minutes

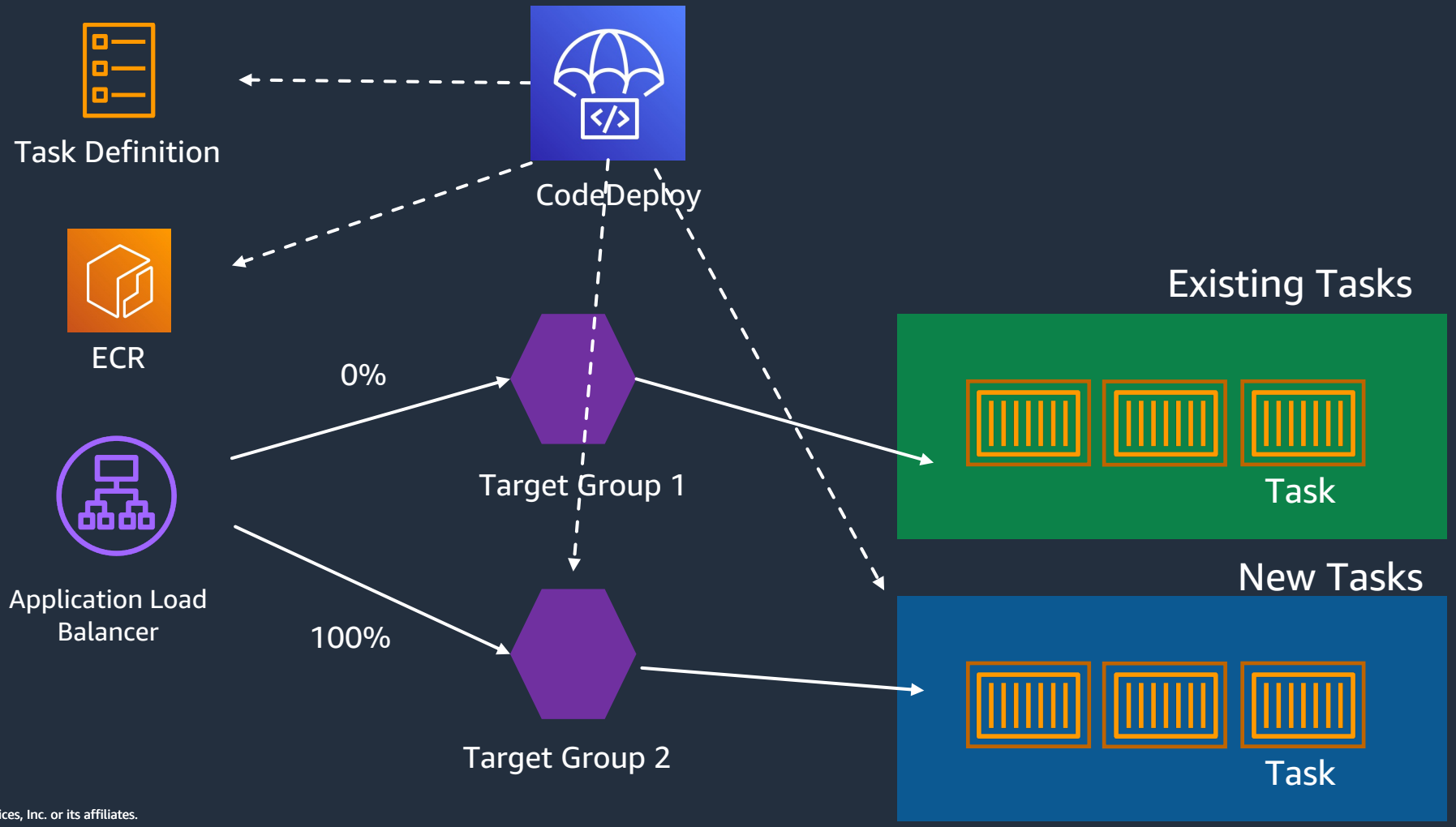
The deployment configuration specifies how traffic is shifted to the updated Amazon ECS task set. Learn more

Service role for CodeDeploy* ecsCodeDeployRole

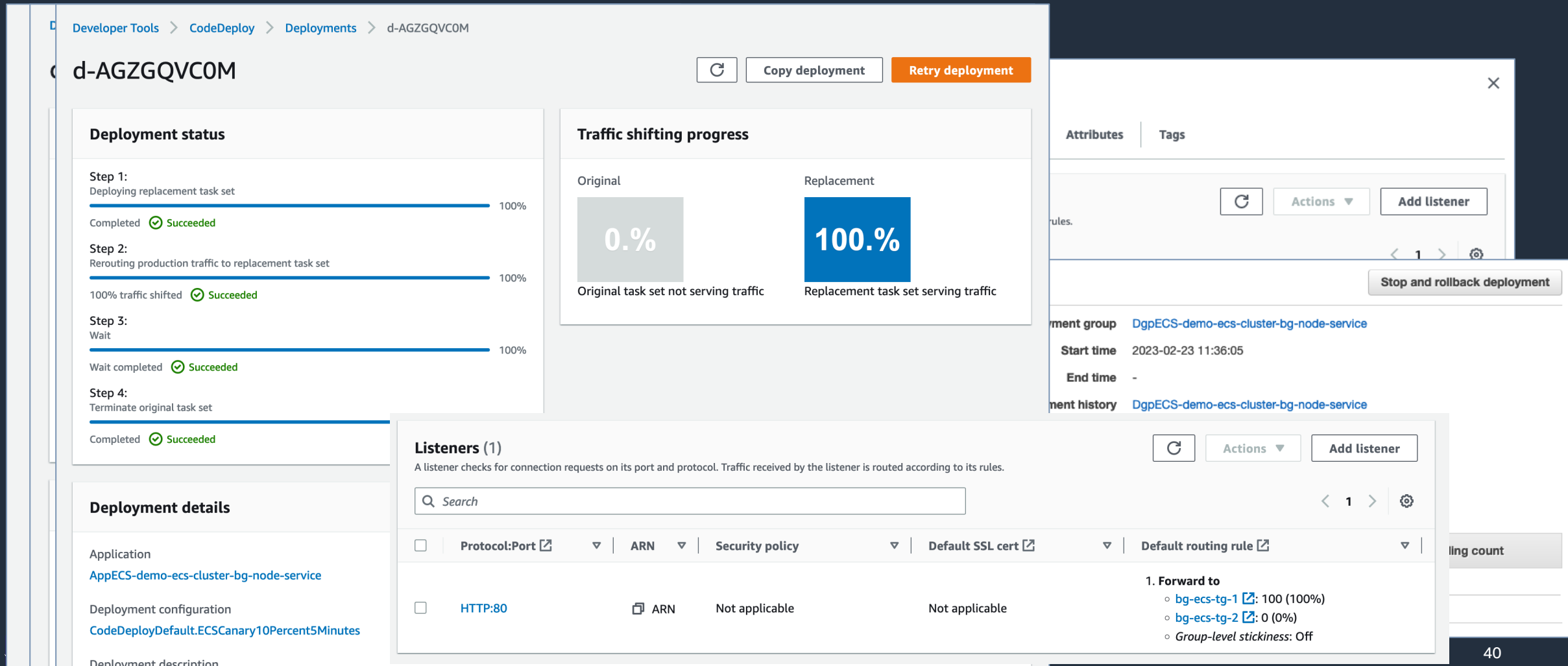
The IAM role the service uses to make API requests to authorized AWS services. Create a service role for CodeDeploy in the IAM console. Learn more

AWS CodeDeploy

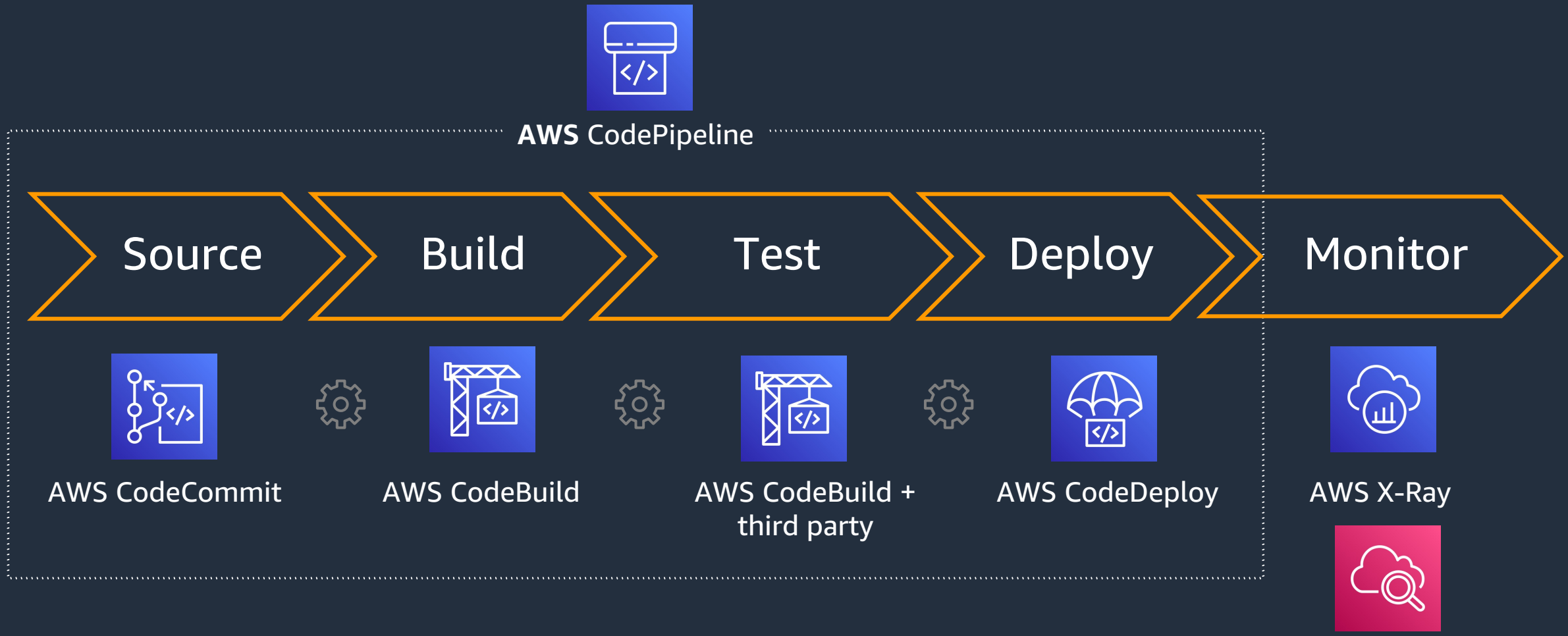
How CodeDeploy works



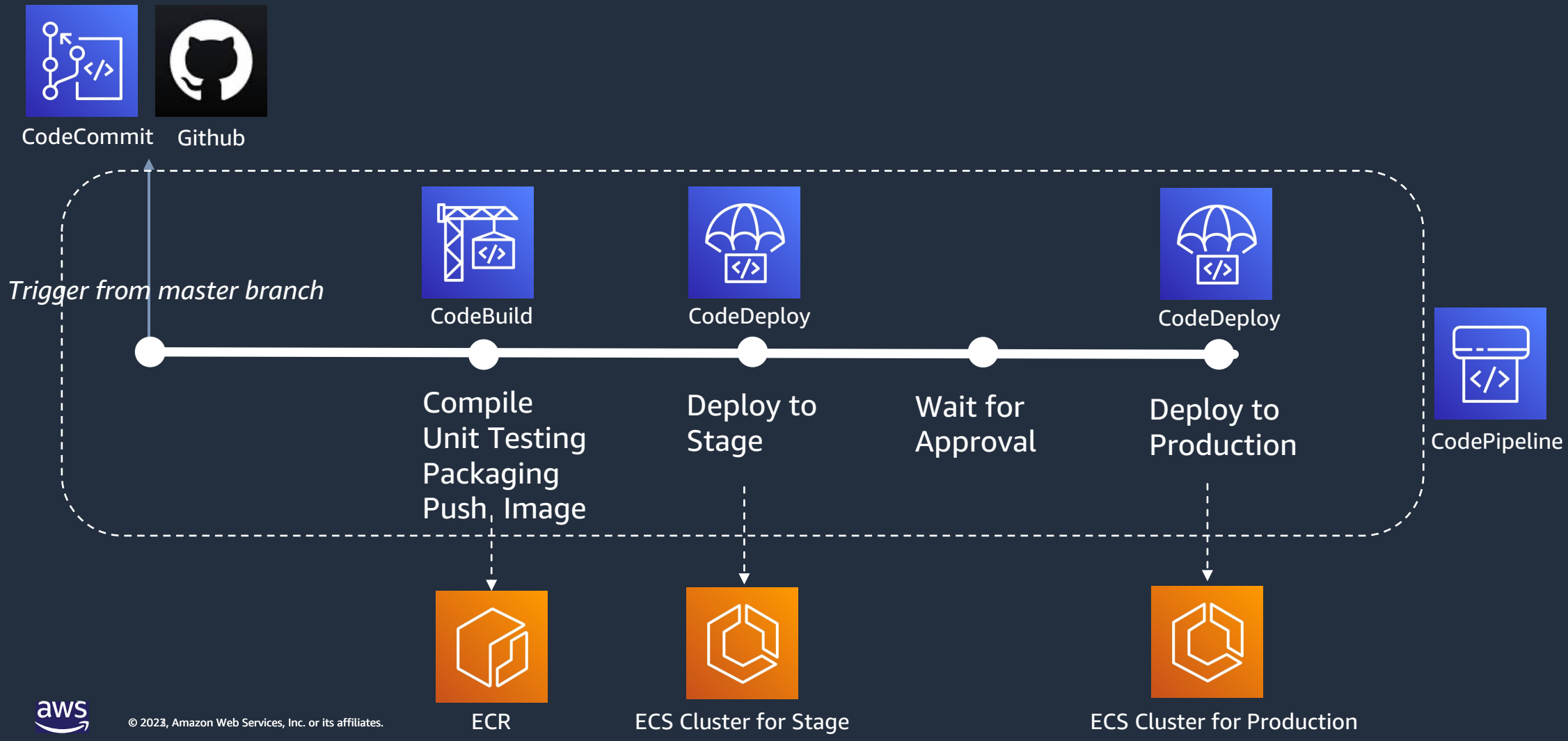
How CodeDeploy works



AWS Developer Tools for CICD



Building CICD Pipeline



Take Away

Take Away

Service Auto Scaling & Cluster Auto Scaling을 활용하여 성능, 안정성, 비용 최적화

다양한 Auto Scaling Policy 고려 (Target Tracking, Step, Scheduled)

Capacity Provider로 작업 배치 전략 구성

On-Demand + Spot Instance를 활용하여 성능 유지 및 비용 절감
(단, Interrupt가 발생할 수 있다)

Rolling Update 시 min/maxHealthPercentage를 활용하여 배포 시 일부 비용 절감

Thank you!