AWS®

Certified Developer

Official Study Guide

Associate (DVA-C01) Exam

Rolling Deployments

You can issue commands to subsets of instances in a stack or layer at a time. If you split the deployment into multiple phases, the blast radius of failures will be minimized to only a few instances that you can replace, roll back, or repair.

Blue/Green Deployments (Separate Stacks)

Much like you use separate stacks for different environments of the same application, you can also use separate stacks for different deployments. This ensures that all features and updates to an application can be thoroughly tested before routing requests to the new environment. Additionally, you can leave the previous environment running for some time to perform backups, investigate logs, or perform other tasks.

When you use Elastic Load Balancing layers and Amazon Route 53, you can route traffic to the new environment with built-in weighted routing policies. You can progressively increase traffic to the new stack as health checks and other monitoring indicate the new application version has deployed without error.

Manage Databases Between Deployments

In either deployment strategy, there will likely be a backend database with which instances running either version will need to communicate. Currently, Amazon RDS layers support registering a database with only one stack at a time.

If you do not want to create a new database and migrate data as part of the deployment process, you can configure both application version instances to connect to the same database (if there are no schema changes that would prevent this). Whichever stack does not have the Amazon RDS instance registered will need to obtain credentials via another means, such as custom JSON or a configuration file in a secure Amazon S3 bucket.

If there are schema changes that are not backward compatible, create a new database to provide the most seamless transition. However, it will be important to ensure that data is not lost or corrupted during the transition process. You should heavily test this before you attempt it in a production deployment.

Using Amazon Elastic Container Service to Deploy Containers

Amazon ECS is a highly scalable, high-performance container orchestration service that supports Docker containers and allows you to easily run and scale containerized applications on AWS. Amazon ECS eliminates the need for you to install and operate your own container orchestration software, manage and scale a cluster of virtual machines, or schedule containers on those virtual machines.

With simple API calls, you can launch and stop Docker-enabled applications, query the complete state of your application, and access many familiar features such as IAM roles, security groups, load balancers, Amazon CloudWatch Events, AWS CloudFormation templates, and AWS CloudTrail logs.

What Is Amazon ECS?

Amazon ECS streamlines the process for managing and scheduling containers across fleets of Amazon EC2 instances, without the need to include separate management tools for container orchestration or cluster scaling. *AWS Fargate* reduces management further as it deploys containers to serverless architecture and removes cluster management requirements entirely. To create a cluster and deploy services, you need only configure the resource requirements of containers and availability requirements. Amazon ECS manages the rest with the use of an agent that runs on cluster instances. AWS Fargate requires no agent management.

To react to changes in demands for your service or application, Amazon ECS supports Amazon EC2 Auto Scaling groups of cluster instances that allow your service to increase running container counts across multiple instances as demand increases. You can define container isolation and dependencies as part of the service definition. You can use the service definition to enforce requirements without user interaction, such as "only one container of type A may run on a cluster instance at a time."

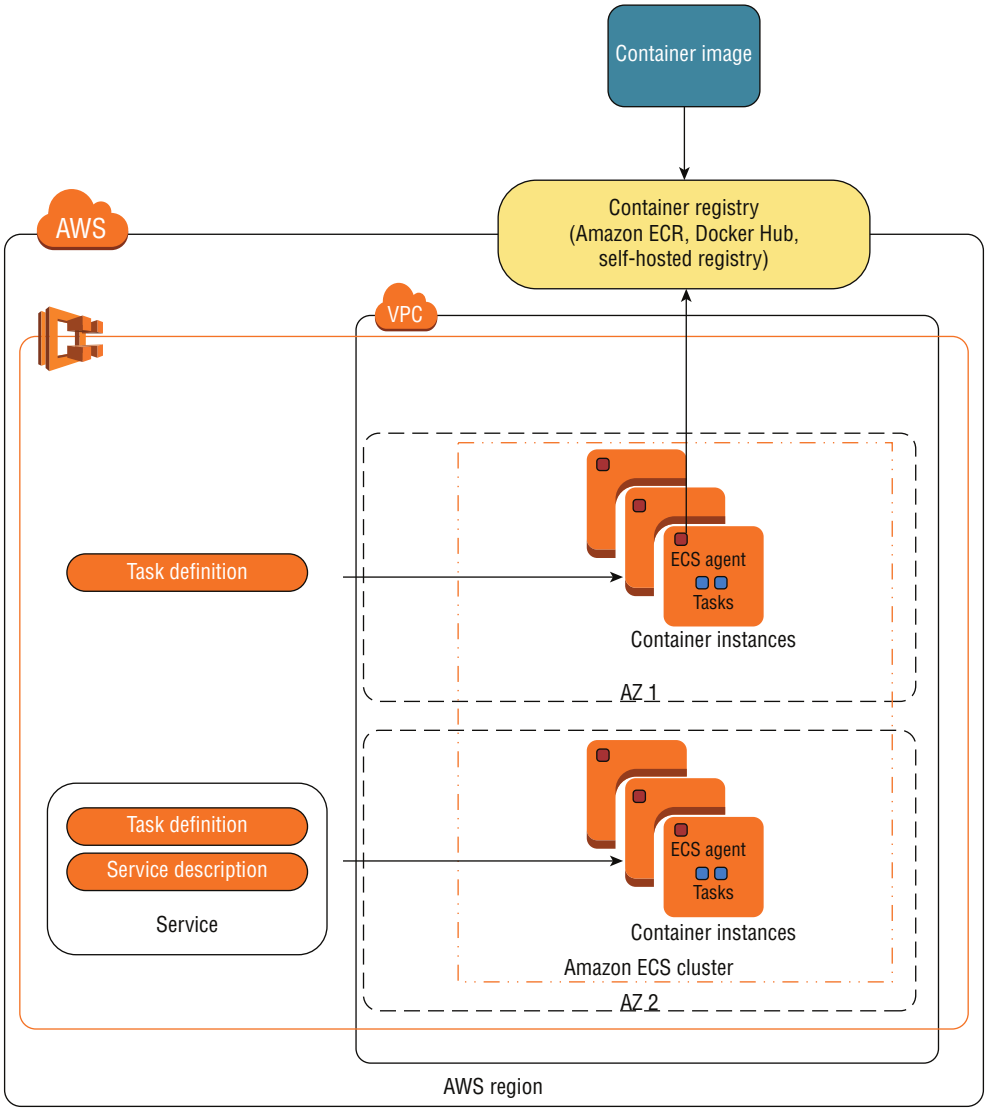
Amazon ECS Concepts

This section details Amazon ECS concepts.

Amazon ECS Cluster

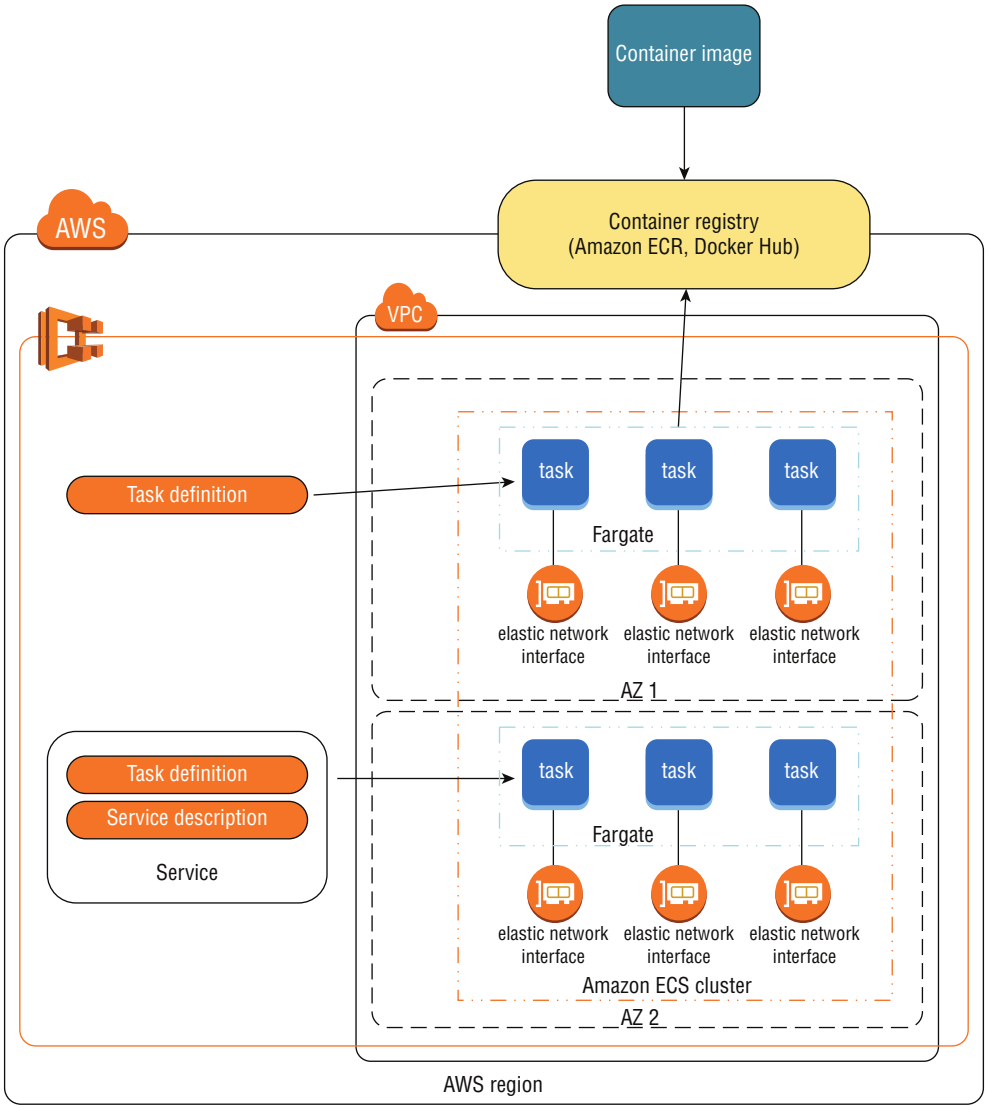
Amazon ECS clusters are the foundational infrastructure components on which containers run. Clusters consist of one or more Amazon EC2 instances in your Amazon VPC. Each instance in a cluster (cluster instance) has an agent installed. The agent is responsible for receiving container scheduling/shutdown commands from the Amazon ECS service and to report the current health status of containers (restart or replace). Figure 9.14 demonstrates an Amazon EC2 launch type, where instances make up the Amazon ECS cluster.

FIGURE 9.14 Amazon ECS architecture



In an AWS Fargate launch type, Amazon ECS clusters are no longer made up of Amazon EC2 instances. Since the tasks themselves launch on the AWS infrastructure, AWS assigns each one an elastic network interface with an Amazon VPC. This provides network connectivity for the container without the need to manage the infrastructure on which it runs. Figure 9.15 demonstrates an AWS Fargate cluster that runs in multiple availability zones (AZs).

FIGURE 9.15 AWS Fargate architecture



An individual cluster can support both Amazon EC2 and AWS Fargate launch types. However, a single cluster instance can belong to only one cluster at a time. Amazon EC2 launch types support both on-demand and spot instances, and they allow you to reduce cost for noncritical workloads.

To enable network connectivity for containers that run on your instance, the corresponding task definition must outline port mappings from the container to the host

instance. When you create a container instance, you can select the instance type to use. The compute resources available to this instance type will determine how many containers can be run on the instance. For example, if a `t2.micro` instance has one vCPU and 1 GB of RAM, it will not be able to run containers that require two vCPUs.

After you add a container instance to a cluster and you place containers on it, there may be situations where you would need to remove the container from the cluster temporarily—for a regular patch, for example. However, if critical tasks run on a container instance, you may want to wait for the containers to terminate gracefully. Container instance draining can be used to drain running containers from an instance and prevent new ones from being started. Depending on the service's configuration, replacement tasks start before or after the original tasks terminate.

- If the value of `minimumHealthyPercent` is less than 100 percent, the service will terminate the task and launch a replacement.
- If the value is greater than 100 percent, the service will attempt to launch a replacement task before it terminates the original.

To make room for launching additional tasks, you can scale out a cluster with Amazon EC2 Auto Scaling groups. For an EC2 Auto Scaling group to work with an Amazon ECS cluster, you must install the Amazon ECS agent either as part of the AMI or via instance `userdata`. To change the number container instances that run, you can adjust the size of the corresponding EC2 Auto Scaling group. If you need to terminate instances, any tasks that run on them will also halt.

Scaling out a cluster does not also increase the running task count. You use service automatic scaling for this process.

AWS Fargate

AWS Fargate simplifies the process of managing containers in your environment and removes the need to manage underlying cluster instances. Instead, you only need to specify the compute requirements of your containers in your task definition. AWS Fargate automatically launches containers without your interaction.

With AWS Fargate, there are several restrictions on the types of tasks that you can launch. For example, when you specify a task definition, containers cannot be run in privileged mode. To verify that a given task definition is acceptable by AWS Fargate, use the **Requires** capabilities field of the Amazon ECS console or the `--requires-capabilities` command option of the AWS CLI.

AWS Fargate requires that containers launch with the network mode set to `awsvpc`. In other words, you can launch only AWS Fargate containers into Amazon VPCs.

AWS Fargate requires the `awslogs` driver to enable log configuration.

Containers and Images

Amazon ECS launches and manages Docker containers. However, Docker is not in scope for the AWS Certified Developer – Associate Exam.

Any workloads that run on Amazon ECS must reside in Docker containers. In a virtual server environment, multiple virtual machines share physical hardware, each of which acts as its own operating system. In a containerized environment, you package components of the operating system itself into containers. This removes the need to run any nonessential aspects of a full-fledged virtual machine to increase portability. In other words, virtual machines share the same physical hardware, while containers share the same operating system.

Container images are similar in concept to AMIs. Images provision a Docker container. You store images in registries, such as a Docker Hub or an Amazon Elastic Container Repository (ECR).

You can create your own private image repository; however, AWS Fargate does not support this launch type.

Docker provides mobility and flexibility of your workload to allow containers to be run on any system that supports Docker. Compute resources can be better utilized when you run multiple containers on the same cluster, which makes the best possible use of resources and reduces idle compute capacity. Since you separate service components into containers, you can update individual components more frequently and at reduced risk.

Task Definition

Though you can package entire applications into a single container, it may be more efficient to run multiple smaller containers, each of which contains a subset of functionality of your full application. This is referred to as *service-oriented architecture* (SOA). In SOA, each unit of functionality for an overall system is contained separately from the rest. Individual services work with one another to perform a larger task. For example, an e-commerce website that uses SOA could have sets of containers for load balancing, credit card processing, order fulfillment, or any other tasks that users require. You design each component of the system as a black box so that other components do not need to be aware of inner workings to interact with them.

A *task definition* is a JSON document that describes what containers launch for your application or system. A single task definition can describe between one and 10 containers and their requirements. Task definitions can also specify compute, networking, and storage

requirements, such as which ports to expose to which containers and which volumes to mount.

You should add containers to the same task definition under the following circumstances:

- The containers all share a common lifecycle.
- The containers need to run on the same common host or container instance.
- The containers need to share local resources or volumes.

An entire application does not need to deploy with a single task definition. Instead, you should separate larger application segments into separate task definitions. This will reduce the impact of breaking changes in your environment. If you allocate the right-sized container instances, you can also better control scaling and resource consumption of the containers.

After a task definition creates and uploads to Amazon ECS, it can launch one or more *tasks*. When a task is created, the containers in the task definition are scheduled to launch into the target cluster via the task scheduler.

Task Definition with Two Containers

The following example demonstrates a task definition with two containers. The first container runs a WordPress installation and binds the container instance's port 80 to the same port on the container. The second container installs MySQL to act as the backend data store of the WordPress container. The task definition also specifies a link between the containers, which allows them to communicate without port mappings if the network setting for the task definition is set to `bridge`.

```
{
  "containerDefinitions": [
    {
      "name": "wordpress",
      "links": [
        "mysql"
      ],
      "image": "wordpress",
      "essential": true,
      "portMappings": [
        {
          "containerPort": 80,
          "hostPort": 80
        }
      ],
```

(continued)

(continued)

```
      "memory": 500,
      "cpu": 10
    },
    {
      "environment": [
        {
          "name": "MYSQL_ROOT_PASSWORD",
          "value": "password"
        }
      ],
      "name": "mysql",
      "image": "mysql",
      "cpu": 10,
      "memory": 500,
      "essential": true
    }
  ],
  "family": "hello_world"
}
```

Services

When creating a *service*, you can specify the task definition and number of tasks to maintain at any point in time. After the service creates, it will launch the desired number of tasks; thus, it launches each of the containers in the task definition. If any containers in the task become unhealthy, the service is responsible and launches replacement tasks.

Deployment Strategies

When you define a service, you can also configure deployment strategies to ensure a minimum number of healthy tasks are available to serve requests while other tasks in the service update. The `maximumPercent` parameter defines the maximum percentage of tasks that can be in `RUNNING` or `PENDING` state. The `minimumHealthyPercent` parameter specifies the minimum percentage of tasks that must be in a healthy (`RUNNING`) state during deployments.

Suppose you configure one task for your service, and you would like to ensure that the application is available during deployments. If you set the `maximumPercent` to 200 percent and `minimumHealthyPercent` to 100 percent, it will ensure that the new task launches before the old task terminates. If you configure two tasks for your service and some loss of availability is acceptable, you can set `maximumPercent` to 100 percent and `minimumHealthyPercent` to 50 percent. This will cause the service scheduler to terminate one task, launch its replacement, and then do the same with the other task. The difference is that the first approach requires double the normal cluster capacity to accommodate the additional tasks.

Balance Loads

You can configure services to run behind a load balancer to distribute traffic automatically to tasks in the service. Amazon ECS supports classic load balancers, application load balancers, and network load balancers to distribute requests. Of the three load balancer types, application load balancers provide several unique features.

Application Load Balancing (ALB) load balancers route traffic at layer 7 (HTTP/HTTPS). Because of this, they can take advantage of dynamic host port mapping when you use them in front of Amazon ECS clusters. ALBs also support path-based routing so that multiple services can listen on the same port. This means that requests will be to different tasks based on the path specified in the request.

Classic load balancers, because they register and deregister instances, require that any tasks being run behind the load balancer all exist on the same container instance. This may not be desirable in some cases, and it would be better to use an ALB.

Schedule Tasks

If you increase the number of instances in an Amazon ECS cluster, it does not automatically increase the number of running tasks as well. When you configure a service, the service scheduler determines how many tasks run on one or more clusters and automatically starts replacement tasks should any fail. This is especially ideal for long-running tasks such as web servers. If you configure it to do so, the service scheduler will ensure that tasks register with an elastic load balancer.

You can also run a task manually with the `RunTask` action, or you can run tasks on a `cron`-like schedule (such as every *N* minutes on Tuesdays and Thursdays). This works well for tasks such as log rotation, batch jobs, or other data aggregation tasks.

To dynamically adjust the run task count dynamically, you use Amazon CloudWatch Alarms in conjunction with Application Auto Scaling to increase or decrease the task count based on alarm status. You can use two approaches for automatically scaling Amazon ECS services and tasks: Target Tracking Policies and Step Scaling Policies.

Target Tracking Policies

Target tracking policies determine when to scale the number of tasks based on a target metric. If the metric is above the target, such as CPU utilization being above 75 percent, Amazon ECS can automatically launch more tasks to bring the metric below the desired value. You can specify multiple target tracking policies for the same service. In the case of a conflict, the policy that would result in the highest task count wins.

Step Scaling Policies

Unlike target tracking policies, *step scaling policies* can continue to scale in or out as metrics increase or decrease. For example, you can configure a step scaling policy to scale out when CPU utilization reaches 75 percent, again at 80 percent, and one final time at 90 percent. With this approach, a single policy can result in multiple scaling activities as metrics increase or decrease.

Task Placement Strategies

Regardless of the method you use, *task placement strategies* determine on which instances tasks launch or which tasks terminate during scaling actions. For example, the `spread` task placement strategy distributes tasks across multiple AZs as much as possible. Task placement strategies perform on a best-effort basis. If the strategy cannot be honored, such as when there are insufficient compute resources in the AZ you select, Amazon ECS will still try to launch the task(s) on other cluster instances. Other strategies include `binpack` (uses CPU and memory on each instance at a time) and `random`.

Task placement strategies associate with specific `attributes`, which are evaluated during task placement. For example, to spread tasks across availability zones, the placement strategy to use is as follows:

```
"placementStrategy": [
    {
        "field": "attribute:ecs.availability-zone",
        "type": "spread"
    }
]
```

Task Placement Constraints

Task placement constraints enforce specific requirements on the container instances on which tasks launch, such as to specify the instance type as `t2.micro`.

```
"placementConstraints": [
    {
        "expression": "attribute:ecs.instance-type == t2.micro",
        "type": "memberOf"
    }
]
```

Amazon ECS Service Discovery

Amazon ECS Service Discovery allows you to assign Amazon Route 53 DNS entries automatically for tasks your service manages. To do so, you create a private service namespace for each Amazon ECS cluster. As tasks launch or terminate, the private service namespace updates to include DNS entries for each task. A service directory maps DNS entries to available service endpoints. Amazon ECS Service Discovery maintains health checks of containers, and it removes them from the service directory should they become unavailable.

To use public namespaces, you must purchase or register the public hosted zone with Amazon Route 53.

Private Image Repositories

Amazon ECS can connect to private image repositories with basic authentication. This is useful to connect to Docker Hub or other private registries with a username and password. To do so, the `ECS_ENGINE_AUTH_TYPE` and `ECS_ENGINE_AUTH_DATA` environment variables must be set with the authorization type and actual credentials to connect. However, you should not set these properties directly. Instead, store your container instance configuration file in an Amazon S3 bucket and copy it to the instance with `userdata`.

Amazon Elastic Container Repository

Amazon Elastic Container Repository (Amazon ECR) is a Docker registry service that is fully compatible with existing Docker CLI tools. Amazon ECR supports resource-level permissions for private repositories and allows you to preserve a secure registry without the need to maintain an additional application. Since it integrates with IAM users and Amazon ECS cluster instances, it can take advantage of IAM users or instance profiles to access and maintain images securely without the need to provide a username and password.

Amazon ECS Container Agent

The *Amazon ECS container agent* is responsible for monitoring the status of tasks that run on cluster instances. If a new task needs to launch, the container agent will download the container images and start or stop containers. If any containers fail health checks, the container agent will replace them. Since the AWS Fargate launch type uses AWS-managed compute resources, you do not need to configure the agent.

To register an instance with an Amazon ECS cluster, you must first install the Amazon ECS Agent. This agent installs automatically on Amazon ECS optimized AMIs. If you would like to use a custom AMI, it must adhere to the following requirements:

- Linux kernel 3.10 or greater
- Docker version 1.9.0 or greater and any corresponding dependencies

The Amazon ECS container agent updates regularly and can update on your instance(s) without any service interruptions. To perform updates to the agent, replace the container instance entirely or use the Update Container Agent command on Amazon ECS optimized AMIs.

You cannot perform agent updates on Windows instances using these methods. Instead, terminate the instance and create a new server in its absence.

To configure the Amazon ECS container agent, update `/etc/ecs/config` on the container instance and then restart the agent. You can configure properties such as the cluster to register with, reserved ports, proxy settings, and how much system memory to reserve for the agent.

Amazon ECS Service Limits

Table 9.4 displays the limits that AWS enforces for Amazon ECS. You can change limits with an asterisk (*) by making a request to AWS Support.

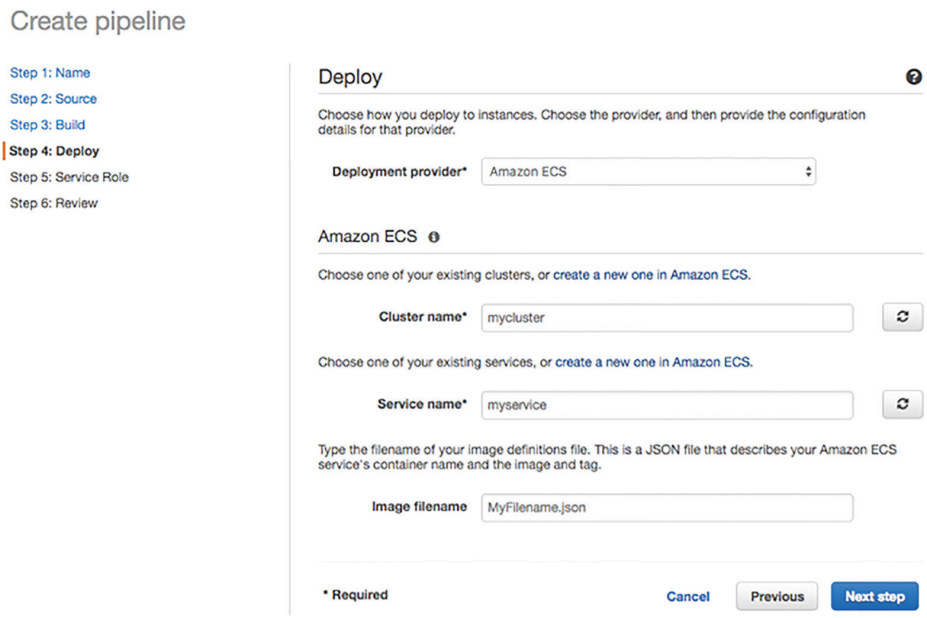
TABLE 9.4 Amazon ECS Service Limits

Limit	Value
Clusters per region per account*	1,000
Container instances per cluster*	1,000
Services per cluster*	500
Tasks that use Amazon EC2 launch type per service*	1,000
Tasks that use AWS Fargate launch type per region per account*	20
Public IP addresses for tasks that use AWS Fargate launch type*	20
Load balancers per service	1
Task definition size	32 KiB
Task definition containers	10
Layer size of image that use AWS Fargate task	4 GB
Shared volume that use AWS Fargate tasks	10 GB
Container storage that use AWS Fargate tasks	10 GB

Using Amazon ECS with AWS CodePipeline

When you select Amazon ECS as a deployment provider, there is no option to create the cluster and service as part of the pipeline creation process. This must be done ahead of time. After the cluster is created, select the appropriate cluster and service names in the AWS CodePipeline console, as shown in Figure 9.16.

FIGURE 9.16 Amazon ECS as a deployment provider



You must provide an image filename as part of this configuration. This is a JSON-formatted document inside your code repository or archive or as an output build artifact, which specifies the service's container name and image tag. We recommend that the cluster contain at least two Amazon EC2 instances so that one can act as primary while the other handles deployment of new containers.

Summary

This chapter includes infrastructure, configuration, and deployment services that you use to deploy configuration as code.

AWS CloudFormation leverages standard AWS APIs to provision and update infrastructure in your account. AWS CloudFormation uses standard configuration management tools such as Chef and Puppet.

Configuration management of infrastructure over an extended period of time is best served with the use of a dedicated tool such as AWS OpsWorks Stacks. You define the configuration in one or more Chef recipes to achieve configuration as code on top of your infrastructure. AWS OpsWorks Stacks can be used to provide a serverless Chef infrastructure to configure servers with Chef code (recipes).

Chef recipe code is declarative in nature, and you do not have to rely on the accuracy of procedural steps, as you would with a `userdata` script you apply to Amazon ECS instances or launch configurations. You can use Amazon ECS instead of instances or serverless functions to use a containerization method to manage applications. If you separate infrastructure from configuration, you also gain the ability to update each on separate cadences.

Amazon ECS supports Docker containers, and it allows you to run and scale containerized applications on AWS. Amazon ECS eliminates the need to install and operate your own container orchestration software, manage and scale a cluster of virtual machines, or schedule containers on those virtual machines.

AWS Fargate reduces management further as it deploys containers to serverless architecture and removes cluster management requirements. To create a cluster and deploy services, you configure the resource requirements of containers and availability requirements. Amazon ECS manages the rest through an agent that runs on cluster instances. AWS Fargate requires no agent management.

Amazon ECS clusters are the foundational infrastructure components on which containers run. Clusters consist of Amazon EC2 instances in your Amazon VPC. Each cluster instance has an agent installed that is responsible for receiving scheduling/shutdown commands from the Amazon ECS service and reporting the current health status of containers (restart or replace).

In lieu of custom JSON, Chef 12.0 stacks support data bags to provide better compatibility with community cookbooks. You can declare data bags in the custom JSON field of the stack, layer, and deployment configurations to provide instances in your stack for any additional data that you would like to provide.

AWS OpsWorks Stacks lets you manage applications and servers on AWS and on-premises. You can model your application as a stack that contains different layers, such as load balancing, database, and application server. You can deploy and configure Amazon EC2 instances in each layer or connect other resources such as Amazon RDS databases. AWS OpsWorks Stacks lets you set automatic scaling for your servers on preset schedules or in response to a constant change of traffic levels, and it uses lifecycle hooks to orchestrate changes as your environment scales. You run Chef recipes with Chef Solo, which allows you to automate tasks such as installing packages and program languages or frameworks, configuring software, and more.

An app is the location where you store application code and other files, such as an Amazon S3 bucket, a Git repository, or an HTTP bundle, and it includes sign-in credentials. The `Deploy` lifecycle event includes any apps that you configure for an instance at the layer or layers to which it corresponds.

At each layer of a stack, you set which Chef recipes to execute at each stage of a node's lifecycle, such as when it comes online or goes offline (lifecycle events). The recipes at each lifecycle event are executed by the AWS OpsWorks Agent in the order you specify.

AWS OpsWorks Stacks allows for management of other resources in your account as part of your stack and include elastic IP addresses, Amazon EBS volumes, and Amazon RDS instances.

The AWS OpsWorks Stacks dashboard monitors up to 13 custom metrics for each instance in the stack. The agent that runs on each instance will publish the information to the AWS OpsWorks Stacks service. If you enable the layer, system, application, and custom logs, they automatically publish to Amazon CloudWatch Logs for review without accessing the instance itself.

When you define a consistent deployment pattern for infrastructure, configuration, and application code, you can convert entire enterprises to code. You can remove manual management of most common processes and replace them with seamless management of entire application stacks through a simple commit action.

Exam Essentials

Understand configuration management and Chef. Configuration management is the process designed to ensure the infrastructure in a given system adheres to a specific set of standards, settings, or attributes. Chef is a Ruby-based configuration management language that AWS OpsWorks Stacks uses to enforce configuration on Amazon EC2 on-premises instances, or *nodes*. Chef uses a declarative syntax to describe the desired state of a node, abstracting the actual steps needed to achieve the desired configuration. This code is organized into *recipes*, which are organized into collections called *cookbooks*.

Know how AWS OpsWorks Stacks organizes configuration code into cookbooks. In traditional Chef implementations, cookbooks belong to a `chef-repo`, which is a versioned directory that contains cookbooks and their underlying recipes and files. A single cookbook repository can contain one or more cookbooks. When you define the custom cookbook location for a stack, all cookbooks copy to instances in the stack.

Know how to update custom cookbooks on a node. When instances first launch in a stack, they will download cookbooks from the custom cookbook repository. You must manually issue an `Update Custom Cookbooks` command to instances in your stack to update the instance.

Understand the different AWS OpsWorks Stacks components. The topmost object in AWS OpsWorks Stacks is a stack, which contains all elements of a given environment or system. Within a stack, one or more layers contain instances you group by common purpose. A single instance references either an Amazon EC2 or on-premises instance and contains additional configuration data. A stack can contain one or more apps, which refer to repositories where application code copies to for deployment. Users are regional resources that you can configure to access one or more stacks in an account.

Know the different AWS OpsWorks Stacks instance types and their purpose. AWS OpsWorks Stacks has three different instance types: 24/7, time-based, and load-based. The 24/7 instances run continuously unless an authorized user manually stops it, and they are useful for handling the minimum expected load of a system. Time-based instances start and stop on a given 24-hour schedule and are recommended for predicable increases in load at

different times of the day. Load-based instances start and stop in response to metrics, such as CPU utilization for a layer, and you use them to respond to sudden increases in traffic.

Understand how AWS OpsWorks Stacks implements auto healing. The AWS OpsWorks Stacks agent that runs on an instance performs a health check every minute and sends the response to AWS. If the AWS OpsWorks Stacks agent does not receive the health check for five continuous minutes, the instance restarts automatically. You can disable this feature. Auto healing events publish to Amazon CloudWatch for reference.

Understand the AWS OpsWorks Stacks permissions model. AWS OpsWorks Stacks provides the ability to manage users at the stack level, independent of IAM permissions. This is useful for providing access to instances in a stack but not to the AWS Management Console or API. You can assign AWS OpsWorks Stacks users to one of four permission levels: `Deny`, `Show`, `Deploy`, and `Manage`. Additionally, you can give users SSH/RDP access to instances in a stack (with or without sudo/administrator permission). AWS OpsWorks Stacks users are regional resources. If you would like to give a user in one region access to a stack in another region, you need to copy the user to the second region. Some AWS OpsWorks Stacks activities are available only through IAM permissions, such as to delete and create stacks.

Know the different AWS OpsWorks Stacks lifecycle events. Instances in a stack are provisioned, configured, and retired using lifecycle events. The AWS OpsWorks Stacks supports the lifecycle events: `Setup`, `Configure`, `Deploy`, `Undeploy`, and `Shutdown`. The `Configure` event runs on all instances in a stack any time one instance comes online or goes offline.

Know the components of an Amazon ECS cluster. A cluster is the foundational infrastructure component on which containers are run. Clusters are made up of one or more Amazon EC2 instances, or they can be run on AWS-managed infrastructure using AWS Fargate. A task definition is a JSON file that describes which containers to launch on a cluster. Task definitions can be defined by grouping containers that are used for a common purpose, such as for compute, networking, and storage requirements. A service launches on a cluster and specifies the task definition and number of tasks to maintain. If any containers become unhealthy, the service is responsible for launching replacements.

Know the difference between Amazon ECS and AWS Fargate launch types. The AWS Fargate launch type uses AWS-managed infrastructure to launch tasks. As a customer, you are no longer required to provision and manage cluster instances. With AWS Fargate, each cluster instance is assigned a network interface in your VPC. Amazon ECS launch types require a cluster in your account, which you must manage over time.

Know how to scale running tasks in a cluster. Changing the number of instances in a cluster does not automatically cause the number of running tasks to scale in or out. You can use target tracking policies and step scaling policies to scale tasks automatically based on target metrics. A target tracking policy determines when to scale based on metrics such as CPU utilization or network traffic. Target tracking policies keep metrics within a certain boundary. For example, you can launch additional tasks if CPU utilization is above 75 percent. Step scaling policies can continuously scale as metrics increase or decrease. You can configure a step scaling policy to scale tasks out when CPU utilization reaches 75 percent

and again at 80 percent and 90 percent. A single step scaling policy can result in multiple scaling activities.

Know how images are stored in Amazon Elastic Container Repository (Amazon ECR). Amazon ECR is a Docker registry service that is fully compatible with existing Docker tools. Amazon ECR supports resource-level permissions for private repositories, and it allows you to maintain a secure registry without the need to maintain additional instances/applications.

Resources to Review

Continuous Deployment to Amazon ECS with AWS CodePipeline, AWS CodeBuild, Amazon ECR, and AWS CloudFormation:

`https://aws.amazon.com/blogs/compute/continuous-deployment-to-amazon-ecs-using-aws-codepipeline-aws-codebuild-amazon-ecr-and-aws-cloudformation/`

How to set up AWS OpsWorks Stacks auto healing notifications in Amazon CloudWatch Events:

`https://aws.amazon.com/blogs/mt/how-to-set-up-aws-opsworks-stacks-auto-healing-notifications-in-amazon-cloudwatch-events/`

Managing Multi-Tiered Applications with AWS OpsWorks:

`https://d0.awsstatic.com/whitepapers/managing-multi-tiered-web-applications-with-opsworks.pdf`

AWS OpsWorks Stacks:

`https://aws.amazon.com/opsworks/stacks/`

How do I implement a configuration management solution on AWS?:

`https://aws.amazon.com/answers/configuration-management/aws-infrastructure-configuration-management/`

Docker on AWS:

`https://d1.awsstatic.com/whitepapers/docker-on-aws.pdf`

What are Containers?

`https://aws.amazon.com/containers/`

Amazon Elastic Container Service (ECS):

`https://aws.amazon.com/ecs/`

JON BONSO AND KENNETH SAMONTE

AWS CERTIFIED DEVOPS ENGINEER PROFESSIONAL

Tutorials Dojo
Study Guide and Cheat Sheets

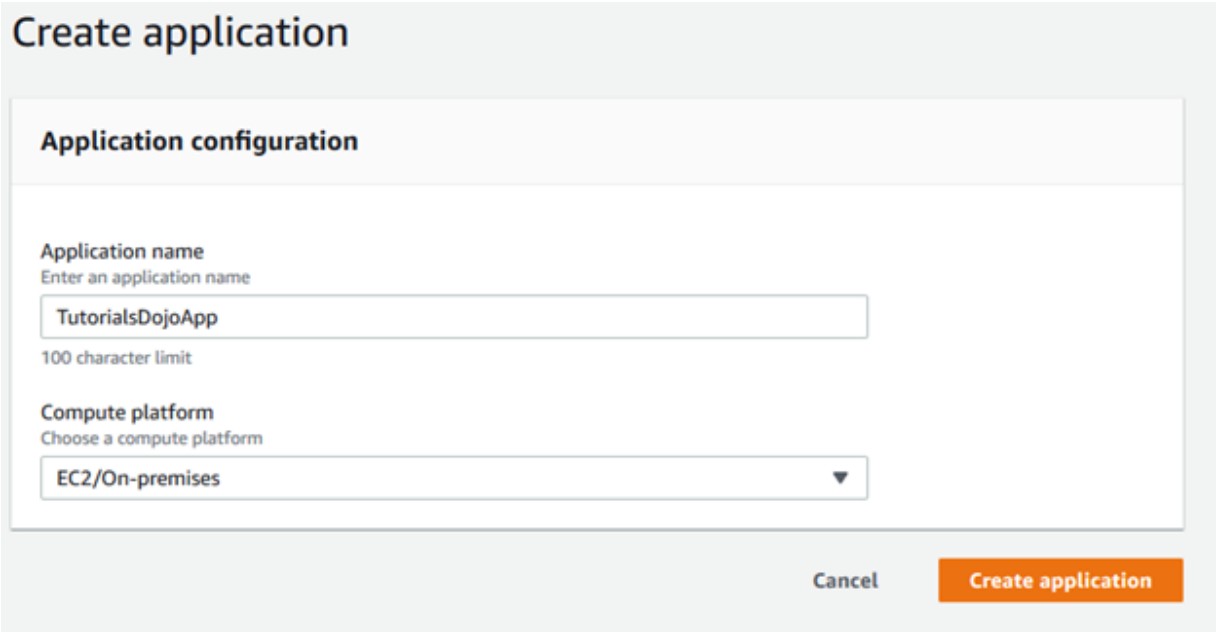
CodeDeploy with CloudWatch Logs, Metrics, and Alarms

AWS CodeDeploy allows you to automate software deployments to a variety of services such as EC2, AWS Fargate, AWS Lambda, and your on-premises servers.

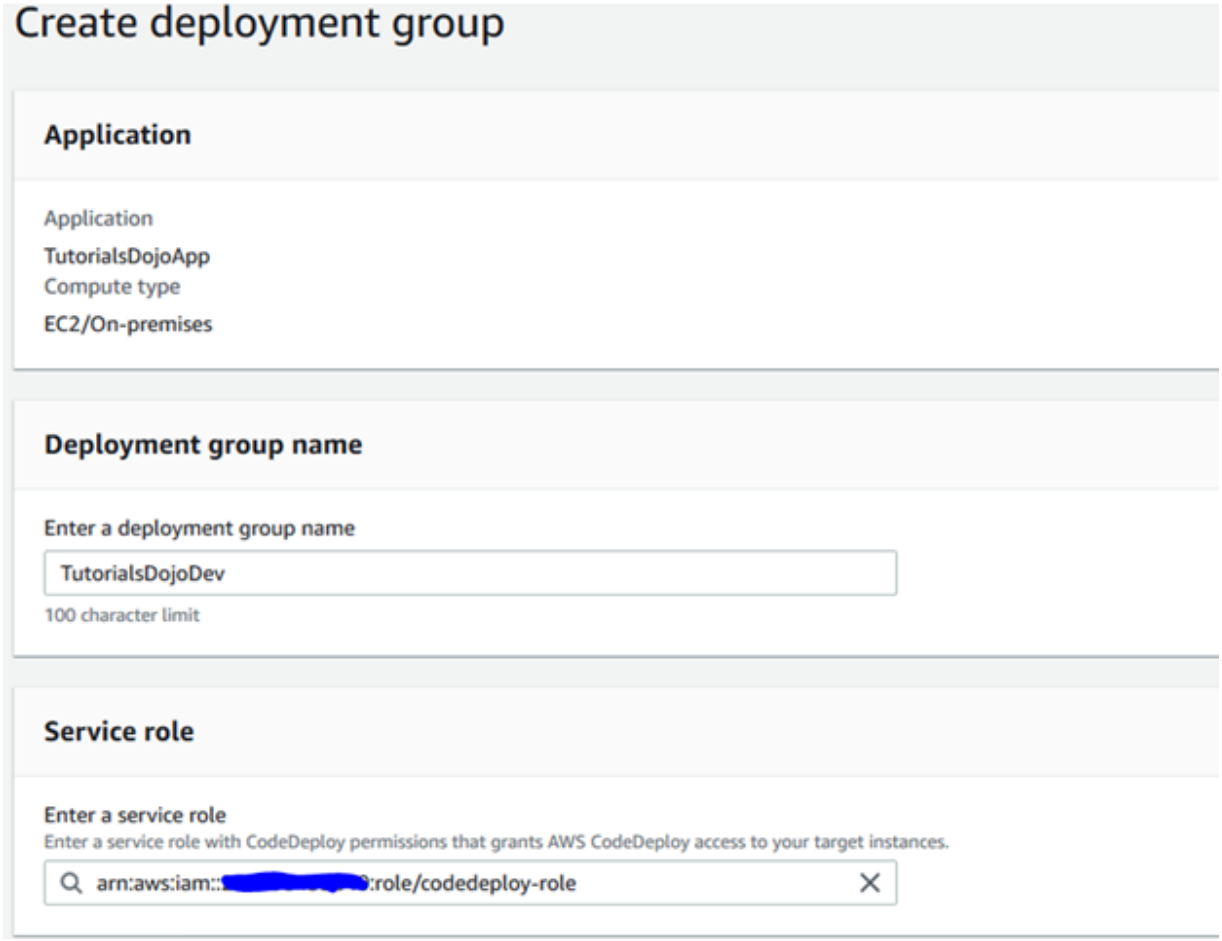
For example, after you have produced from AWS CodeBuild, you can have CodeDeploy fetch the artifact from the AWS S3 bucket and then deploy it to your AWS instances. Please note that the target instances need to have the AWS CodeDeploy agent installed on them for this to be successful and have proper Tags.

Here are the steps to create a deployment on AWS CodeDeploy, including a discussion on how it integrates with AWS CloudWatch.

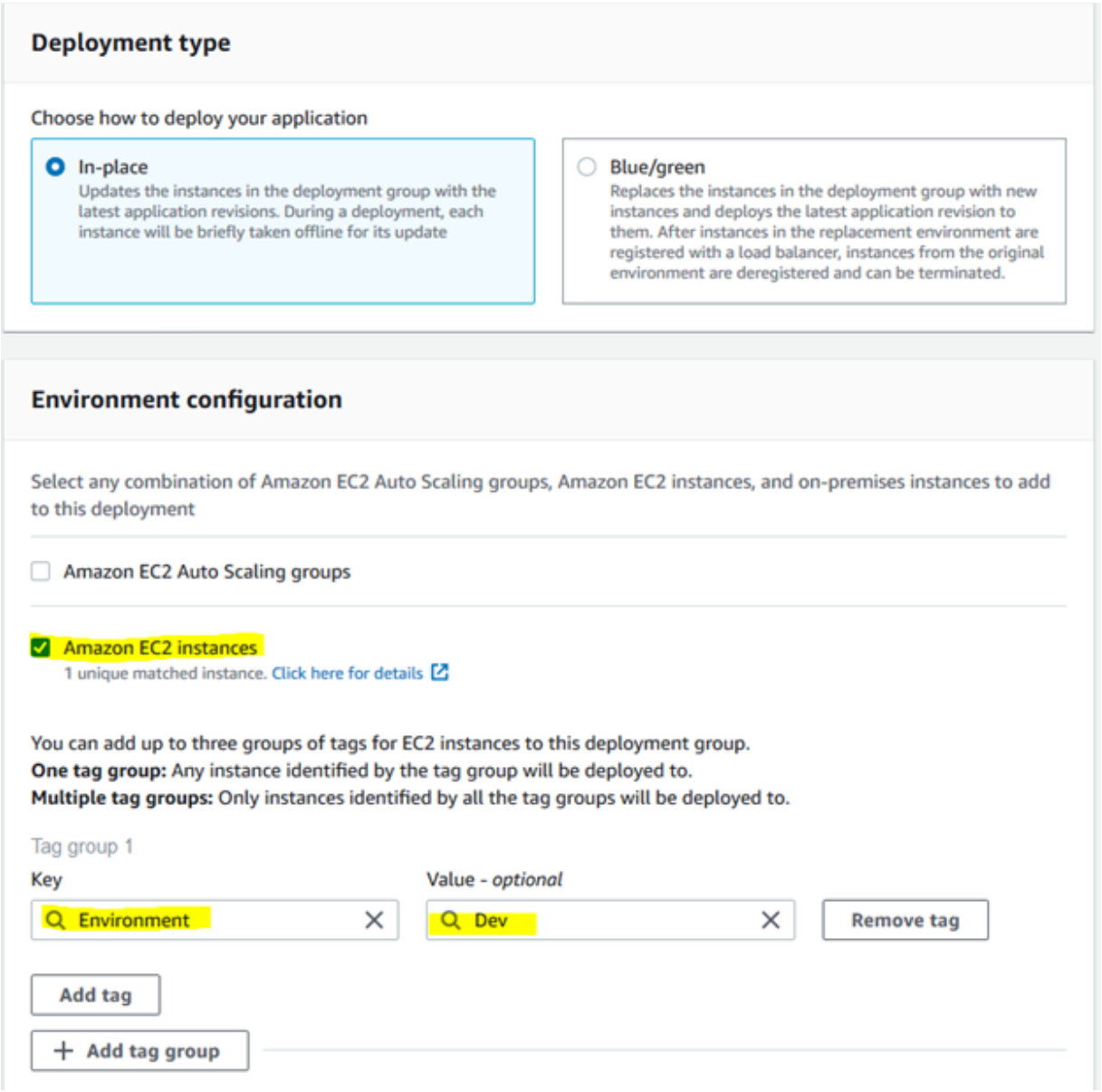
1. Go to AWS CodeDeploy > Applications and click “Create Application”. Input details of your application and which platform it is running.



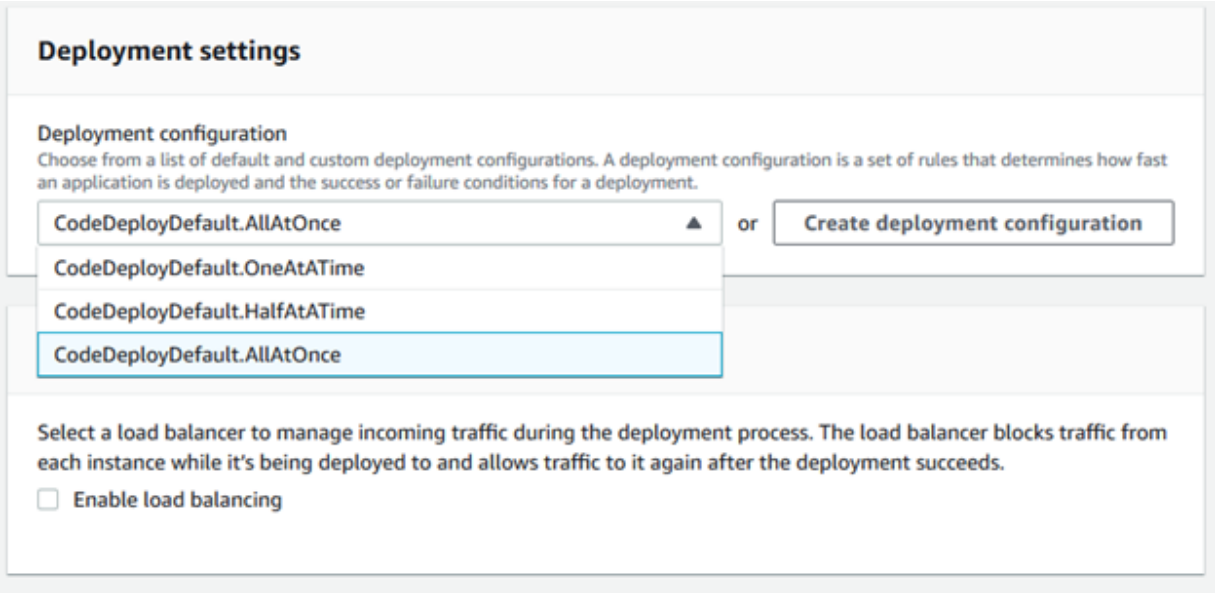
2. Select your application and create a Deployment Group. CodeDeploy needs to have an IAM permission to access your targets as well as read the AWS S3 bucket containing the artifact to be deployed.



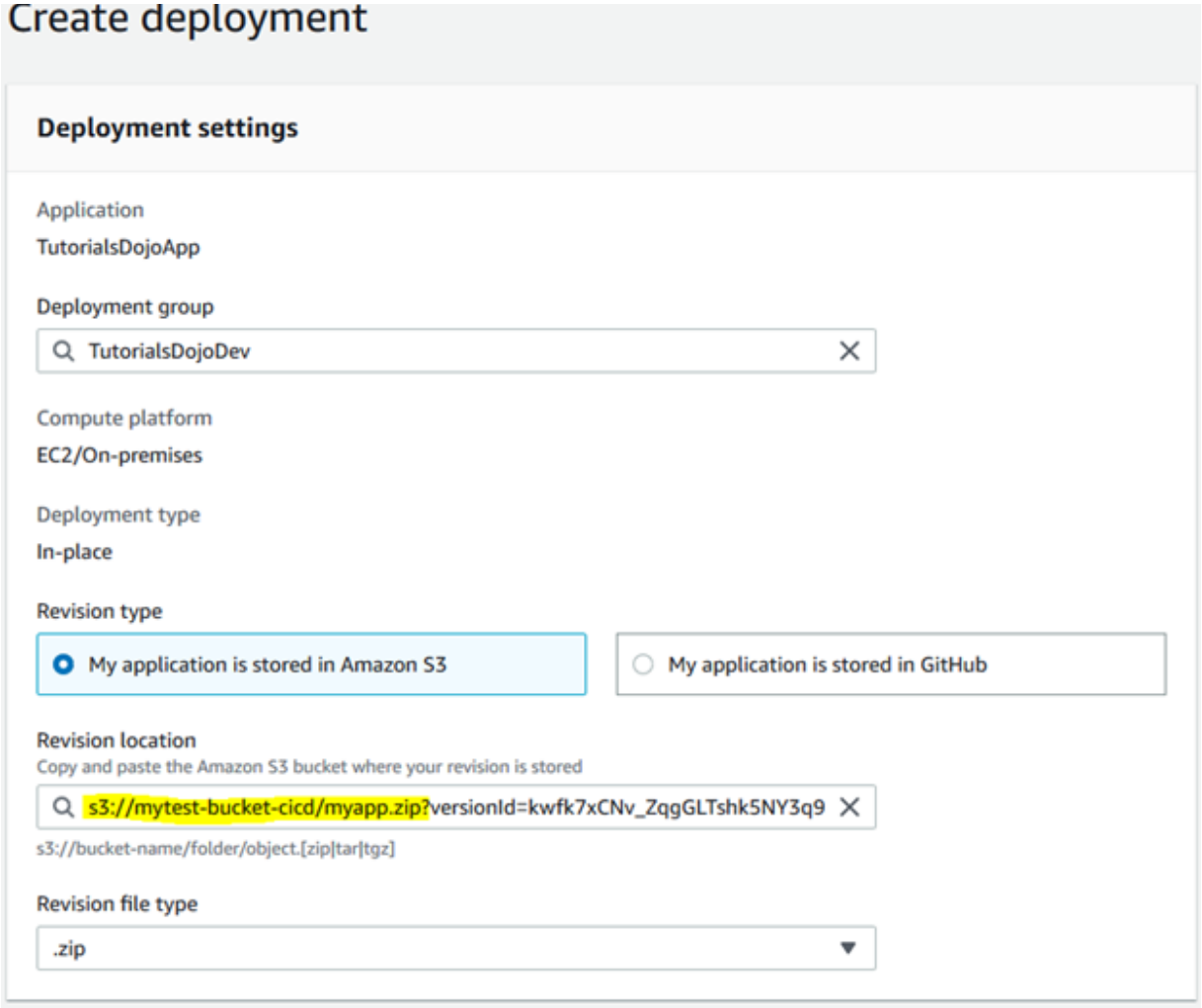
3. Select your deployment type. You can use In-place deployment if you already have the instances running. Specify the Tags of those EC2 instances, for example **Environment:Dev**. CodeDeploy will use this as an identifier for your target instances.



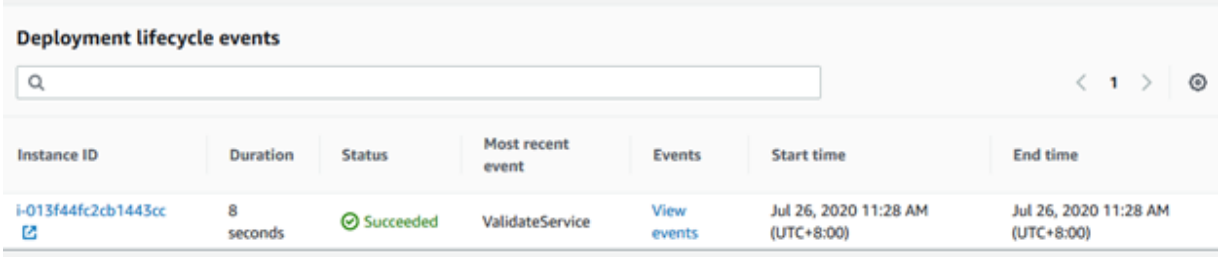
4. Select how you want the new code to be deployed, such as All-at-once, or one-at-a-time, etc. These deployment settings will be discussed on the succeeding sections.



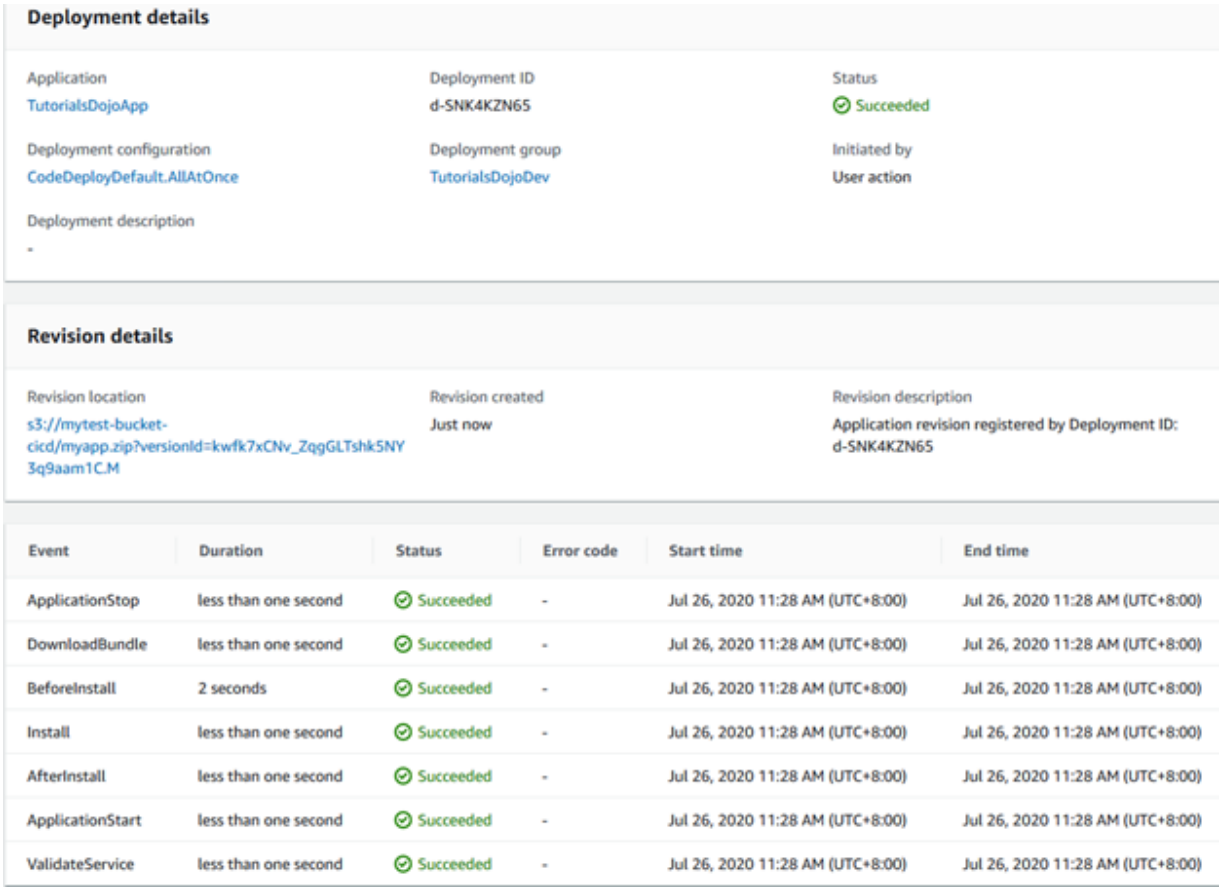
5. Now on your application Deployment group, create a Deployment. Input details for the artifact source. On the S3 bucket, you can specify the versionID of the artifact file.



6. After you click “Create deployment”, the deployment of the artifact on your EC2 instances will begin. You should see that the deployment will succeed.



7. You can click on “View events” to see the stages of the deployment process. Here you can view the deployment events as well as the status of the deployment lifecycle hooks you have defined. See the Lifecycle Event hooks section of this book for the details regarding each event.



Deployment details

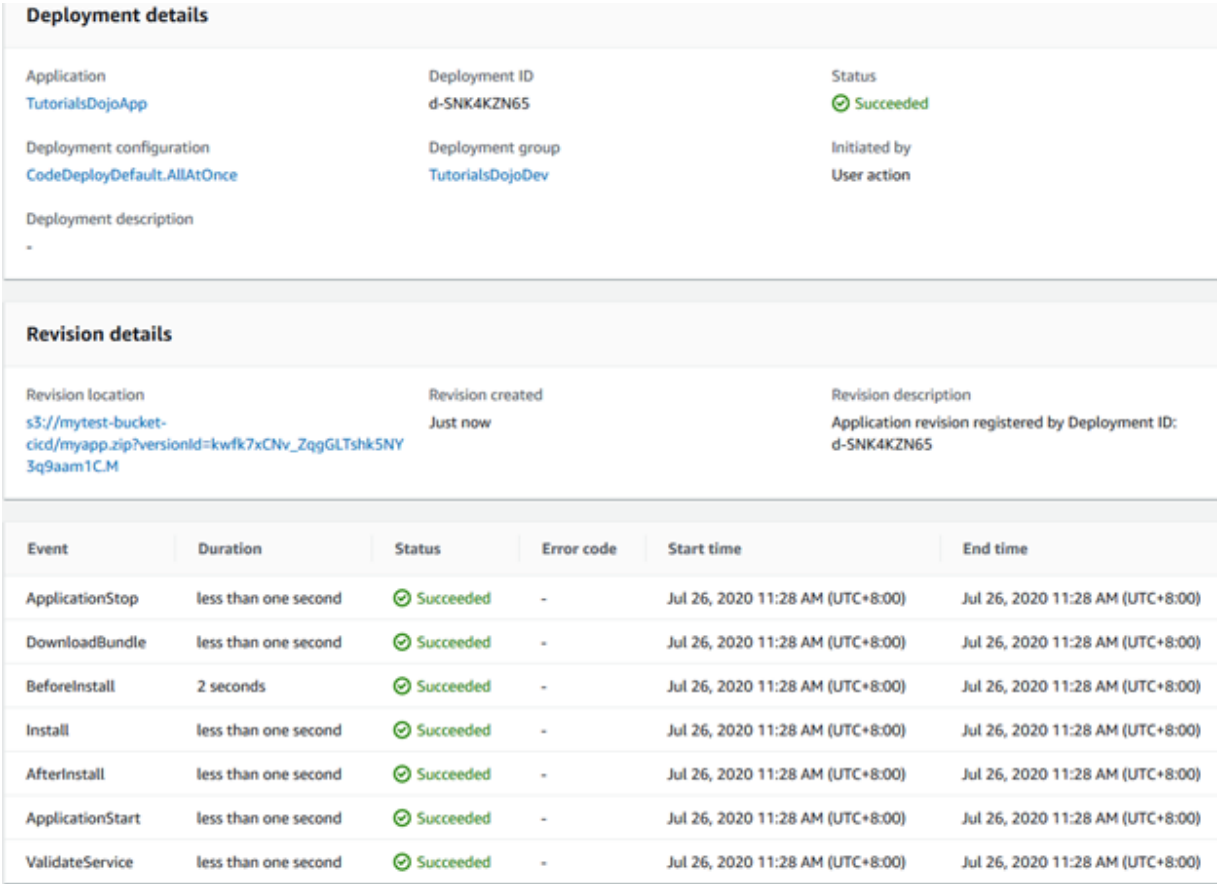
Application	Deployment ID	Status
TutorialsDojoApp	d-SNK4KZN65	Succeeded
Deployment configuration	**Deployment group**	**Initiated by**
CodeDeployDefault.AllAtOnce	TutorialsDojoDev	User action
Deployment description		
-		

Revision details

Revision location	Revision created	Revision description
s3://mytest-bucket-cicd/myapp.zip?versionId=kwfk7xCNv_ZqgGLTshk5NY3q9aam1C.M	Just now	Application revision registered by Deployment ID: d-SNK4KZN65

Event	Duration	Status	Error code	Start time	End time
ApplicationStop	less than one second	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)
DownloadBundle	less than one second	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)
BeforeInstall	2 seconds	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)
Install	less than one second	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)
AfterInstall	less than one second	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)
ApplicationStart	less than one second	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)
ValidateService	less than one second	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)

8. CodePipeline is integrated with AWS CloudWatch rule. You can create a CloudWatch Events rule that will detect CodeDeploy status changes, such as a Successful or Failed deployment. Then have it invoke a Lambda function to perform a custom action such as sending a notification on a Slack channel or setting an SNS topic to send an email to you about the status of your deployment.



Deployment details

Application	Deployment ID	Status
TutorialsDojoApp	d-SNK4KZN65	Succeeded
Deployment configuration	**Deployment group**	**Initiated by**
CodeDeployDefault.AllAtOnce	TutorialsDojoDev	User action

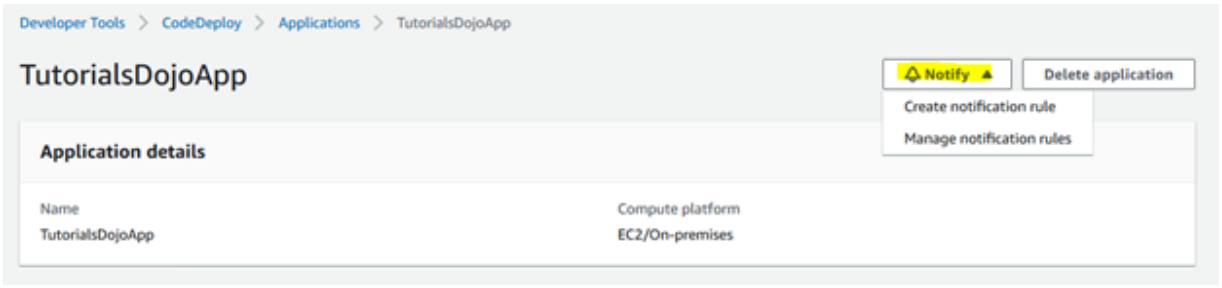
Deployment description
-

Revision details

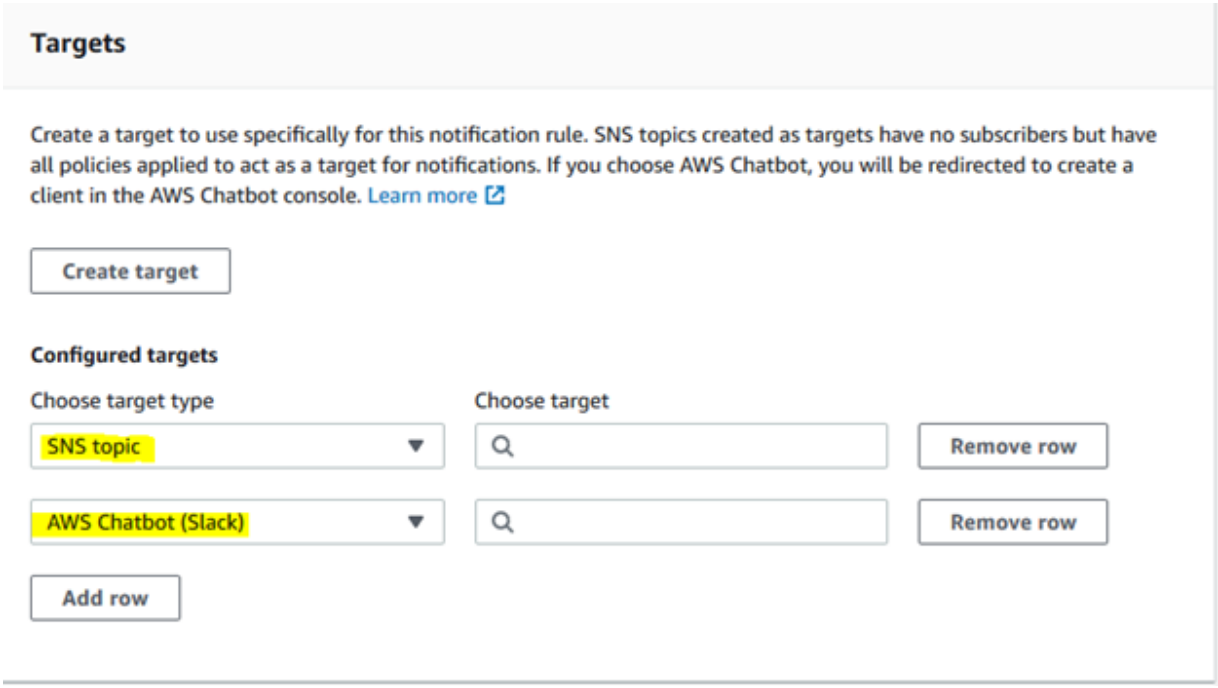
Revision location	Revision created	Revision description
s3://mytest-bucket-cicd/myapp.zip?versionId=kwfk7xCNv_ZqgGLTshk5NY3q9aam1C.M	Just now	Application revision registered by Deployment ID: d-SNK4KZN65

Event	Duration	Status	Error code	Start time	End time
ApplicationStop	less than one second	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)
DownloadBundle	less than one second	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)
BeforeInstall	2 seconds	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)
Install	less than one second	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)
AfterInstall	less than one second	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)
ApplicationStart	less than one second	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)
ValidateService	less than one second	Succeeded	-	Jul 26, 2020 11:28 AM (UTC+8:00)	Jul 26, 2020 11:28 AM (UTC+8:00)

9. CodeDeploy also has built-in notification triggers. Click “Notify” on your application.



10. For events happening on your deployment, you can have targets much like AWS CloudWatch Events, however, this is limited to only an SNS Topic or AWS chatbot to Slack.



DevOps Exam Notes:

AWS CodePipeline is integrated with AWS CloudWatch Events. You can create a CloudWatch Events rule that will detect CodeDeploy status changes, such as a Successful or Failed deployment. With the rule targets, you invoke a Lambda function to perform a custom action i.e. set an SNS topic to send an email to you about the status of your deployment.

CodeDeploy also has built-in notification triggers to notify you of your deployment status, however, this is limited to only an SNS Topic or AWS Chatbot to Slack.

CodeDeploy Supports ECS and Lambda Deployments

Aside from EC2 instances, AWS CodeDeploy also supports deployment for ECS instances and Lambda deployments. The general steps for deployment are still the same - create an **Application**, create a **deployment group** for your instances, and create a **deployment** for your application.

However, for ECS, the artifact is a Docker image which can be stored from AWS ECR or from DockerHub. For Lambda deployments, the artifact will come from a zip file from an S3 bucket. Be sure to have the proper filename of your Lambda handler file for successful code deployments.

Deployments for ECS also support deployment configuration such as all-at-once, one-at-a-time, half-of-the-time. These deployment configurations will be discussed on a separate section. Lambda on the other hand supports a percentage of traffic shifting such as linear or canary deployment. This is also discussed in a separate section.

Sources:
https://docs.aws.amazon.com/codedeploy/latest/userguide/monitoring-create-alarms.html
https://docs.aws.amazon.com/codedeploy/latest/userguide/monitoring.html
https://docs.aws.amazon.com/codedeploy/latest/userguide/deployments-create-console-ecs.html
https://docs.aws.amazon.com/codedeploy/latest/userguide/deployments-create-console-lambda.html

CloudFormation Template for ECS, Auto Scaling and ALB

Amazon Elastic Container Service (ECS) allows you to manage and run Docker containers on clusters of EC2 instances. You can also configure your ECS to use Fargate launch type which eliminates the need to manage EC2 instances.

With CloudFormation, you can define your ECS clusters and tasks definitions to easily deploy your containers. For high availability of your Docker containers, ECS clusters are usually configured with an auto scaling group behind an application load balancer. These resources can also be declared on your CloudFormation template.

DevOps Exam Notes:

Going on to the exam, be sure to remember the syntax needed to declare your ECS cluster, Auto Scaling group, and application load balancer. The **AWS::ECS::Service** resource creates an ECS cluster and the **AWS::ECS::TaskDefinition** resource creates a task definition for your container. The **AWS::ElasticLoadBalancingV2::LoadBalance**r resource creates an application load balancer and the **AWS::AutoScaling::AutoScalingGroup** resource creates an EC2 auto scaling group.

AWS provides an example template which you can use to deploy a web application in an Amazon ECS container with auto scaling and application load balancer. Here's a snippet of the template with the core resources:

```
{
  "AWSTemplateFormatVersion":"2010-09-09",
  "Resources":{
      "ECSCluster":{
      "Type":"AWS::ECS::Cluster"

      },

……

      "taskdefinition":{
      "Type":"AWS::ECS::TaskDefinition",

      "Properties":{

……

      "ECSALB":{
      "Type":"AWS::ElasticLoadBalancingV2::LoadBalancer",
```

```
        "Properties":{

……

        "ECSAutoScalingGroup":{
        "Type":"AWS::AutoScaling::AutoScalingGroup",
        "Properties":{
        "VPCZoneIdentifier":{
              "Ref":"SubnetId"

        },
```

Sources:
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/quickref-ecs.html
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-ecs-service.html
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-ecs-service-loadbalancers.html

Amazon Elastic Container Service (ECS)

- A container management service to run, stop, and manage Docker containers on a cluster.
- ECS can be used to create a consistent deployment and build experience, manage, and scale batch and **Extract-Transform-Load** (ETL) workloads, and build sophisticated application architectures on a microservices model.

Features

- You can create ECS clusters within a new or existing VPC.
- After a cluster is up and running, you can define task definitions and services that specify which Docker container images to run across your clusters.

Components

- Containers and Images
  - Your application components must be architected to run in **containers** — containing everything that your software application needs to run: code, runtime, system tools, system libraries, etc.
  - Containers are created from a read-only template called an **image**.
- Task Components
  - **Task definitions** specify various parameters for your application. It is a text file, in JSON format, that describes one or more containers, up to a maximum of ten, that form your application.
  - Task definitions are split into separate parts:
    - Task family - the name of the task, and each family can have multiple revisions.
    - IAM task role - specifies the permissions that containers in the task should have.
    - Network mode - determines how the networking is configured for your containers.
    - Container definitions - specify which image to use, how much CPU and memory the container are allocated, and many more options.
- Tasks and Scheduling
  - A **task** is the instantiation of a task definition within a cluster. After you have created a task definition for your application, you can specify the number of tasks that will run on your cluster.
    - Each task that uses the Fargate launch type has its own isolation boundary and does not share the underlying kernel, CPU resources, memory resources, or elastic network interface with another task.
  - You can upload a new version of your application task definition, and the ECS scheduler automatically starts new containers using the updated image and stop containers running the previous version.
- Clusters
  - When you run tasks using ECS, you place them in a **cluster**, which is a logical grouping of resources.
  - Clusters can contain tasks using both the Fargate and EC2 launch types.

  - When using the Fargate launch type with tasks within your cluster, ECS manages your cluster resources.
  - Enabling managed Amazon ECS cluster auto scaling allows ECS to manage the scale-in and scale-out actions of the Auto Scaling group.
- Services
  - ECS allows you to run and maintain a specified number of instances of a task definition simultaneously in a cluster.
  - In addition to maintaining the desired count of tasks in your service, you can optionally run your service behind a load balancer.
  - There are two deployment strategies in ECS:
    - **Rolling Update**
      - This involves the service scheduler replacing the current running version of the container with the latest version.

    Blue/Green Deployment with AWS CodeDeploy
      - This deployment type allows you to verify a new deployment of a service before sending production traffic to it.
      - The service must be configured to use either an Application Load Balancer or Network Load Balancer.
- Container Agent (AWS ECS Agent)
  - The **container agent** runs on each infrastructure resource within an ECS cluster.
  - It sends information about the resource's current running tasks and resource utilization to ECS, and starts and stops tasks whenever it receives a request from ECS.
  - Container agent is only supported on Amazon EC2 instances.

AWS Fargate

- You can use Fargate with ECS to run containers without having to manage servers or clusters of EC2 instances.
- You no longer have to provision, configure, or scale clusters of virtual machines to run containers.
- Fargate only supports container images hosted on Elastic Container Registry (ECR) or Docker Hub.

Task Definitions for Fargate Launch Type

- Fargate task definitions require that the network mode is set to *awsvpc*. The *awsvpc* network mode provides each task with its own elastic network interface.
- Fargate task definitions only support the *awslogs* log driver for the log configuration. This configures your Fargate tasks to send log information to Amazon CloudWatch Logs.
- Task storage is **ephemeral**. After a Fargate task stops, the storage is deleted.

Monitoring

- You can configure your container instances to send log information to CloudWatch Logs. This enables you to view different logs from your container instances in one convenient location.
- With CloudWatch Alarms, watch a single metric over a time period that you specify, and perform one or more actions based on the value of the metric relative to a given threshold over a number of time periods.
- Share log files between accounts, monitor CloudTrail log files in real time by sending them to CloudWatch Logs.

AWS CodeDeploy

- A fully managed deployment service that automates software deployments to a variety of compute services such as Amazon EC2, AWS Fargate, AWS Lambda, and your on-premises servers.
- **Concepts**
  - An Application is a name that uniquely identifies the application you want to deploy. CodeDeploy uses this name, which functions as a container, to ensure the correct combination of revision, deployment configuration, and deployment group are referenced during a deployment.
  - Compute platform is the platform on which CodeDeploy deploys an application (EC2, ECS, Lambda, On-premises servers).
  - Deployment configuration is a set of deployment rules and deployment success and failure conditions used by CodeDeploy during a deployment.
  - Deployment group contains individually tagged instances, Amazon EC2 instances in Amazon EC2 Auto Scaling groups, or both.
    1. In an Amazon ECS deployment, a deployment group specifies the Amazon ECS service, load balancer, optional test listener, and two target groups. It also specifies when to reroute traffic to the replacement task set and when to terminate the original task set and ECS application after a successful deployment.
    2. In an AWS Lambda deployment, a deployment group defines a set of CodeDeploy configurations for future deployments of an AWS Lambda function.
    3. In an EC2/On-Premises deployment, a deployment group is a set of individual instances targeted for a deployment.
       - In an in-place deployment, the instances in the deployment group are updated with the latest application revision.
       - In a blue/green deployment, traffic is rerouted from one set of instances to another by deregistering the original instances from a load balancer and registering a replacement set of instances that typically has the latest application revision already installed.
  - A deployment goes through a set of predefined phases called deployment lifecycle events. A deployment lifecycle event gives you an opportunity to run code as part of the deployment.
    1. ApplicationStop
    2. DownloadBundle
    3. BeforeInstall
    4. Install
    5. AfterInstall
    6. ApplicationStart
    7. ValidateService
  - Features
    - CodeDeploy protects your application from downtime during deployments through rolling updates and deployment health tracking.
    - AWS CodeDeploy tracks and stores the recent history of your deployments.

- CodeDeploy is platform and language agnostic.
- CodeDeploy uses a file and command-based install model, which enables it to deploy any application and reuse existing setup code. The same setup code can be used to consistently deploy and test updates across your environment release stages for your servers or containers.
- CodeDeploy integrates with Amazon Auto Scaling, which allows you to scale EC2 capacity according to conditions you define such as traffic spikes. Notifications are then sent to AWS CodeDeploy to initiate an application deployment onto new instances before they are placed behind an Elastic Load Balancing load balancer.
- When using AWS CodeDeploy with on-premises servers, make sure that they can connect to AWS public endpoints.
- AWS CodeDeploy offers two types of deployments:
  - With in-place deployments, the application on each instance in the deployment group is stopped, the latest application revision is installed, and the new version of the application is started and validated. Only deployments that use the EC2/On-Premises compute platform can use in-place deployments.
  - With blue/green deployments, once the new version of your application is tested and declared ready, CodeDeploy can shift the traffic from your old version (blue) to your new version (green) according to your specifications.
- Deployment groups are used to match configurations to specific environments, such as a staging or production environments. An application can be deployed to multiple deployment groups.
- You can integrate AWS CodeDeploy with your continuous integration and deployment systems by calling the public APIs using the AWS CLI or AWS SDKs.

- Application Specification Files
  - The AppSpec file is a YAML-formatted or JSON-formatted file that is used to manage each deployment as a series of lifecycle event hooks.
  - For ECS Compute platform, the file specifies
    - The name of the ECS service and the container name and port used to direct traffic to the new task set.
    - The functions to be used as validation tests.
  - For Lambda compute platform, the file specifies
    - The AWS Lambda function version to deploy.
    - The functions to be used as validation tests.
  - For EC2/On-Premises compute platform, the file is always written in YAML and is used to
    - Map the source files in your application revision to their destinations on the instance.
    - Specify custom permissions for deployed files.
    - Specify scripts to be run on each instance at various stages of the deployment process.
- Deployments

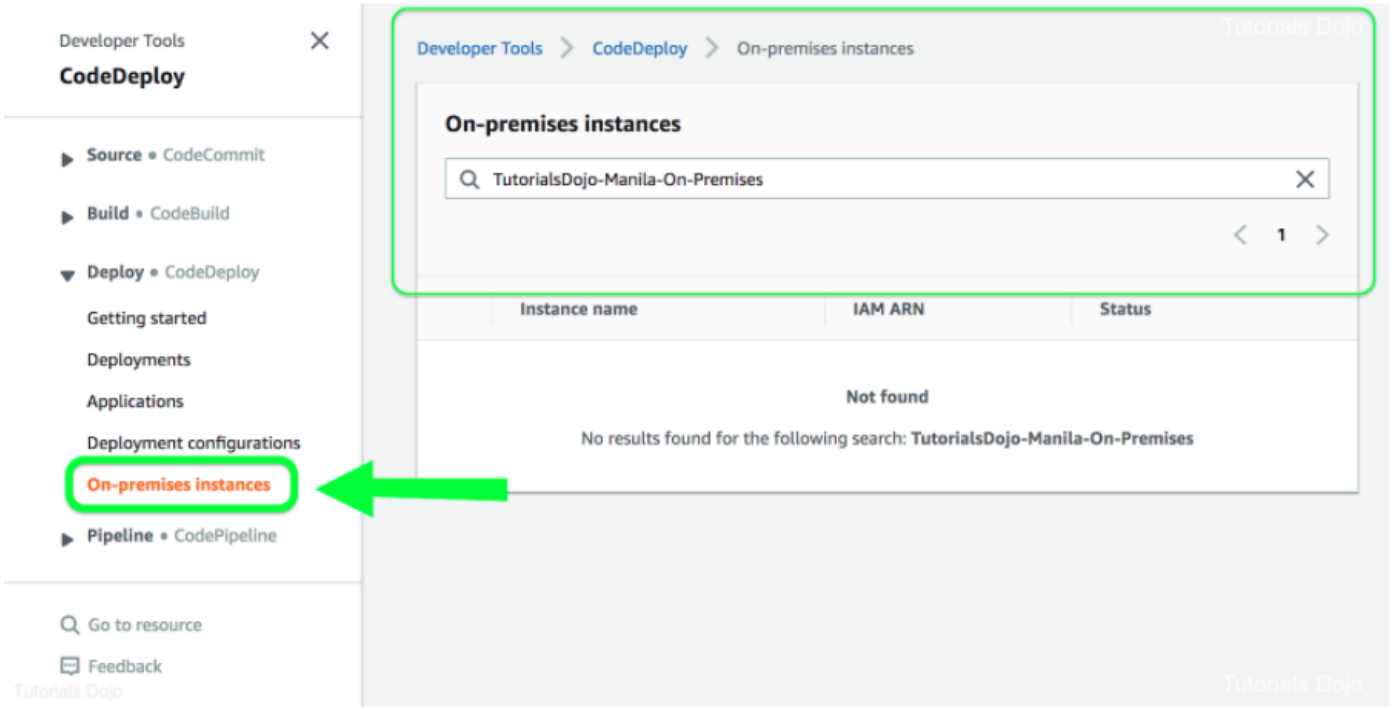
- You can use the CodeDeploy console or the create-deployment command to deploy the function revision specified in the AppSpec file to the deployment group.
- You can use the CodeDeploy console or the stop-deployment command to stop a deployment. When you attempt to stop the deployment, one of three things happens:
  - The deployment stops, and the operation returns a status of SUCCEEDED.
  - The deployment does not immediately stop, and the operation returns a status of pending. After the pending operation is complete, subsequent calls to stop the deployment return a status of SUCCEEDED.
  - The deployment cannot stop, and the operation returns an error.
- With Lambda functions and EC2 instances, CodeDeploy implements rollbacks by redeploying, as a new deployment, a previously deployed revision.
- With ECS services, CodeDeploy implements rollbacks by rerouting traffic from the replacement task set to the original task set.
- The CodeDeploy agent is a software package that, when installed and configured on an EC2/on-premises instance, makes it possible for that instance to be used in CodeDeploy deployments. The agent is not required for deployments that use the Amazon ECS or AWS Lambda.
- CodeDeploy monitors the health status of the instances in a deployment group. For the overall deployment to succeed, CodeDeploy must be able to deploy to each instance in the deployment and deployment to at least one instance must succeed.
- You can specify a minimum number of healthy instances as a number of instances or as a percentage of the total number of instances required for the deployment to be successful.
- CodeDeploy assigns two health status values to each instance:
  - Revision health - based on the application revision currently installed on the instance. Values include Current, Old and Unknown.
  - Instance health - based on whether deployments to an instance have been successful. Values include Healthy and Unhealthy.

- Blue/Green Deployments
  - EC2/On-Premises compute platform
    - You must have one or more Amazon EC2 instances with identifying Amazon EC2 tags or an Amazon EC2 Auto Scaling group.
    - Each Amazon EC2 instance must have the correct IAM instance profile attached.
    - The CodeDeploy agent must be installed and running on each instance.
    - During replacement, you can either
      - use the Amazon EC2 Auto Scaling group you specify as a template for the replacement environment; or
      - specify the instances to be counted as your replacement using EC2 instance tags, EC2 Auto Scaling group names, or both.

  - AWS Lambda platform

        - You must choose one of the following deployment configuration types to specify how traffic is shifted from the original Lambda function version to the new version:
            - Canary: Traffic is shifted in two increments. You can choose from predefined canary options that specify the percentage of traffic shifted to your updated Lambda function version in the first increment and the interval, in minutes, before the remaining traffic is shifted in the second increment.
            - Linear: Traffic is shifted in equal increments with an equal number of minutes between each increment. You can choose from predefined linear options that specify the percentage of traffic shifted in each increment and the number of minutes between each increment.
            - All-at-once: All traffic is shifted from the original Lambda function to the updated Lambda function version all at once.

        - With Amazon ECS, production traffic shifts from your ECS service's original task set to a replacement task set all at once.
    - Advantages of using Blue/Green Deployments vs In-Place Deployments
        - An application can be installed and tested in the new replacement environment and deployed to production simply by rerouting traffic.
        - If you're using the EC2/On-Premises compute platform, switching back to the most recent version of an application is faster and more reliable. Traffic can just be routed back to the original instances as long as they have not been terminated. With an in-place deployment, versions must be rolled back by redeploying the previous version of the application.
        - If you're using the EC2/On-Premises compute platform, new instances are provisioned and contain the most up-to-date server configurations.
        - If you're using the AWS Lambda compute platform, you control how traffic is shifted from your original AWS Lambda function version to your new AWS Lambda function version.
- With AWS CodeDeploy, you can also deploy your applications to your on-premises data centers. Your on-premises instances will have a prefix of “mi-xxxxxxxxx”.


Developer Tools
CodeDeploy
Source • CodeCommit
Build • CodeBuild
Deploy • CodeDeploy
Getting started
Deployments
Applications
Deployment configurations
On-premises instances
Pipeline • CodePipeline
Go to resource
Feedback
Developer Tools > CodeDeploy > On-premises instances
On-premises instances
TutorialsDojo-Manila-On-Premises
1
Instance name
IAM ARN
Status
Not found
No results found for the following search: TutorialsDojo-Manila-On-Premises

AWS CodePipeline

- A fully managed **continuous delivery service** that helps you automate your release pipelines for application and infrastructure updates.
- You can easily integrate AWS CodePipeline with third-party services such as GitHub or with your own custom plugin.
- **Concepts**
    - A **pipeline** defines your release process workflow, and describes how a new code change progresses through your release process.
    - A pipeline comprises a series of **stages** (e.g., build, test, and deploy), which act as logical divisions in your workflow. Each stage is made up of a sequence of actions, which are tasks such as building code or deploying to test environments.
        - Pipelines must have **at least two stages**. The first stage of a pipeline is required to be a source stage, and the pipeline is required to additionally have at least one other stage that is a build or deployment stage.
    - Define your pipeline structure through a **declarative JSON** document that specifies your release workflow and its stages and actions. These documents enable you to update existing pipelines as well as provide starting templates for creating new pipelines.
    - A **revision** is a change made to the source location defined for your pipeline. It can include source code, build output, configuration, or data. A pipeline can have multiple revisions flowing through it at the same time.
    - A **stage** is a group of one or more actions. A pipeline can have two or more stages.
    - An **action** is a task performed on a revision. Pipeline actions occur in a specified order, in serial or in parallel, as determined in the configuration of the stage.
        - You can add actions to your pipeline that are in an AWS Region different from your pipeline.
        - There are six types of actions
            - Source
            - Build
            - Test
            - Deploy
            - Approval
            - Invoke
    - When an action runs, it acts upon a file or set of files called **artifacts**. These artifacts can be worked upon by later actions in the pipeline. You have an artifact store which is an S3 bucket in the same AWS Region as the pipeline to store items for all pipelines in that Region associated with your account.
    - The stages in a pipeline are connected by **transitions.** Transitions can be disabled or enabled between stages. If all transitions are enabled, the pipeline runs continuously.

    - An **approval action** prevents a pipeline from transitioning to the next action until permission is granted. This is useful when you are performing code reviews before code is deployed to the next stage.
- **Features**
    - AWS CodePipeline provides you with a graphical user interface to create, configure, and manage your pipeline and its various stages and actions.
    - A pipeline starts automatically (default) when a change is made in the source location, or when you manually start the pipeline. You can also set up a rule in CloudWatch to automatically start a pipeline when events you specify occur.
    - You can model your build, test, and deployment actions to run **in parallel** in order to increase your workflow speeds.
    - AWS CodePipeline can pull source code for your pipeline directly from AWS CodeCommit, GitHub, Amazon ECR, or Amazon S3.
    - It can run builds and unit tests in AWS CodeBuild.
    - It can deploy your changes using AWS CodeDeploy, AWS Elastic Beanstalk, Amazon ECS, AWS Fargate, Amazon S3, AWS Service Catalog, AWS CloudFormation, and/or AWS OpsWorks Stacks.
    - You can use the CodePipeline Jenkins plugin to easily register your existing build servers as a custom action.
    - When you use the console to create or edit a pipeline that has a GitHub source, CodePipeline creates a **webhook**. A webhook is an HTTP notification that detects events in another tool, such as a GitHub repository, and connects those external events to a pipeline. CodePipeline deletes your webhook when you delete your pipeline.
- As a best practice, when you use a Jenkins build provider for your pipeline's build or test action, install Jenkins on an Amazon EC2 instance and configure a separate EC2 instance profile. Make sure the instance profile grants Jenkins only the AWS permissions required to perform tasks for your project, such as retrieving files from Amazon S3.