AWS®

Certified Developer

Official Study Guide

Associate (DVA-C01) Exam

AWS CodePipeline provides a number of built-in integrations to other AWS services, such as AWS CloudFormation, AWS CodeBuild, AWS CodeCommit, AWS CodeDeploy, Amazon Elastic Container Service (ECS), Elastic Beanstalk, AWS Lambda, AWS OpsWorks Stacks, and Amazon Simple Storage Service (Amazon S3). Some partner tools include GitHub (`https://github.com`) and Jenkins (`https://jenkins.io`). Customers also have the ability to create their own integrations, which provides a great degree of flexibility.

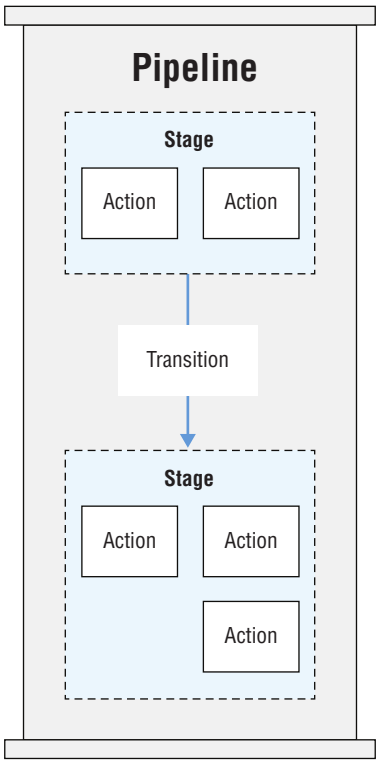
You define workflow steps through a visual editor within the AWS Management Console or via a JavaScript Object Notation (JSON) structure for use in the AWS CLI or AWS SDKs. Access to create and manage release workflows is controlled by AWS Identity and Access Management (IAM). You can grant users fine-grained permissions, controlling what actions they can perform and on which workflows.

AWS CodePipeline provides a dashboard where you can review real-time progress of revisions, attempt to retry failed actions, and review version information about revisions that pass through the pipeline.

AWS CodePipeline Concepts

There are a number of different components that make up AWS CodePipeline and the workflows (*pipelines*) created by customers. Figure 7.3 displays the AWS CodePipeline concepts.

FIGURE 7.3 Pipeline structure



Pipeline

A *pipeline* is the overall workflow that defines what transformations software changes will undergo.

You cannot change the name of a pipeline. If you would like to change the name, you must create a new pipeline.

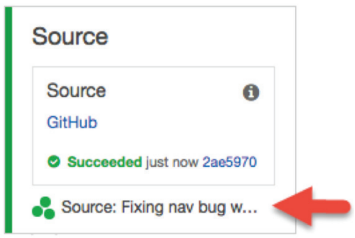
Revision

A *revision* is the work item that passes through a pipeline. It can be a change to your source code or data stored in AWS CodeCommit or GitHub or a change to the version of an archive in Amazon S3. A pipeline can have multiple revisions flowing through it at the same time, but a single stage can process one revision at a time. A revision is immediately picked up by a source action when a change is detected in the source itself (such as a commit to an AWS CodeCommit repository).

If you use Amazon S3 as a source action, you must enable versioning on the bucket.

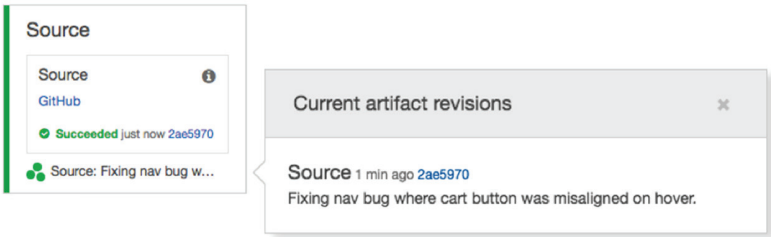
Details of the most recent revision to pass through a stage are kept within the stage itself and are accessible from the console or AWS CLI. To see the last revision that was passed through a source stage, for example, you can select the revision details at the bottom of the stage, as shown in Figure 7.4.

FIGURE 7.4 Source stage



Depending on the source type (Amazon S3, AWS CodeCommit, or GitHub), additional information will be accessible from the revision details pane (such as a link to the commit on `https://github.com`), as shown in Figure 7.5.

FIGURE 7.5 Revision details



Stage

A *stage* is a group of one or more actions. Each stage must have a unique name. Should any one action in a stage fail, the entire stage fails for this revision.

Action

An *action* defines the work to perform on the revision. You can configure pipeline actions to run in series or in parallel. If all actions in a stage complete successfully for a revision, it passes to the next stage in the pipeline. However, if one action fails in the stage, the revision will not pass further through the pipeline. At this point, the stage that contains the failed action can be retried for the same revision. Otherwise, a new revision is able to pass through the stage.

A pipeline must have two or more stages. The first stage includes one or more source actions only. Only the first stage may include source actions.

Every action in the same stage must have a unique name.

Source

The *source* action defines the location where you store and update source files. Modifications to files in a source repository or archive trigger deployments to a pipeline. AWS CodePipeline supports these sources for your pipeline:

- Amazon S3
- AWS CodeCommit
- GitHub

A single pipeline can contain multiple source actions. If a change is detected in one of the sources, all source actions will be invoked.

To use GitHub as a source provider for AWS CodePipeline, you must authenticate to GitHub when you create a pipeline. You provide GitHub credentials to authorize AWS CodePipeline to connect to GitHub to list and view repositories accessible by the authenticating account. For this link, AWS recommends that you create a service account user so that the lifecycle of personal accounts is not tied to the link between AWS CodePipeline and GitHub.

After you authenticate GitHub, a link is created between AWS CodePipeline for this AWS region and GitHub. This allows IAM users to list repositories and branches accessible by the authenticated GitHub user.

Build

You use a *build* action to define tasks such as compiling source code, running unit tests, and performing other tasks that produce output artifacts for later use in your pipeline. For example, you can use a build stage to import large assets that are not part of a source bundle into the artifact to deploy it to Amazon Elastic Compute Cloud (Amazon EC2) instances. AWS CodePipeline supports the integrations for the following build actions:

- AWS CodeBuild
- CloudBees
- Jenkins
- Solano CI
- TeamCity

Test

You can use *test* actions to run various tests against source and compiled code, such as `lint` or syntax tests on source code, and unit tests on compiled, running applications. AWS CodePipeline supports the following test integrations:

- AWS CodeBuild
- BlazeMeter
- Ghost Inspector
- Hewlett Packard Enterprise (HPE) StormRunner Load
- Nouvola
- Runscope

Deploy

The *deploy* action is responsible for taking compiled or prepared assets and installing them on instances, on-premises servers, serverless functions, or deploying and updating infrastructure using AWS CloudFormation templates. The following services are supported as deploy actions:

- AWS CloudFormation
- AWS CodeDeploy
- Amazon Elastic Container Service
- AWS Elastic Beanstalk
- OpsWorks Stacks
- Xebia Labs

Approval

An *approval* action is a manual gate that controls whether a revision can proceed to the next stage in a pipeline. Further progress by a revision is halted until a manual approval by an IAM user or IAM role occurs.

Specifically, the `codepipeline:PutApprovalResult` action must be included in the IAM policy.

Upon approval, AWS CodePipeline approves the revision to proceed to the next stage in the pipeline. However, if the revision is not approved (rejected or the approval expires), the change halts and will stop progress through the pipeline. The purpose of this action is to allow manual review of the code or other quality assurance tasks prior to moving further down the pipeline.

Approval actions cannot occur within source stages.

You must approve actions manually within seven days; otherwise, AWS CodePipeline rejects the code. When an approval action rejects, the outcome is equivalent to when the stage fails. You can retire the action, which initiates the approval process again. Approval actions provide several options that you can use to provide additional information about what you choose to approve.

Publish approval notifications Amazon Simple Notification Service (Amazon SNS) sends notices to one or more targets that approval is pending.

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."

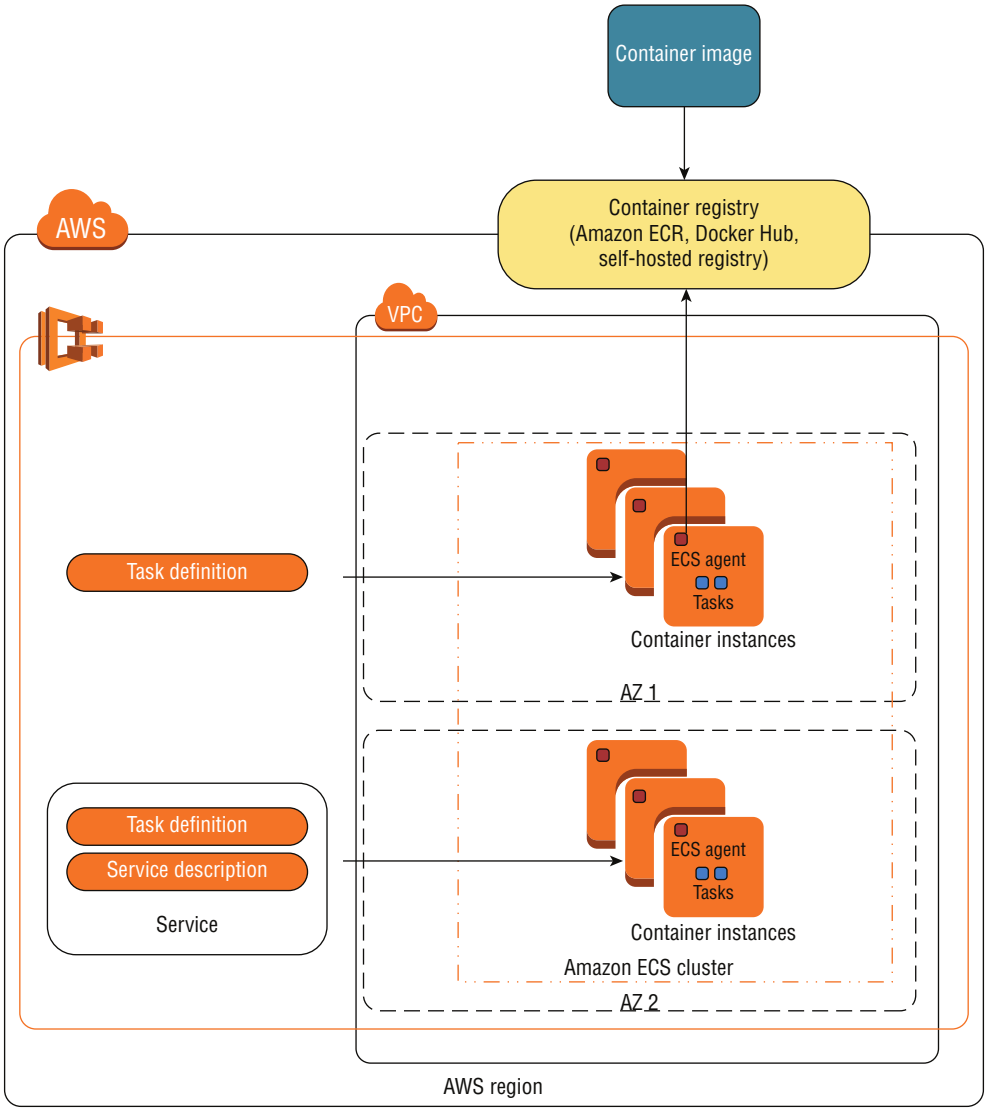
Amazon ECS Concepts

This section details Amazon ECS concepts.

Amazon ECS Cluster

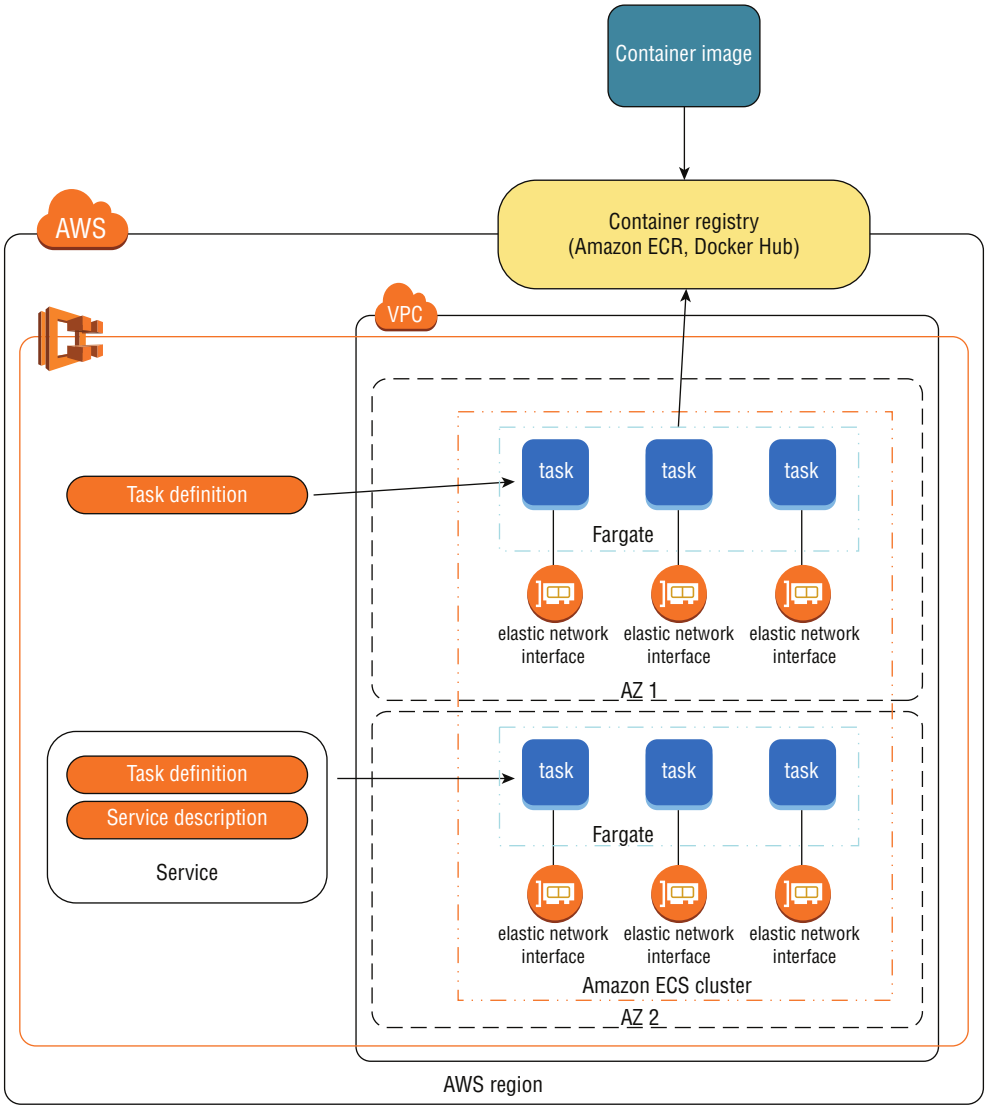
Amazon ECS clusters are the foundational infrastructure components on which containers run. Clusters consist of one or more Amazon EC2 instances in your Amazon VPC. Each instance in a cluster (cluster instance) has an agent installed. The agent is responsible for receiving container scheduling/shutdown 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.

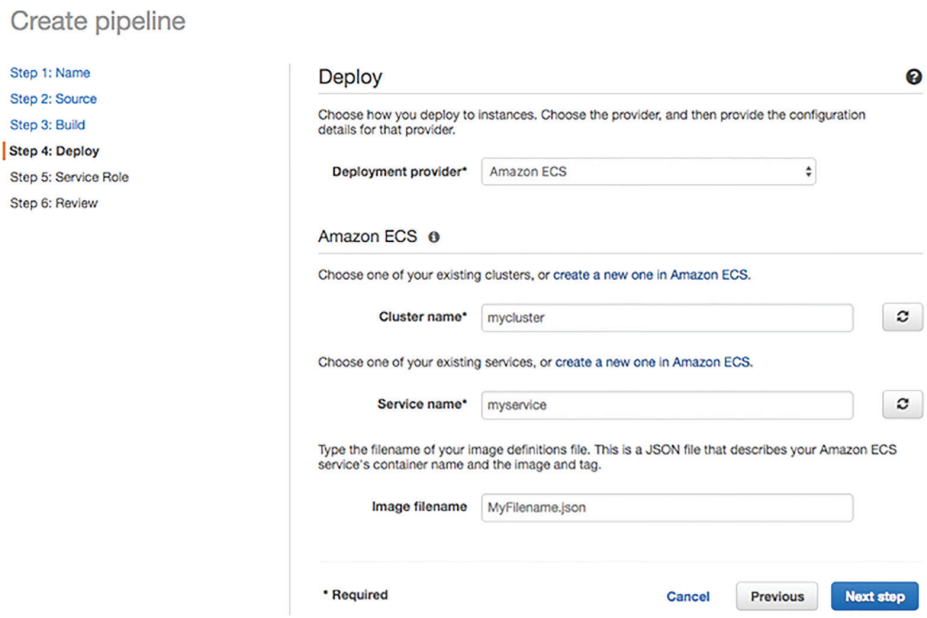
TABLE 9.4 Amazon ECS Service Limits

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/`

You can access the features of AWS Auto Scaling using the AWS CLI, which provides commands to use with Amazon EC2 and Amazon CloudWatch and Elastic Load Balancing.

To scale a resource other than Amazon EC2, you can use the Application Auto Scaling API, which allows you to define scaling policies to scale your AWS resources automatically or schedule one-time or recurring scaling actions.

Using Containers

Containers provide a standard way to package your application's code, configurations, and dependencies into a single object. Containers share an operating system installed on the server and run as resource-isolated processes, ensuring quick, reliable, and consistent deployments, regardless of environment.

Containers provide process isolation that lets you granularly set CPU and memory utilization for better use of compute resources.

Containerize Everything

Containers are a powerful way for developers to package and deploy their applications. They are lightweight and provide a consistent, portable software environment for applications to run and scale effortlessly anywhere.

Use Amazon Elastic Container Service (Amazon ECS) to build all types of containerized applications easily, from long-running applications and microservices to batch jobs and machine learning applications. You can migrate legacy Linux or Windows applications from on-premises to the AWS Cloud and run them as containerized applications using Amazon ECS.

Amazon ECS enables you to use containers as building blocks for your applications by eliminating the need for you to install, operate, and scale your own cluster management infrastructure. You can schedule long-running applications, services, and batch processes using Docker containers. Amazon ECS maintains application availability and allows you to scale your containers up or down to meet your application's capacity requirements. Amazon ECS is integrated with familiar features like Elastic Load Balancing, EBS volumes, virtual private cloud (VPC), and AWS Identity and Access Management (IAM). Use APIs to integrate and use your own schedulers or connect Amazon ECS into your existing software delivery process.

Containers without Servers

AWS Fargate technology is available with Amazon ECS. With Fargate, you no longer have to select Amazon EC2 instance types, provision and scale clusters, or patch and update each server. You do not have to worry about task placement strategies, such as binpacking or host spread, and tasks are automatically balanced across Availability Zones. Fargate manages the availability of containers for you. You define your application's requirements,

select Fargate as your launch type in the AWS Management Console or AWS CLI, and Fargate takes care of all of the scaling and infrastructure management required to run your containers.

For developers who require more granular, server-level control over the infrastructure, Amazon ECS EC2 launch type enables you to manage a cluster of servers and schedule placement of containers on the servers.

Using Serverless Approaches

Serverless approaches are ideal for applications whereby load can vary dynamically. Using a serverless approach means no compute costs are incurred when there is no user traffic, while still offering instant scale to meet high demand, such as a flash sale on an ecommerce site or a social media mention that drives a sudden wave of traffic. All of the actual hardware and server software are handled by AWS.

Benefits gained by using AWS Serverless services include the following:

- No need to manage servers
- No need to ensure application fault tolerance, availability, and explicit fleet management to scale to peak load
- No charge for idle capacity

You can focus on product innovation and rapidly construct these applications:

- Amazon S3 offers a simple hosting solution for static content.
- AWS Lambda, with Amazon API Gateway, supports dynamic API requests using functions.
- Amazon DynamoDB offers a simple storage solution for session and per-user state.
- Amazon Cognito provides a way to handle user registration, authentication, and access control to resources.
- AWS Serverless Application Model (AWS SAM) can be used by developers to describe the various elements of an application.
- AWS CodeStar can set up a CI/CD toolchain with a few clicks.

Compared to traditional infrastructure approaches, an application is also often less expensive to develop, deliver, and operate when it has been architected in a serverless fashion. The serverless application model is generic, and it applies to almost any type of application from a startup to an enterprise.

Here are a few examples of application use cases:

- Web applications and websites
- Mobile backends
- Media and log processing