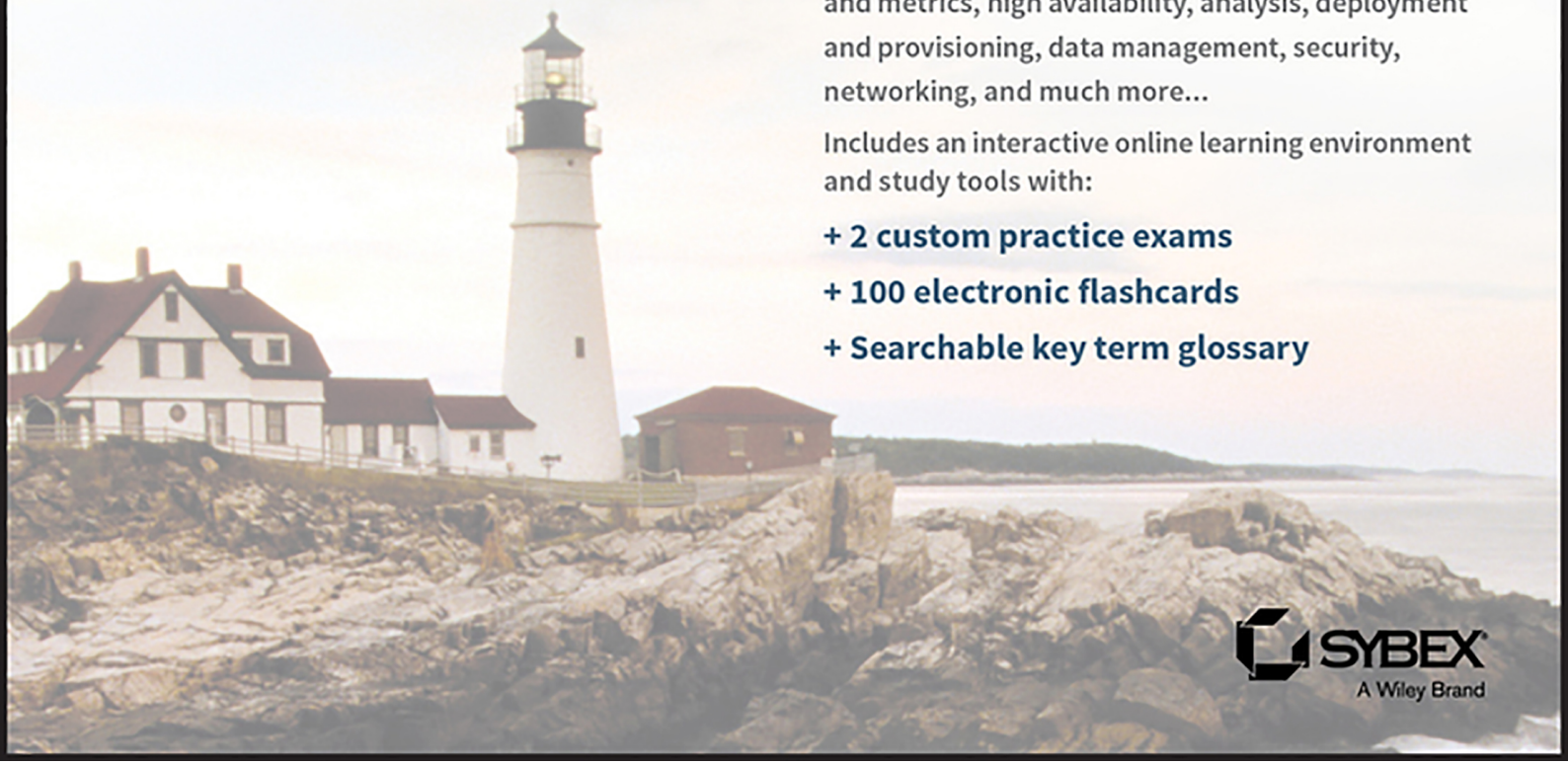
amazon web services

Stephen Cole, Gareth Digby, Chris Fitch,
Steve Friedberg, Shaun Qualheim, Jerry Rhoads,
Michael Roth, Blaine Sundrud

# AWS Certified SysOps Administrator

## OFFICIAL STUDY GUIDE

**ASSOCIATE EXAM**



Covers exam objectives, including monitoring and metrics, high availability, analysis, deployment and provisioning, data management, security, networking, and much more...

Includes an interactive online learning environment and study tools with:

**+ 2 custom practice exams**

**+ 100 electronic flashcards**

**+ Searchable key term glossary**

SYBEX
A Wiley Brand

Chapter

8

# Application Deployment and Management

## THE AWS CERTIFIED SYSOPS ADMINISTRATOR - ASSOCIATE EXAM TOPICS COVERED IN THIS CHAPTER MAY INCLUDE, BUT ARE NOT LIMITED TO, THE FOLLOWING:

**Domain 2.0 High Availability**

**✓ 2.1 Implement scalability and elasticity based on scenarios**

**Content may include the following:**

- Which AWS compute service to use for deploying scalable and elastic environments
- Including scalability of specific AWS compute service in the deployment and management of applications
- Methods for implementing upgrades while maintaining high availability

**✓ 2.2 Ensure level of fault tolerance based on business needs**

**Content may include the following:**

- What AWS Cloud deployment services can be used for deploying fault-tolerant applications

**Domain 4.0 Deployment and Provisioning**

**✓ 4.1 Demonstrate the ability to build the environment to conform to architectural design**

**Content may include the following:**

- Choosing the appropriate AWS Cloud service to meet requirements for deploying applications

**✓ 4.2 Demonstrate the ability to provision cloud resources and manage implementation automation**

**Content may include the following:**

- Automating the deployment and provisioning of AWS Cloud services

# Introduction to Application Deployment and Management

As a candidate for the AWS Certified SysOps Administrator – Associate certification, you will need to be familiar with application deployment strategies and services used for the deployment and management of applications. AWS offers many capabilities for provisioning your infrastructure and deploying your applications. The deployment model varies from customer to customer depending on the capabilities required to support operations. Understanding these capabilities and techniques will help you pick the best strategy and toolset for deploying the infrastructure that can handle your workload.

An experienced systems operator is aware of the "one size doesn't fit all" philosophy. For enterprise computing or to create the next big social media or gaming company, AWS provides multiple customization options to serve a broad range of use cases. The AWS platform is designed to address scalability, performance, security, and ease of deployment, as well as to provide tools to help migrate applications and an ecosystem of developers and architects that are deeply involved in the growth of its products and services.

This chapter details different deployment strategies and services. It reviews common features available on these deployment services, articulates strategies for updating application stacks, and presents examples of common usage patterns for various workloads.

# Deployment Strategies

AWS offers several key features that are unique to each deployment service that will be discussed later in this chapter. There are some characteristics that are common to these services, however. This section discusses various common features and capabilities that a systems operator will need to understand to choose deployment strategies. Each feature can influence service adoption in its own way.

## Provisioning Infrastructure

You can work with building-block services individually, such as provisioning an Amazon Elastic Compute Cloud (Amazon EC2) instance, Amazon Elastic Block Store (Amazon EBS)

volume, Amazon Simple Storage Service (Amazon S3) bucket, Amazon Virtual Private Cloud (Amazon VPC) environment, and so on. Alternately, you can use automation provided by deployment services to provision infrastructure components. The main advantage of using automated capabilities is the rich feature set that they offer for deploying and configuring your application and all the resources it requires. For example, you can use an AWS CloudFormation template to treat your infrastructure as code. The template describes all of the resources that will be provisioned and how they should be configured.

## Deploying Applications

The AWS deployment services can also make it easier to deploy your application on the underlying infrastructure. You can create an application, specify the source repository to your desired deployment service, and let the deployment service handle the complexity of provisioning the AWS resources needed to run your application. Despite providing similar functionality in terms of deployment, each service has its own unique method for deploying and managing your application.

## Configuration Management

Why does a systems operator need to consider configuration management? You may need to deploy resources quickly for a variety of reasons—upgrading your deployments, replacing failed resources, automatically scaling your infrastructure, etc. It is important for a systems operator to consider how the application deployment should be configured to respond to scaling events automatically.

In addition to deploying your application, the deployment services can customize and manage the application configuration. The underlying task could be replacing custom configuration files in your custom web application or updating packages that are required by your application. You can customize the software on your Amazon EC2 instance as well as the infrastructure resources in your stack configuration.

Systems operators need to track configurations and any changes made to the environments. When you need to implement configuration changes, the strategy you use will allow you to target the appropriate resources. Configuration management also enables you to have an automated and repeatable process for deployments.

### Tagging

Another advantage of using deployment services is the automation of tag usage. A *tag* consists of a user-defined key and value. For example, you can define tags such as application, project, cost centers, department, purpose, and stack so that you can easily identify a resource. When you use tags during your deployment steps, the tools automatically propagate the tags to underlying resources such as Amazon EC2 instances, Auto Scaling groups, or Amazon Relational Database Service (Amazon RDS) instances.

Appropriate use of tagging can provide a better way to manage your budgets with cost allocation reports. Cost allocation reports aggregate costs based on tags. This way, you can determine how much you are spending for each application or a particular project.

## Custom Variables

When you develop an application, you want to customize configuration values, such as database connection strings, security credentials, and other information, which you don't want to hardcode into your application. Defining variables can help loosely couple your application configuration and give you the flexibility to scale different tiers of your application independently. Embedding variables outside of your application code also helps improve portability of your application. Additionally, you can differentiate environments into development, test, and production based on customized variables. The deployment services facilitate customizing variables so that once they are set, the variables become available to your application environments. For example, an AWS CloudFormation template could contain a parameter that's used for your web-tier Amazon EC2 instance to connect to an Amazon RDS instance. This parameter is inserted into the user data script so that the application installed on the Amazon EC2 instance can connect to the database.

## Baking Amazon Machine Images (AMIs)

An *Amazon Machine Image (AMI)* provides the information required to launch an instance. It contains the configuration information for instances, including the block device mapping for volumes and what snapshot will be used to create the volume. A *snapshot* is an image of the volume. The root volume would consist of the base operating system and anything else within the volume (such as additional applications) that you've installed.

In order to launch an Amazon EC2 instance, you need to choose which AMI you will use for your application. A common practice is to install an application on an instance at the first boot. This process is called *bootstrapping an instance*.


The bootstrapping process can be slower if you have a complex application or multiple applications to install. Managing a fleet of applications with several build tools and dependencies can be a challenging task during rollouts. Furthermore, your deployment service should be designed to perform faster rollouts to take advantage of Auto Scaling.

*Baking an image* is the process of creating your own AMI. Instead of using a bootstrap script to deploy your application, which could take an extended amount of time, this custom AMI could contain a portion of your application artifacts within it. However, during deployment of your instance, you can also use user data to customize application installations further. AMIs are regionally scoped—to use an image in another region, you will need to copy the image to all regions where it will be used.

The key factor to keep in mind is how long it takes for the instance to launch. If scripted installation of each instance takes an extended amount of time, this could impact your ability to scale quickly. Alternatively, copying the block of a volume where your application is already installed could be faster. The disadvantage is that if your application is changed, you'll need either to bake a new image, which can also be automated, or use a configuration management tool to apply the changes.

For example, let's say that you are managing an environment consisting of web, application, and database tiers. You can have logical grouping of your base AMIs that can take 80 percent of application binaries loaded on these AMI sets. You can choose to install the remaining applications during the bootstrapping process and alter the installation based on configuration sets grouped by instance tags, Auto Scaling groups, or other instance artifacts. You can set a tag on your resources to track for which tier of your environment they are used. When deploying an update, the process can query for the instance tag, validate whether it's the most current version of the application, and then proceed with the installation. When it's time to update the AMI, you can simply swap your existing AMI with the most recent version in the underlying deployment service and update the tag.

You can script the process of baking an AMI. In addition, there are multiple third-party tools for baking AMIs. Some well-known ones are Packer by HashiCorp and Aminator by Netflix. You can also choose third-party tools for your configuration management, such as Chef, Puppet, Salt, and Ansible.

## Logging

*Logging* is an important element of your application deployment cycle. Logging can provide important debugging information or provide key characteristics of your application behavior. The deployment services make it simpler to access these logs through a combination of the AWS Management Console, AWS Command Line Interface (AWS CLI), and Application Programming Interface (API) methods so that you don't have to log in to Amazon EC2 instances to view them.

In addition to built-in features, the deployment services provide seamless integration with Amazon CloudWatch Logs to expand your ability to monitor the system, application, and custom log files. You can use Amazon CloudWatch Logs to monitor logs from Amazon EC2 instances in real time, monitor AWS CloudTrail events, or archive log data in Amazon S3 for future analysis.

## Instance Profiles

Applications that run on an Amazon EC2 instance must include AWS credentials in their API requests. You could have your developers store AWS credentials directly within the Amazon EC2 instance and allow applications in that instance to use those credentials. But developers would then have to manage the credentials and ensure that they securely pass the credentials to each instance and update each Amazon EC2 instance when it's time to rotate the credentials. That's a lot of additional work. There is also the potential that the credentials could be compromised, copied from the Amazon EC2 instance, and used elsewhere.

Instead, you can and should use an AWS Identity and Access Management (IAM) role to manage temporary credentials for applications that run on an Amazon EC2 instance. When you use a role, you don't have to distribute long-term credentials to an Amazon EC2 instance. Instead, the role supplies temporary permissions that applications can use when they make calls to other AWS resources. When you launch an Amazon EC2 instance, you specify an IAM role to associate with the instance. Applications that run on the instance can then use the role-supplied temporary credentials to sign API requests.

Instance profiles are a great way of embedding necessary IAM roles that are required to carry out an operation to access an AWS resource. An instance profile is a container for an IAM role that you can use to pass role information to an Amazon EC2 instance when the instance starts. An instance profile can contain only one IAM role, although a role can be included in multiple instance profiles. These IAM roles can be used to make API requests securely from your instances to AWS Cloud services without requiring you to manage security credentials. The deployment services integrate seamlessly with instance profiles to simplify credentials management and relieve you from hardcoding API keys in your application configuration.

For example, if your application needs to access an Amazon S3 bucket with read-only permission, you can create an instance profile and assign read-only Amazon S3 access in the associated IAM role. The deployment service will take the complexity of passing these roles to Amazon EC2 instances so that your application can securely access AWS resources with the privileges that you define.

## Scalability Capabilities

*Scaling* your application fleet automatically to handle periods of increased demand not only provides a better experience for your end users, but it also keeps the cost low. As demand decreases, resources can automatically be scaled in. Therefore, you're only paying for the resources needed based on the load.

For example, you can configure Auto Scaling to add or remove Amazon EC2 instances dynamically based on metrics triggers that you set within Amazon CloudWatch (such as CPU, memory, disk I/O, and network I/O). This type of Auto Scaling configuration is integrated seamlessly into AWS Elastic Beanstalk and AWS CloudFormation. Similarly, AWS OpsWorks and Amazon EC2 Container Services (Amazon ECS) have capabilities to manage scaling automatically based on time or load. Amazon ECS has Service Auto Scaling that uses a combination of the Amazon ECS, Amazon CloudWatch, and Application Auto Scaling APIs to scale application containers automatically.

## Monitoring Resources

*Monitoring* gives you visibility into the resources you launch in the cloud. Whether you want to monitor the resource utilization of your overall stack or get an overview of your application health, the deployment services are integrated with monitoring capabilities to provide this info within your dashboards. You can navigate to the Amazon CloudWatch console to get a system-wide view into all of your resources and operational health. Alarms can be created for metrics that you want to monitor. When the threshold is surpassed, the alarm is triggered and can send an alert message or take an action to mitigate an issue. For example, you can set an alarm that sends an email alert when an Amazon EC2 instance fails on status checks or trigger a scaling event when the CPU utilization meets a certain threshold.

Each deployment service provides the progress of your deployment. You can track the resources that are being created or removed via the AWS Management Console, AWS CLI, or APIs.

## Continuous Deployment

This section introduces various deployment methods, operations principles, and strategies a systems operator can use to automate integration, testing, and deployment.

Depending on your choice of deployment service, the strategy for updating your application code could vary a fair amount. AWS deployment services bring agility and improve the speed of your application deployment cycle, but using a proper tool and the right strategy is key for building a robust environment.

The following section looks at how the deployment service can help while performing application updates. Like any deployment lifecycle, the methods you use have trade-offs and considerations, so the method you implement will need to meet the specific requirements of a given deployment.

### Deployment Methods

There are two primary methods that you can use with deployment services to update your application stack: in-place upgrade and replacement upgrade. An *in-place upgrade* involves performing application updates on existing Amazon EC2 instances. A *replacement upgrade*, however, involves provisioning new Amazon EC2 instances, redirecting traffic to the new resources, and terminating older instances.

An in-place upgrade is typically useful in a rapid deployment with a consistent rollout schedule. It is designed for stateless applications. You can still use the in-place upgrade method for stateful applications by implementing a rolling deployment schedule and by following the guidelines mentioned in the section below on blue/green deployments.

In contrast, replacement upgrades offer a simpler way to deploy by provisioning new resources. By deploying a new stack and redirecting traffic from the old to the new one, you don't have the complexity of upgrading existing resource and potential failures. This is also useful if your application has unknown dependencies. The underlying Amazon EC2 instance usage is considered temporary or ephemeral in nature for the period of deployment until the current release is active. During the new release, a new set of Amazon EC2 instances is rolled out by terminating older instances. This type of upgrade technique is more common in an immutable infrastructure.

There are several deployment services that are especially useful for an in-place upgrade: AWS CodeDeploy, AWS OpsWorks, and AWS Elastic Beanstalk. *AWS CodeDeploy* is a deployment service that automates application deployments to Amazon EC2 instances or on-premises instances in your own facility. AWS CodeDeploy makes it easier for you to release new features rapidly, helps you avoid downtime during application deployment, and handles the complexity of updating your applications without many of the risks associated with error-prone manual deployments. You can also use *AWS OpsWorks* to manage your application deployment and updates. When you deploy an application, AWS OpsWorks Stacks triggers a Deploy event, which runs each layer's Deploy recipes. AWS OpsWorks Stacks also installs stack configuration and deployment attributes that contain all of the information needed to deploy the application, such as the application's repository and database connection data. *AWS Elastic Beanstalk* provides several options for how deployments are processed, including deployment policies (All at Once, Rolling, Rolling with Additional

Batch, and Immutable) and options that let you configure batch size and health check behavior during deployments.

For replacement upgrades, you provision a new environment with the deployment services, such as AWS Elastic Beanstalk, AWS CloudFormation, and AWS OpsWorks. A full set of new instances running the new version of the application in a separate Auto Scaling group will be created alongside the instances running the old version. Immutable deployments can prevent issues caused by partially completed rolling deployments. Typically, you will use a different Elastic Load Balancing load balancer for both the new stack and the old stack. By using Amazon Route 53 with weighted routing, you can roll traffic to the load balancer of the new stack.

## In-Place Upgrade

AWS CodeDeploy is a deployment service that automates application deployments to Amazon EC2 instances or on-premises instances in your own facility. You can deploy a nearly unlimited variety of application content, such as code, web and configuration files, executables, packages, scripts, multimedia files, and so on. AWS CodeDeploy can deploy application content stored in Amazon S3 buckets, GitHub repositories, or Bitbucket repositories. Once you prepare deployment content and the underlying Amazon EC2 instances, you can deploy an application and its revisions on a consistent basis. You can push the updates to a set of instances called a *deployment group* that is made up of tagged Amazon EC2 instances and/or Auto Scaling groups. In addition, AWS CodeDeploy works with various configuration management tools, continuous integration and deployment systems, and source control systems. You can find a complete list of product integration options in the AWS CodeDeploy documentation.

AWS CodeDeploy is used for deployments by AWS CodeStar. *AWS CodeStar* enables you to develop, build, and deploy applications quickly on AWS. AWS CodeStar provides a unified user interface, enabling you to manage your software development activities easily in one place. With AWS CodeStar, you can set up your entire continuous delivery toolchain in minutes, allowing you to start releasing code faster. AWS CodeStar stores your application code securely on *AWS CodeCommit*, a fully managed source control service that eliminates the need to manage your own infrastructure to host Git repositories. AWS CodeStar compiles and packages your source code with *AWS CodeBuild*, a fully managed build service that makes it possible for you to build, test, and integrate code more frequently. AWS CodeStar accelerates software release with the help of *AWS CodePipeline*, a Continuous Integration and Continuous Delivery (CI/CD) service. AWS CodeStar integrates with AWS CodeDeploy and AWS CloudFormation so that you can easily update your application code and deploy to Amazon EC2 and AWS Lambda.

Another service to use for managing the entire lifecycle of an application is AWS OpsWorks. You can use built-in layers or deploy custom layers and recipes to launch your application stack. In addition, numerous customization options are available for configuration and pushing application updates. When you deploy an application, AWS OpsWorks Stacks triggers a Deploy event, which runs each layer's Deploy recipes. AWS OpsWorks Stacks also installs stack configuration and deployment attributes that contain all of the information needed to deploy the application, such as the application's repository and database connection data.

## Replacement Upgrade

The replacement upgrade method replaces in-place resources with newly provisioned resources. There are advantages and disadvantages between the in-place upgrade method and replacement upgrade method. You can perform a replacement upgrade in a number of ways. You can use an Auto Scaling policy to define how you want to add (scale out) or remove (scale in) instances. By coupling this with your update strategy, you can control the rollout of an application update as part of the scaling event.

For example, you can create a new Auto Scaling Launch Configuration that specifies a new AMI containing the new version of your application. Then you can configure the Auto Scaling group to use the new launch configuration. The Auto Scaling termination policy by default will first terminate the instance with the oldest launch configuration and that is closest to the next billing hour. This in effect provides the most cost-effective method to phase out all instances that use the previous configuration. If you are using Elastic Load Balancing, you can attach an additional Auto Scaling configuration behind the load balancer and use a similar approach to phase in newer instances while removing older instances.

Similarly, you can configure rolling deployments in conjunction with deployment services such as AWS Elastic Beanstalk and AWS CloudFormation. You can use update policies to describe how instances in an Auto Scaling group are replaced or modified as part of your update strategy. With these deployment services, you can configure the number of instances to get updated concurrently or in batches, apply the updates to certain instances while isolating in-service instances, and specify the time to wait between batched updates. In addition, you can cancel or roll back an update if you discover a bug in your application code. These features can help increase the availability of your application during updates.

## Blue/Green Deployments

*Blue/green* is a method where you have two identical stacks of your application running in their own environments. You use various strategies to migrate the traffic from your current application stack (blue) to a new version of the application (green). This method is used for a replacement upgrade. During a blue/green deployment, the latest application revision is installed on replacement instances and traffic is rerouted to these instances either immediately or as soon as you are done testing the new environment.

This is a popular technique for deploying applications with zero downtime. Deployment services like AWS Elastic Beanstalk, AWS CloudFormation, or AWS OpsWorks are particularly useful for blue/green deployments because they provide a simple way to duplicate your existing application stack.

Blue/green deployments offer a number of advantages over in-place deployments. An application can be installed and tested on the new instances ahead of time and deployed to production simply by switching traffic to the new servers. Switching back to the most recent version of an application is faster and more reliable because traffic can 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. Because the instances provisioned for a blue/green deployment are new, they reflect

the most up-to-date server configurations, which helps you avoid the types of problems that sometimes occur on long-running instances.

For a stateless web application, the update process is pretty straightforward. Simply upload the new version of your application and let your deployment service deploy a new version of your stack (green). To cut over to the new version, you simply replace the Elastic Load Balancing URLs in your Domain Name Server (DNS) records. AWS Elastic Beanstalk has a Swap Environment URLs feature to facilitate a simpler cutover process. If you use Amazon Route 53 to manage your DNS records, you need to swap Elastic Load Balancing endpoints for AWS CloudFormation or AWS OpsWorks deployment services.

For applications with session states, the cutover process can be complex. When you perform an update, you don't want your end users to experience downtime or lose data. You should consider storing the sessions outside of your deployment service because creating a new stack will re-create the session database with a certain deployment service. In particular, consider storing the sessions separately from your deployment service if you are using an Amazon RDS database.

If you use Amazon Route 53 to host your DNS records, you can consider using the Weighted Round Robin (WRR) feature for migrating from blue to green deployments. The feature helps to drive the traffic gradually rather than instantly. If your application has a bug, this method helps ensure that the blast radius is minimal, as it only affects a small number of users. This method also simplifies rollbacks if they become necessary by redirecting traffic back to the blue stack. In addition, you only use the required number of instances while you scale up in the green deployment and scale down in the blue deployment. For example, you can set WRR to allow 10 percent of the traffic to go to green deployment while keeping 90 percent of traffic on blue. You gradually increase the percentage of green instances until you achieve a full cutover. Keeping the DNS cache to a shorter Time To Live (TTL) on the client side also ensures that the client will connect to the green deployment with a rapid release cycle, thus minimizing bad DNS caching behavior. For more information on Amazon Route 53, see Chapter 5, "Networking."

### Hybrid Deployments

You can also use the deployment services in a hybrid fashion for managing your application fleet. For example, you can combine the simplicity of managing AWS infrastructure provided by AWS Elastic Beanstalk and the automation of custom network segmentation provided by AWS CloudFormation. Leveraging a hybrid deployment model also simplifies your architecture because it decouples your deployment method so that you can choose different strategies for updating your application stack.

## Deployment Services

AWS deployment services provide easier integration with other AWS Cloud services. Whether you need to load-balance across multiple Availability Zones by using Elastic Load Balancing or by using Amazon RDS as a back end, the deployment services like AWS Elastic

Beanstalk, AWS CloudFormation, and AWS OpsWorks make it simpler to use these services as part of your deployment.

If you need to use other AWS Cloud services, you can leverage tool-specific integration methods to interact with the resource. For example, if you are using AWS Elastic Beanstalk for deployment and want to use Amazon DynamoDB for your back end, you can customize your environment resources by including a configuration file within your application source bundle. With AWS OpsWorks, you can create custom recipes to configure the application so that it can access other AWS Cloud services. Similarly, several template snippets with a number of example scenarios are available for you to use within your AWS CloudFormation templates.

AWS offers multiple strategies for provisioning infrastructure. You could use the building blocks (for example Amazon EC2, Amazon EBS, Amazon S3, and Amazon RDS) and leverage the integration provided by third-party tools to deploy your application. But for even greater flexibility, you can consider the automation provided by the AWS deployment services.

## AWS Elastic Beanstalk

AWS Elastic Beanstalk is the fastest and simplest way to get an application up and running on AWS. It is ideal for developers who want to deploy code and not worry about managing the underlying infrastructure. AWS Elastic Beanstalk reduces management complexity without restricting choice or control. You simply upload your application, and AWS Elastic Beanstalk automatically handles the details of capacity provisioning, load balancing, scaling, and application health monitoring.

AWS Elastic Beanstalk provides platforms for programming languages (such as Java, PHP, Python, Ruby, or Go), web containers (for example Tomcat, Passenger, and Puma), and Docker containers, with multiple configurations of each. AWS Elastic Beanstalk provisions the resources needed to run your application, including one or more Amazon EC2 instances. The software stack running on the Amazon EC2 instances depends on the configuration. In a configuration name, the version number refers to the version of the platform configuration. You can also perform most deployment tasks, such as changing the size of your fleet of Amazon EC2 instances or monitoring your application, directly from the AWS Elastic Beanstalk Console.

### Applications

The first step in using AWS Elastic Beanstalk is to create an application that represents your web application in AWS. In AWS Elastic Beanstalk, an application serves as a container for the environments that run your web application and versions of your web application's source code, saved configurations, logs, and other artifacts that you create while using AWS Elastic Beanstalk.

### Environment Tiers

When you launch an AWS Elastic Beanstalk environment, you choose an environment tier, platform, and environment type. The environment tier that you choose determines whether

Git. A stack can have only one custom cookbook repository, but the repository can contain any number of cookbooks. When you install or update the cookbooks, AWS OpsWorks Stacks installs the entire repository in a local cache on each of the stack's instances. When an instance needs, for example, to run one or more recipes, it uses the code from the local cache.

Repositories can use Source Control Managers, Git, or Subversion, or the repository can be a stored in Amazon S3 or an HTTP location. Cookbooks are organized as directories in the root of the structure. Each cookbook directory has at least one, and typically all, of the standard directories and files, which must use standard names. Standard names of directories are attributes, recipes, templates, and other. The cookbook directory should also include `metadata.rb`, which is the cookbook's metadata. Templates must be in a subdirectory of the templates directory, which contains at least one, and optionally multiple, subdirectories. Those subdirectories can optionally have further subdirectories as well.

## AWS CloudFormation

AWS CloudFormation takes care of provisioning and configuring resources for you. You don't need to create and configure AWS resources individually and figure out what's dependent on what. AWS CloudFormation provides the systems operator, architect, and other personnel with the ability to provision and manage stacks of AWS resources based on templates that you create to model your infrastructure architecture. You can manage anything from a single Amazon EC2 instance to a complex multi-tier, multi-regional application. Using templates means that you can impose version control on your infrastructure and easily replicate your infrastructure stack quickly and with repeatability.

This deployment service is ideal for simplifying deploying infrastructure, while providing capabilities to address complex architectures. For a scalable web application that also includes a back end database, you might use an Auto Scaling group, an Elastic Load Balancing load balancer, and an Amazon RDS DB instance. Normally, you might use each individual service to provision these resources, after which you would have to configure them to work together. All of these tasks can add complexity and time before you even get your application up and running. Instead, you can create or modify an existing AWS CloudFormation template. A template describes all of your resources and their properties.

By using AWS CloudFormation, you can quickly replicate your infrastructure. If your application requires additional availability, you might replicate it in multiple regions so that if one region becomes unavailable, your users can still use your application in other regions. The challenge in replicating your application is that it also requires you to replicate your resources. Not only do you need to record all of the resources that your application requires, but you must also provision and configure those resources in each region. You can reuse your AWS CloudFormation template to set up your resources consistently and repeatedly. You can describe your resources once and then provision the same resources in multiple regions by creating a stack in each region with one template. AWS CloudFormation is also useful for moving from development to test to production phases—simply create the stack and name it for its purpose.

AWS CloudFormation is recommended if you want a tool for granular control over the provisioning and management of your own infrastructure. As part of your infrastructure

deployment, you may want to incorporate the components necessary to deploy application updates. For example, AWS CodeDeploy is a recommended adjunct to AWS CloudFormation for managing the application deployments and updates.

## Creating Stacks

A *stack* is a collection of AWS resources that you can manage as a single unit. In other words, you can create, update, or delete a collection of resources by creating, updating, or deleting stacks. All of the resources in a stack are defined by the stack's AWS CloudFormation template. A stack, for example, can include all of the resources required to run a web application, such as a web server, a database, and networking rules. If you no longer require that web application, you can simply delete the stack and all of its related resources are deleted.

AWS CloudFormation ensures that all stack resources are created or deleted as appropriate. Because AWS CloudFormation treats the stack resources as a single unit, they must all be created or deleted successfully for the stack to be created or deleted. If a resource cannot be created, by default, AWS CloudFormation rolls back the stack creation and automatically deletes any resources that were created. If a resource cannot be deleted, any remaining resources are retained until the stack can be successfully deleted. You can work with stacks by using the AWS CloudFormation console, API, or AWS CLI.

Before you create a stack, you must have a template that describes what resources AWS CloudFormation will include in your stack. Creating a stack on the AWS CloudFormation console is an easy, wizard-driven process.


After you launch a stack, use the AWS CloudFormation console, API, or AWS CLI to update resources in your stack. Do not make changes to stack resources outside of AWS CloudFormation. Doing so can create a mismatch between your stack's template and the current state of your stack resources, which can cause errors if you update or delete the stack.

## Deleting Stacks

After the stack deletion is complete, the stack will be in the `DELETE_COMPLETE` state. Stacks in the `DELETE_COMPLETE` state are not displayed in the AWS CloudFormation console by default. When stacks are in the `DELETE_FAILED` state, because AWS CloudFormation couldn't delete a resource, rerun the deletion with the `RetainResources` parameter and specify the resource that AWS CloudFormation can't delete to bypass or correct the error and rerun. Typically, a delete stack failure most commonly occurs for one of two reasons: There are dependencies external to the stack, or you do not have IAM permissions to delete the resource. Some resources must be empty before they can be deleted. For example, you must delete all objects in an Amazon S3 bucket or remove all instances in an Amazon EC2 security group before you can delete the bucket or security group. In addition to AWS CloudFormation permissions, you must be allowed to use the underlying services, such as Amazon S3 or Amazon EC2.

## Updating Stacks

When you need to make changes to a stack's settings or change its resources, you update the stack instead of deleting it and creating a new stack. For example, if you have a stack with an Amazon EC2 instance, you can update the stack to change the instance's AMI ID. When you update a stack, you submit changes, such as new input parameter values or an updated template. AWS CloudFormation compares the changes you submit with the current state of your stack and updates only the changed resources.


When updating a stack, AWS CloudFormation might interrupt resources or replace updated resources, depending on which properties you update. Review how AWS CloudFormation updates a given resource and whether it will affect end users if it is interrupted.

When you update a stack, AWS CloudFormation updates resources based on differences between what you submit and the stack's current template. Resources that have not changed run without disruption during the update process. For updated resources, AWS CloudFormation uses one of these update behaviors: Update with No Interruption, Updates with Some Interruption, or Replacement. The method AWS CloudFormation uses depends on which property you update for a given resource type. The update behavior for each property is described in the AWS Resource Types Reference section of the AWS CloudFormation User Guide.

Depending on the update behavior, you can decide when to modify resources to reduce the impact of these changes on your application. In particular, you can plan when resources must be replaced during an update. For example, if you update the Port property of an `AWS::RDS::DBInstance` resource type, AWS CloudFormation replaces the DB instance by creating a new DB instance with the updated port setting and deleting the old DB instance.

## Using Change Sets

AWS CloudFormation provides two methods for updating stacks: direct update or creating and executing change sets. When you directly update a stack, you submit changes, and AWS CloudFormation immediately deploys them. Use direct updates when you want to deploy your updates quickly.

With change sets, you can preview the changes that AWS CloudFormation will make to your stack and then decide whether to apply those changes. *Change sets* are JSON-formatted documents that summarize the changes that AWS CloudFormation will make to a stack. Use change sets when you want to ensure that AWS CloudFormation doesn't make unintentional changes, or when you want to consider several options. For example, you can use a change set to verify that AWS CloudFormation won't replace your stack's database instances during an update. Understanding how your changes will affect running resources before you implement them can help you update stacks with confidence. You can create and manage change sets using the AWS CloudFormation console, AWS CLI, or AWS CloudFormation API.

Change sets don't indicate whether AWS CloudFormation will successfully update a stack. For example, a change set doesn't check if you will surpass an account limit, if you're updating a resource that doesn't support updates, or if you have insufficient permissions to modify a resource, all of which can cause a stack update to fail. If an update fails, AWS CloudFormation attempts to roll back your resources to their original state. After you execute a change, AWS CloudFormation removes all change sets that are associated with the stack because they aren't applicable to the updated stack.

When you directly modify resources in the stack's template to generate a change set, AWS CloudFormation classifies the change as a direct modification, as opposed to changes triggered by an updated parameter value. The following change set, which added a new tag to the Amazon EC2 `i-1abc23d4` instance, is an example of a direct modification. All other input values, such as the parameter values and capabilities, are unchanged, so we'll focus on the `Changes` structure.

```
"Changes": [
    {
        "ResourceChange": {
            "ResourceType": "AWS::EC2::Instance",
            "PhysicalResourceId": "i-1abc23d4",
            "Details": [
                {
                    "ChangeSource": "DirectModification",
                    "Evaluation": "Static",
                    "Target": {
                        "Attribute": "Tags",
                        "RequiresRecreation": "Never"
                    }
                }
            ],
            "Action": "Modify",
            "Scope": [
                "Tags"
            ],
            "LogicalResourceId": "MyEC2Instance",
            "Replacement": "False"
        },
        "Type": "Resource"
    }
]
```

In the `Changes` structure, there's only one `ResourceChange` structure. This structure describes information such as the type of resource that AWS CloudFormation will change, the action that AWS CloudFormation will take, the ID of the resource, the scope of the change, and whether the change requires a replacement (where AWS CloudFormation creates a new resource and then deletes the old one). In the example, the change set indicates that AWS CloudFormation will modify the `Tags` attribute of the `i-1abc23d4` Amazon EC2 instance and doesn't require the instance to be replaced.

In the `Details` structure, AWS CloudFormation labels this change as a direct modification that will never require the instance to be re-created (replaced). You can confidently execute this change, knowing that AWS CloudFormation won't replace the instance.

AWS CloudFormation shows this change as a `Static` evaluation. A *Static* evaluation means that AWS CloudFormation can determine the tag's value before executing the change set. In some cases, AWS CloudFormation can determine a value only after you execute a change set. AWS CloudFormation labels those changes as `Dynamic` evaluations. For example, if you reference an updated resource that is conditionally replaced, AWS CloudFormation can't determine whether the reference to the updated resource will change.

## Template Anatomy

To provision and configure your stack resources, you must understand *AWS CloudFormation templates*, which are formatted text files in JSON or YAML. These templates describe the resources that you want to provision in your AWS CloudFormation Stacks. You can use AWS CloudFormation Designer or any text editor to create and save templates.

When you provision your infrastructure with AWS CloudFormation, the AWS CloudFormation template describes exactly what resources are provisioned and their settings. Because these templates are text files, you simply track differences in your templates to track changes to your infrastructure, similar to the way developers control revisions to source code. For example, you can use a version control system with your templates so that you know exactly what changes were made, who made them, and when. If you need to reverse changes to your infrastructure, you can use a previous version of your template.

You can author AWS CloudFormation templates in JSON or YAML formats. AWS supports all AWS CloudFormation features and functions for both formats, including in AWS CloudFormation Designer. When deciding which format to use, pick the format in which you're most comfortable working. Also consider that YAML inherently provides some features, such as commenting, that aren't available in JSON. The following example shows a JSON-formatted template fragment. Be aware that the `AWSTemplateFormatVersion` section of the template identifies the capabilities of the template. The latest template format version is `2010-09-09`, and it is currently the only valid value.

```
{
  "AWSTemplateFormatVersion" : " 2010-09-09",
  "Description" : "JSON string",
  "Metadata" : {
    template metadata
```

```
  },
  "Parameters" : {
    set of parameters
  },
  "Mappings" : {
    set of mappings
  },
  "Conditions" : {
    set of conditions
  },
  "Transform" : {
    set of transforms
  },
  "Resources" : {
    set of resources
  },
  "Outputs" : {
    set of outputs
  }
}
```

## Template Metadata

You can use the optional `Metadata` section to include arbitrary JSON or YAML objects that provide details about the template. For example, you can include template implementation details about specific resources. During a stack update, however, you cannot update the `Metadata` section by itself. You can update it only when you include changes that add, modify, or delete resources.

```
"Metadata" : {
  "Instances" : {"Description" : "Information about the instances"},
  "Databases" : {"Description" : "Information about the databases"}
}
```

Some AWS CloudFormation features retrieve settings or configuration information that you define from the `Metadata` section. You define this information in AWS CloudFormation-specific metadata keys.

`AWS::CloudFormation::Init` defines configuration tasks for the `cfn-init` helper script. This script is useful for configuring and installing applications on Amazon EC2 instances.

`AWS::CloudFormation::Interface` is a metadata key that defines how parameters are grouped and sorted in the AWS CloudFormation console. When you create or update stacks in the console, the console lists input parameters in alphabetical order by their logical IDs. By using this key, you can define your own parameter grouping and ordering so that users

can efficiently specify parameter values. In addition to grouping and ordering parameters, you can define labels for parameters. A *label* is a user-friendly name or description that the console displays instead of a parameter's logical ID. Labels are useful for helping users understand the values to specify for each parameter.

```
"Metadata" : {
  "AWS::CloudFormation::Interface" : {
    "ParameterGroups" : [ ParameterGroup, ... ],
    "ParameterLabels" : ParameterLabel
  }
}
```

## Template Parameters

You can use the optional `Parameters` section to pass values into your template when you create a stack. With parameters, you can create templates that are customized each time that you create a stack. Each parameter must contain a value when you create a stack. You can specify a default value to make the parameter optional so that you don't need to pass in a value when creating a stack.

```
"Parameters" : {
  "InstanceTypeParameter" : {
    "Type" : "String",
    "Default" : "t2.micro",
    "AllowedValues" : ["t2.micro", "m1.small", "m1.large"],
    "Description" : "Select t2.micro, m1.small, or m1.large. Default is t2.micro."
  }
}
```

Within the same template, you can use the `Ref` intrinsic function to use the parameter value in other parts of the template. The following snippet uses the `InstanceTypeParameter` parameter to specify the instance type for an Amazon EC2 instance resource. Keep in mind that parameter and resource names are set by you and must be named uniquely within the template.

```
"Resources" : {
  "EC2Instance" : {
    "Type" : "AWS::EC2::Instance",
    "Properties" : {
      "InstanceType" : { "Ref" : "InstanceTypeParameter" },
      "ImageId" : "ami-2f726546"
    }
  }
}
```

When using parameter types in your templates, AWS CloudFormation can validate parameter values during the stack creation and update process, saving you time, effort, and cost. Using parameter types also enables AWS CloudFormation to show you a more intuitive user interface in the stack creation and update wizards. For example, a drop-down user interface of valid key-pair names is displayed when using a parameter of type `AWS::EC2::KeyPair::KeyName`. It is a best practice to use a parameter type that is the most restrictive appropriate type. You could use a string parameter type for an Amazon EC2 instance key-pair name, but that would allow operators who are deploying it to enter any string, even if it may not be a valid key pair within the region for the given account.

```
"Parameters" : {
  "KeyPairParameter " : {
    "Type" : " AWS::EC2::KeyPair::KeyName ",
    "Description" : " Key pair used for EC2 instance. "
  }
},
"Resources" : {
  "EC2Instance" : {
    "Type" : "AWS::EC2::Instance",
    "Properties" : {
      "InstanceType" : { "Ref" : "InstanceTypeParameter" },
      "KeyName" : { "Ref" : "KeyPairParameter" },
      "ImageId" : "ami-2f726546"
    }
  }
}
```

## Template Mapping

The optional `Mappings` section matches a key to a corresponding set of named values. For example, if you want to set values based on a region, you can create a mapping that uses the region name as a key and contains the values that you want to specify for each specific region. You use the `Fn::FindInMap` intrinsic function to retrieve values in a map. However, you cannot include parameters, pseudo parameters, or intrinsic functions in the `Mappings` section.

The `Mappings` section consists of the key name mappings. The keys and values in mappings must be literal strings. Within a mapping, each map is a key followed by another mapping. The key identifies a map of name/value pairs and must be unique within the mapping. The name can contain only alphanumeric characters.

```
"Mappings" : {
  "Mapping01" : {
    "Key01" : {
      "Name" : "Value01"
    },
```

```
    "Key02" : {
      "Name" : "Value02"
    },
    "Key03" : {
      "Name" : "Value03"
    }
  }
}
```

Most commonly, you'll see a `Mappings` section used for AMI IDs. If you want to use the template in multiple regions, you'll need a mapping to use the corresponding ID for the region. An AMI ID is unique for the account within a region. That means that if you hardcode the image value for an Amazon EC2 instance, you would only be able to use the template within the region where that image exists. Even if you copy an image to another region, that copied image will have a different ID, which you will need to map as well.

As mentioned previously, you can use the `Fn::FindInMap` function to return a named value based on a specified key. This is a three-parameter function. First, you specify the name of the map within which to look. Second, you specify the key to find within the map. Third, you specify which attribute you want to return from the item.

In the following example, let's assume that you have two images for an instance, and these images have two different versions of your application. You could test each version by creating a stack for each and selecting the appropriate parameter. Additionally, instead of specifying a parameter for the region, you can use a pseudo parameter. `AWS::Region` is a pseudo parameter. Because AWS CloudFormation is a regional service and you specify the region to which to deploy, you don't have to supply the region name manually.

```
"Parameters":{
  "InstanceType" : {
    "Type"        : "String",
    "Default"     : "Version1",
    "AllowedValues" : [
        "Version1",
        "Version2"
    ]
},
"Mappings" : {
  "RegionMap" : {
    "us-east-1"      : { "Version1" : "ami-6411e20d", "Version2" : "ami-7a11e213" },
    "us-west-1"      : { "Version1" : "ami-c9c7978c", "Version2" : "ami-cfc7978a" },
    "eu-west-1"      : { "Version1" : "ami-37c2f643", "Version2" : "ami-31c2f645" },
    "ap-southeast-1" : { "Version1" : "ami-66f28c34", "Version2" : "ami-60f28c32" },
    "ap-northeast-1" : { "Version1" : "ami-9c03a89d", "Version2" : "ami-a003a8a1" }
  }
},
```

```
"Resources" : {
  "myEC2Instance" : {
    "Type" : "AWS::EC2::Instance",
    "Properties" : {
      "ImageId" : { "Fn::FindInMap" : [
        "RegionMap",
        { "Ref" : "AWS::Region" },
        { "Ref" : "VersionParameter" }]
      },
      "InstanceType" : "t2.micro"
    }
  }
}
```

## Template Conditions

The optional `Conditions` section includes statements that define when a resource is created or when a property is defined. For example, you can compare whether a value is equal to another value. Based on the result of that condition, you can conditionally create resources. If you have multiple conditions, separate them with commas.

You might use conditions when you want to reuse a template that can create resources in different contexts, such as a test environment versus a production environment. In your template, you can add an `EnvironmentType` input parameter, which accepts either prod or test as inputs. For the production environment, you might include Amazon EC2 instances with certain capabilities. For the test environment, however, you probably want to use reduced capabilities to save money. With conditions, you can define which resources are created and how they're configured for each environment type.

Conditions are evaluated based on input parameter values that you specify when you create or update a stack. Within each condition, you can reference another condition, a parameter value, or a mapping. After you define all of your conditions, you can associate them with resources and resource properties in the `Resources` and `Outputs` sections of a template.

At stack creation or stack update, AWS CloudFormation evaluates all of the conditions in your template before creating any resources. Any resources that are associated with a true condition are created. Any resources that are associated with a false condition are ignored.


During a stack update, you cannot update conditions by themselves. You can update conditions only when you include changes that add, modify, or delete resources.

To create resources conditionally, you must include statements in at least three different sections of a template: `Parameters`, `Conditions`, and `Resources`. The `Parameters` section defines the input values that you want to evaluate in your conditions. Conditions will result

in a true or false result, based on values from these input parameters. The `Conditions` section is where you define the intrinsic condition functions. These conditions determine when AWS CloudFormation creates the associated resources. The `Resources` section is where you associate conditions with the resources that you want to create conditionally. Use the condition key and a condition's logical ID to associate it with a resource or output.

You can use the following intrinsic functions to define conditions:

`Fn::And`

`Fn::Equals`

`Fn::If`

`Fn::Not`

`Fn::Or`

The following sample template includes an `EnvType` input parameter, where you can specify `prod` to create a stack for production or test to create a stack for testing. The `CreateProdResources` condition evaluates to true if the `EnvType` parameter is equal to `prod`. In the sample template, the Amazon EC2 instance is associated with the `CreateProdResources` condition. Therefore, the resources are created only if the `EnvType` parameter is equal to `prod`. When the parameter is not set to `prod`, it does not create the instance.

```
{
  "AWSTemplateFormatVersion" : "2010-09-09",
  "Parameters" : {
    "EnvType" : {
      "Default" : "test",
      "Type" : "String",
      "AllowedValues" : ["prod", "test"],
    }
  },
  "Conditions" : {
    "CreateProdResources" : {"Fn::Equals" : [{"Ref" : "EnvType"}, "prod"]}
  },
  "Resources" : {
    "EC2Instance" : {
      "Type" : "AWS::EC2::Instance",
      "Condition" : "CreateProdResources",
      "Properties" : {
        "ImageId" : { "Fn::FindInMap" : [ "RegionMap", { "Ref" : "AWS::Region"
}, "AMI" ]}
      }
    }
  }
}
```

## Template Resources

The required `Resources` section declares the AWS resources that you want to include in the stack, such as an Amazon EC2 instance or an Amazon S3 bucket. You must declare each resource separately. If you have multiple resources of the same type, however, you can declare them together by separating them with commas. The `Resources` section consists of the key name resources.

Each of the resources within this section has a logical ID. The logical ID must be alphanumeric (`A-Za-z0-9`) and unique within the template. Use the logical ID to reference the resource in other parts of the template. For example, if you want to map an Amazon EBS volume to an Amazon EC2 instance, you reference the logical IDs to associate the block stores with the instance.

```
"Resources" : {
    "MyLogicalID" : {
        "Type" : "Resource type",
        "Properties" : {
            Set of properties
        }
    }
}
```

In addition to the logical ID, certain resources also have a physical ID, which is the actual assigned name for that resource, such as an Amazon EC2 instance ID or an Amazon S3 bucket name. Use the physical IDs to identify resources outside of AWS CloudFormation templates, but only after the resources have been created. For example, you might give an Amazon EC2 instance resource a logical ID of `MyEC2Instance`, but when AWS CloudFormation creates the instance, AWS CloudFormation automatically generates and assigns a physical ID (such as `i-28f9ba55`) to the instance. You can use this physical ID to identify the instance and view its properties (such as the DNS name) by using the Amazon EC2 Console. Since you do not know the physical ID of a resource until the stack is created, you use references to associate resources where you would normally use a physical ID. In the following example, an Elastic IP resource needs to be associated with an Amazon EC2 instance by providing the instance ID. Since we do not know the instance ID ahead of stack creation, we reference the Amazon EC2 instance also created with the AWS CloudFormation template.

```
"Resources" : {
  "EC2Instance" : {
    "Type" : "AWS::EC2::Instance",
    "Properties" : {
      "ImageId" : "ami-7a11e213"
    }
  },
  "MyEIP" : {
```

```
      "Type" : "AWS::EC2::EIP",
      "Properties" : {
         "InstanceId" : { "Ref" : "EC2Instance" }
      }
   }
}
```

The resource type identifies the type of resource that you are declaring. For example, `AWS::EC2::Instance` declares an Amazon EC2 instance. Resource properties are additional options that you can specify for a resource. For example, for each Amazon EC2 instance, you must specify an AMI ID for that instance. Property values can be literal strings, lists of strings, Booleans, parameter references, pseudo references, or the value returned by a function, depending on what the property is used for. The following are the properties available for an `AWS::EC2::Instance` AWS CloudFormation resource.

```
{
   "Type" : "AWS::EC2::Instance",
   "Properties" : {
      "Affinity" : String,
      "AvailabilityZone" : String,
      "BlockDeviceMappings" : [ EC2 Block Device Mapping, ... ],
      "DisableApiTermination" : Boolean,
      "EbsOptimized" : Boolean,
      "HostId" : String,
      "IamInstanceProfile" : String,
      "ImageId" : String,
      "InstanceInitiatedShutdownBehavior" : String,
      "InstanceType" : String,
      "Ipv6AddressCount" : Integer,
      "Ipv6Addresses" : [ IPv6 Address Type, ... ],
      "KernelId" : String,
      "KeyName" : String,
      "Monitoring" : Boolean,
      "NetworkInterfaces" : [ EC2 Network Interface, ... ],
      "PlacementGroupName" : String,
      "PrivateIpAddress" : String,
      "RamdiskId" : String,
      "SecurityGroupIds" : [ String, ... ],
      "SecurityGroups" : [ String, ... ],
      "SourceDestCheck" : Boolean,
      "SsmAssociations" : [ SSMAssociation, ... ],
      "SubnetId" : String,
```

```
      "Tags" : [ Resource Tag, ... ],
      "Tenancy" : String,
      "UserData" : String,
      "Volumes" : [ EC2 MountPoint, ... ],
      "AdditionalInfo" : String
   }
}
```

There are hundreds of different AWS resources supported directly within AWS CloudFormation, each with their own set of properties. While not every AWS resource is directly supported, you can use custom resources where they are not supported directly. Custom resources enable you to write custom provisioning logic in templates that AWS CloudFormation runs any time you create, update (if you changed the custom resource), or delete stacks. Use the `AWS::CloudFormation::CustomResource` or `Custom::String` resource type to define custom resources in your templates. Custom resources require one property, the service token, which specifies where AWS CloudFormation sends requests (for example, an Amazon Simple Notification Service [Amazon SNS] topic).

## Template Outputs

The optional `Outputs` section declares output values that you can import into other stacks (to create cross-stack references), return in response (to describe stack calls), or view on the AWS CloudFormation console. For example, you can output the Amazon S3 bucket name from a stack to make the bucket easier to find. The `Outputs` section consists of the key name outputs followed by a space and a single colon. The value of the property is returned by the AWS CloudFormation `describe-stacks` command. The value of an output can include literals, parameter references, pseudo parameters, a mapping value, or intrinsic functions.

In the following example, the output named `LoadBalancerDNSName` returns the DNS name for the resource with the logical ID `LoadBalancer` only when the `CreateProdResources` condition is true. The second output shows how to specify multiple outputs and use a reference to a resource to return its physical ID.

```
"Outputs" : {
  "LoadBalancerDNSName" : {
    "Description": "The DNSName of the backup load balancer",
    "Value" : { "Fn::GetAtt" : [ "LoadBalancer", "DNSName" ]},
    "Condition" : "CreateProdResources"
  },
  "InstanceID" : {
    "Description": "The Instance ID",
    "Value" : { "Ref" : "EC2Instance" }
  }
}
```

Outputs also have an export field. Exported values are useful for cross-stack referencing. In the following examples, the output named `StackVPC` returns the ID of an Amazon VPC and then exports the value for cross-stack referencing with the name `VPCID` appended to the stack's name.

```
"Outputs" : {
  "StackVPC" : {
    "Description" : "The ID of the VPC",
    "Value" : { "Ref" : "MyVPC" },
    "Export" : {
      "Name" : {"Fn::Sub": "${AWS::StackName}-VPCID" }
    }
  }
}
```

## Best Practices

*Best practices* are recommendations that can help you use AWS CloudFormation more effectively and securely throughout its entire workflow. Planning and organizing your stacks, creating templates that describe your resources and the software applications that run on them, and managing your stacks and their resources are all good best practices.

Use the lifecycle and ownership of your AWS resources to help you decide what resources should go in each stack. Normally, you might put all of your resources in one stack. As your stack grows in scale and broadens in scope, however, managing a single stack can be cumbersome and time consuming. By grouping resources with common lifecycles and ownership, owners can make changes to their set of resources by using their own process and schedule without affecting other resources. A layered architecture organizes stacks into multiple horizontal layers that build on top of one another, where each layer has a dependency on the layer directly below it. You can have one or more stacks in each layer, but within each layer your stacks should have AWS resources with similar lifecycles and ownership.

When you organize your AWS resources based on lifecycle and ownership, you might want to build a stack that uses resources that are in another stack. You can hardcode values or use input parameters to pass resource names and IDs. However, these methods can make templates difficult to reuse or can increase the overhead to get a stack running. Instead, use cross-stack references to export resources from a stack so that other stacks can use them. Stacks can use the exported resources by calling them using the `Fn::ImportValue` function.

You can reuse your templates to replicate your infrastructure in multiple environments. For example, you can create environments for development, testing, and production so that you can test changes before implementing them into production. To make templates reusable, use the `Parameters`, `Mappings`, and `Conditions` sections so that you can customize your stacks when you create them. For example, for your development environments, you can specify a lower-cost instance type compared to your production environment, but all other configurations and settings remain the same. Creating your stacks based on the organizational structure is also useful. For example, the network architects create a template

that provisions the Amazon VPC in a standard way for the organization. By parameterizing the template, they can reuse it for different deployments.

## AWS Command Line Interface (AWS CLI)

The AWS CLI can be used to automate systems operations. You can use it to manage an operational environment that is currently in production, automating the provisioning and updating of application deployments.

### Generating Skeletons

Most AWS CLI commands support `--generate-cli-skeleton` and `--cli-input-json` parameters that you can use to store parameters in JSON and read them from a file instead of typing them at the command line. You can generate AWS CLI skeleton outputs in JSON that outline all of the parameters that can be specified for the operation. This is particularly useful for generating templates that are used in scripting the deployment of applications. For example, you can use it to generate a template for Amazon ECS task definitions or an AWS CloudFormation resource's `Properties` section.

To generate an Amazon ECS task definition template, run the following AWS CLI command.

```
aws ecs register-task-definition --generate-cli-skeleton
```

You can use this template to create your task definition, which can then be pasted into the console JSON input area or saved to a file and used with the AWS CLI `--cli-input-json` option.

This process is also useful in generating the properties attributes for an AWS CloudFormation resource. You could use the output generated in the previous command.

```
{
  "Type" : "AWS::ECS::TaskDefinition",
  "Properties" : {
       <paste-generate-cli-skeleton>
  }
}
```

This can be done for other resources in an AWS CloudFormation template, such as an Amazon EC2 instance. Run the following command and paste the output into the `Properties` section of the resource.

```
aws ec2 run-instances --generate-cli-skeleton
```

```
{
  "Type" : "AWS::EC2::Instance",
  "Properties" : {
        <paste-generate-cli-skeleton>
  }
}
```

# Summary

In this chapter, we discussed the following deployment strategies:

- Provisioning Infrastructure
  - Automating the provisioning of building-block services with AWS Cloud deployment services
- Deploying Applications
  - Methods of deploying applications onto your infrastructure
- Configuration Management
  - Using tagging to track resources and manage infrastructure
  - Using custom variables to make deployments flexible and reusable
  - Strategies for creating AMIs
  - Configuring infrastructure for security and troubleshooting
- Scalability Capabilities
  - Including scaling capabilities for the infrastructure of an application
  - Scalability considerations during the upgrade of an application
- Continuous Deployment
  - In-place vs. replacement upgrades
  - Methods for deploying application and infrastructure upgrades

In this chapter, we discussed the following deployment services:

- AWS Elastic Beanstalk
  - Creating and managing AWS Elastic Beanstalk environments
  - Deploying and managing an application in an AWS Elastic Beanstalk environment
- Amazon ECS
  - Building an Amazon ECS cluster
  - Deploying instances used for an Amazon ECS cluster
  - Managing containers with Amazon ECS tasks and services
  - Using a repository for container images
- AWS OpsWorks Stacks
  - Creating an AWS OpsWorks stack
  - Using layers for managing Amazon EC2 instances
  - Deploying applications to a layer of your stack
- AWS CloudFormation
  - Creating, updating, and deleting an AWS CloudFormation Stack
  - Methods and considerations for updating an AWS CloudFormation Stack

- Anatomy of AWS CloudFormation templates
- Strategies and best practices for managing AWS CloudFormation Stacks and templates

It is important to understand the various capabilities of each deployment service, including what they can and cannot do. The exam will not ask you about the actual commands used in deploying applications or how to create a template, however.

# Resources to Review

**AWS YouTube channel:** `https://www.youtube.com/user/AmazonWebServices`

**AWS What's New:** `https://aws.amazon.com/new`

**AWS CloudFormation documentation:**
`http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html`

**AWS OpsWorks Stacks documentation:**
`http://docs.aws.amazon.com/opsworks/latest/userguide/welcome.html`

**Amazon ECS documentation:**
`http://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html`

**Amazon ECR documentation:**
`http://docs.aws.amazon.com/AmazonECR/latest/userguide/what-is-ecr.html`

**AWS Elastic Beanstalk documentation:**
`http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html`

**Blue/Green Deployments on AWS whitepaper:**
`https://d0.awsstatic.com/whitepapers/AWS_Blue_Green_Deployments.pdf`

**Creating an Amazon EBS-backed Linux AMI:**
`http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/creating-an-ami-ebs.html`

# Exam Essentials

**Know how to work with AWS CloudFormation Stacks.** This includes creating, updating, and deleting an AWS CloudFormation Stack.

**Understand the anatomy of an AWS CloudFormation template.** Templates include several major sections. These sections are `Metadata`, `Parameters`, `Mappings`, `Conditions`, `Resources`, `Transform`, and `Outputs`. The `Resources` section is the only required section.

**Understand an AWS CloudFormation change set.** Understanding how your changes will affect running resources before you implement them can help you update stacks.

**Know how to create an AWS OpsWorks stack.** How to create an AWS OpsWorks stack, create layers within that stack, and deploy applications to a layer

**Know how to deploy an Amazon ECS cluster.** Using the Amazon ECS-optimized AMI for use in a cluster. Assigning an instance profile so that the agent can communicate with the Amazon ECS service

**Understand various deployment methodologies.** Understand blue/green deployments, in-place upgrades, and replacement upgrades, how they differ, and why you might use one method instead of another.

**Know how to create a custom AMI.** Understand why baking your own AMI can be useful in deploying applications. You could decide to create only a base image, one that includes application frameworks, or one that includes everything, including your application code. Including everything can accelerate deployments; however, that requires creating a new image with every upgrade.

**Understand how to use tagging for configuring resources.** Tagging can be used for targeting groups of resources to which you wish to deploy applications. It can also be useful in version control, customizing deployments, managing resources, and so on.

**Know how to deploy an application with AWS Elastic Beanstalk.** Understand how to create an environment and specify the type of platform it will run.

**Know the AWS Elastic Beanstalk environment tiers.** There is a web tier and a platform tier. If your application performs operations or workflows that take a long time to complete, you can offload those tasks to a dedicated worker environment.

**Know that you can use AWS Elastic Beanstalk for blue/green deployments.** AWS Elastic Beanstalk performs an in-place upgrade when you update your application versions. By performing a blue/green deployment, you deploy the new version to a separate environment and then swap CNAMEs of the two environments to redirect traffic to the new version instantly.

**Understand how to use Amazon EC2 instance profiles and why they are used.** An instance profile is a container for an IAM role that you can use to pass role information to an Amazon EC2 instance when the instance starts. This allows you to provide automatic key rotation without storing credentials on the instance.

### Test Taking Tip

*The test is exact*—do not add any conditions (for example, encrypted at rest) to the questions or presented scenarios.