AWS® Certified Cloud Practitioner

STUDY GUIDE

FOUNDATIONAL (CLF-C01) EXAM

Includes interactive online learning environment and study tools:

Two custom practice exams

100 electronic flashcards

Searchable key term glossary

BEN PIPER
DAVID CLINTON

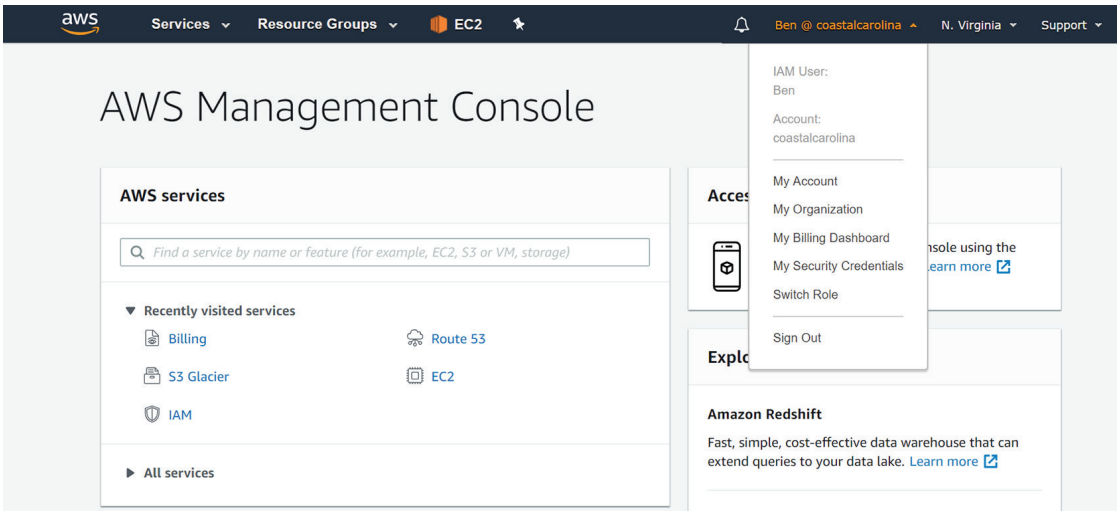
SYBEX®
A Wiley Brand

The Account Name Menu

The account name menu in the navigation bar displays the IAM user and account alias or ID you're logged in as, or the name associated with your AWS account if you're logged in as root. Choosing the menu gives you the following options, as shown in Figure 6.8:

- My Account—Takes you to the Billing service console page that displays your account information
- My Organization—Takes you to the AWS Organizations service console
- My Billing Dashboard—Takes you to the Billing and Cost Management Dashboard
- My Security Credentials—Takes you to the My Password page in the IAM service console where you can change your password
- Switch Role—Lets you assume an IAM role where you can perform tasks as a different principal with different permissions
- Sign Out—Signs you out of the AWS Management Console

FIGURE 6.8 The account name menu when you're logged in as an IAM user

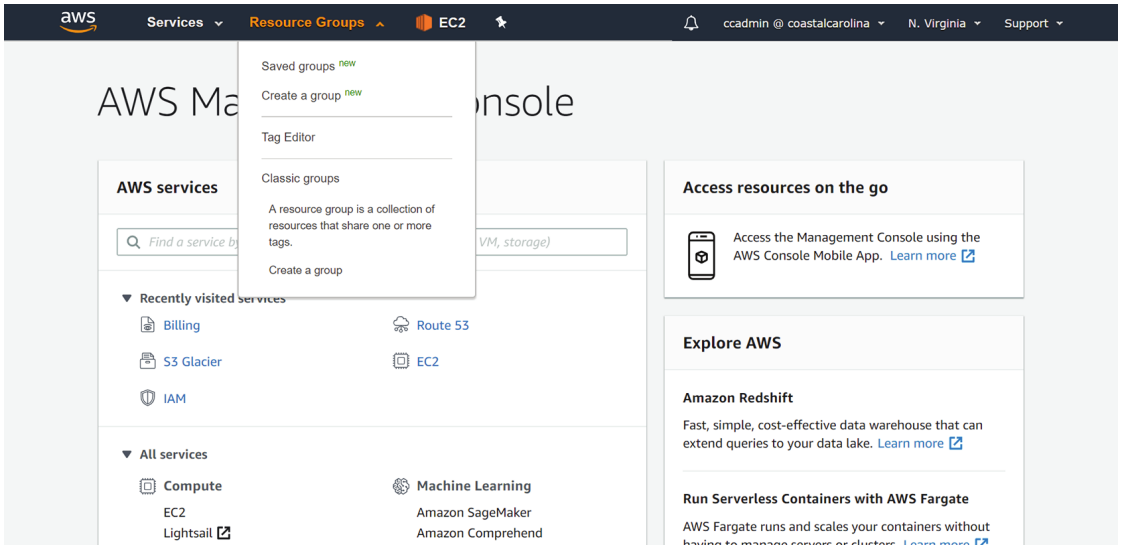


Resource Groups

Resource groups are one of the options readily available from the AWS Management Console. Choosing the Resource Groups link in the navigation bar, as shown in Figure 6.9, will reveal options to create new resource groups and view existing ones.

Resource groups let you view, manage, and automate tasks on multiple AWS resources at a time. They are ideal for grouping AWS resources that all compose a particular application. Rather than having to remember which resources belong to which application, you can add those resources to a resource group and let AWS remember for you! You can then perform bulk actions against those resources.

FIGURE 6.9 The Resource Groups menu



A resource group is a collection of AWS resources in the same region that match the results of a query. You can create a resource group from a query based on resource tags or from a CloudFormation Stack.

Resource tags *Resource tags* are optional metadata that you can assign to AWS resources. A tag must contain a label called a *key* and may optionally contain a value. For example, you may tag your production resources using a key named `Environment` and a value named `Production`. Tag keys and values can include letters, numbers, spaces, and these characters: `+ - = . _ : / @`. You can assign multiple tags to a resource, up to 50. Tag keys and values are case-sensitive. To create a resource group using tags, you can query based on a tag key only or both a tag key and its value.

AWS CloudFormation stacks CloudFormation, which you'll learn about in Chapter 11, "Automating Your AWS Workloads," lets you programmatically deploy and manage multiple AWS resources as a single unit called a *stack*. You can create resource groups that contain some or all of the resources in a CloudFormation stack. You can choose all resources in the stack or filter by resource type, such as EC2 instance, EC2 security group, or S3 bucket.

Tag Editor

If you want to create a resource group based on tags, you'll first need to tag the resources you want to include in the group. Thankfully the AWS Management Console makes this easy. Choose the Resource Groups link in the navigation bar of the AWS Management Console, and then choose Tag Editor, as shown in Figure 6.9.

To get started with Tag Editor, create a query to find the resources you want to tag. At a minimum, you must select at least one region and one or more resource types, such as EC2 instances, EC2 volumes, VPCs, or S3 buckets. You can select all resources if you want. Optionally, you can list resources that already have an existing tag. For example, you

The Declarative Approach

Using a declarative approach, you write code that declares the desired result of the task, rather than how to carry it out. For example, rather than using the AWS CLI to create a new virtual private cloud (VPC) and subnets, you could write declarative code that simply defines the configuration for the VPC and subnets. This approach naturally requires some intelligent software to figure out the operations required to achieve the desired result. CloudFormation, which is covered in the first part of this chapter, is the most well-known AWS service that takes a declarative approach to building and configuring your AWS infrastructure. In short, you write code containing the specifics of the AWS resources you want and present that code to CloudFormation, and it builds and configures those resources on your behalf in a matter of seconds.

Infrastructure as Code

Using code to define your infrastructure and configurations is commonly called the *infrastructure as code* (IaC) approach. Defining IaC is a fundamental element of automation in the cloud. Because automation relies on code being executed by a machine, it reduces the risk of human error inherent in carrying out the same work manually. This in turn reduces rework and other problems down the line.

In this chapter, we cover the following AWS services that enable automation in different ways:

- CloudFormation—Lets you automate the building and configuration of your AWS resources
- The AWS Developer Tools of CodeCommit, CodeBuild, CodeDeploy, and CodePipeline—Automate the testing, building, and deployment of applications to EC2 and on-premises instances
- EC2 Auto Scaling—Automatically launches, configures, and terminates EC2 instances as needed to meet fluctuating demand
- Systems Manager—Automates common operational tasks such as patching instances and backing up Elastic Block Store (EBS) volumes
- OpsWorks—A collection of three different offerings that help automate instance configuration and application deployments using the popular Chef and Puppet configuration management platforms

CloudFormation

CloudFormation automatically creates and configures your AWS infrastructure from code that defines the resources you want it to create and how you want those resources configured.

Templates

The code that defines your resources is stored in text files called *templates*. Templates use the proprietary CloudFormation language, which can be written in JavaScript Object Notation (JSON) or YAML format.

Templates contain a description of the AWS resources you want CloudFormation to create, so they simultaneously function as infrastructure documentation. Because templates are files, you can store them in a version-controlled system such as an S3 bucket or a Git repository, allowing you to track changes over time.

You can use the same template to build AWS infrastructure repeatedly, such as for development or testing. You can also use a single template to build many similar environments. For example, you can use one template to create identical resources for both production and test environments. Both environments can have the same infrastructure—VPCs, subnets, application load balancers, and so on—but they'll be separate environments. CloudFormation uses random identifiers when naming resources to ensure the resources it creates don't conflict with each other.

You can write templates to optionally take parameter inputs, allowing you to customize your resources without modifying the original template. For instance, you can write a template that asks for a custom resource tag to apply to all the resources CloudFormation creates. You can make a parameter optional by specifying a default value, or you can require the user to specify a value in order to use the template.

Stacks

To provision resources from a template, you must specify a stack name that's unique within your account. A *stack* is a container that organizes the resources described in the template.

The purpose of a stack is to collectively manage related resources. If you delete a stack, CloudFormation automatically deletes all of the resources in it. This makes CloudFormation perfect for test and development environments that need to be provisioned as pristine and then thrown away when no longer needed. Deleting a stack helps ensure that all resources are deleted and that no forgotten resources are left lingering, running up charges.

One of the fundamental principles of good cloud design is to make your test environments mirror your production environments as closely as possible. CloudFormation makes this not only possible but almost trivially easy!

Stack Updates

You can have CloudFormation change individual resources in a stack. Just modify the template to delete, modify, or add resources, and then instruct CloudFormation to perform a stack update using the template. CloudFormation will automatically update the resources accordingly. If any other resources are dependent upon a resource that you've updated,

CloudFormation will detect that and make sure those downstream resources are also reconfigured properly.

Alternatively, you can create a change set. Make the desired changes to your template, and then have CloudFormation generate a change set based on the template. CloudFormation will let you review the specific changes it will make, and then you can decide whether to execute those changes or leave everything as is.

CloudFormation makes it easy to update stacks, increasing the possibility that a less skilled user might inadvertently update a stack, making undesired changes to a critical resource. If you're concerned about this possibility, you can create a stack policy to guard against accidental updates. A stack policy is a JSON document, separate from a template, that specifies what resources may be updated. You can use it prevent updates to any or all resources in a stack. If you absolutely need to update a stack, you can temporarily override the policy.

Tracking Stack Changes

Each stack contains a timestamped record of events that occur within the stack, including when resources were created, updated, or deleted. This makes it easy to see all changes made to your stack.

It's important to understand that resources in CloudFormation stacks can be changed manually, and CloudFormation doesn't prevent this. For instance, you can manually delete a VPC that's part of a CloudFormation stack. Drift detection is a feature that monitors your stacks for such changes and alerts you when they occur.

CloudFormation vs. the AWS CLI

You can script AWS CLI commands to create resources fast and automatically, but those resources won't be kept in a stack, so you won't get the same advantages that stacks provide.

When you're using the AWS CLI, you can't update your resources as easily as you can with CloudFormation. With CloudFormation, you simply adjust the template or parameters to change your resources, and CloudFormation figures out how to perform the changes. With the AWS CLI, it's up to you to understand how to change each resource and to ensure that a change you make to one resource doesn't break another.

EXERCISE 11.1

Explore the CloudFormation Designer

In this exercise, you'll use the CloudFormation Designer to look at a sample template that defines a simple environment containing a Linux EC2 instance. In addition to letting you view and edit the template directly, the CloudFormation Designer also visualizes the resources in a template, making it easy to see exactly what CloudFormation will build.

1. Browse to the CloudFormation service console.
2. Choose the Create New Stack button.

EXERCISE 11.1 *(continued)*

3. In the Choose A Template section, choose the Select A Sample Template drop-down, and select LAMP Stack.
4. Choose the View/Edit Template In Designer link.
5. The Designer will show you icons for two resources: an EC2 instance and a security group. Choose the instance icon. The Designer will take you to the section of the template that defines the instance.

AWS Developer Tools

The AWS Developer Tools are a collection of tools designed to help application developers develop, build, test, and deploy their applications onto EC2 and on-premises instances. These tools facilitate and automate the tasks that must take place to get a new application revision released into production.

However, the AWS Developer Tools enable more than just application development. You can use them as part of any IaC approach to automate the deployment and configuration of your AWS infrastructure.

In this section, you'll learn about the following AWS Developer Tools:

- CodeCommit
- CodeBuild
- CodeDeploy
- CodePipeline

CodeCommit

CodeCommit lets you create private Git repositories that easily integrate with other AWS services. Git (`https://git-scm.com`) is a version control system that you can use to store source code, CloudFormation templates, documents, or any arbitrary files—even binary files such as Amazon Machine Images (AMI) and Docker containers. These files are stored in a repository, colloquially known as a *repo*.

Git uses a process called *versioning*, where all changes or commits to a repository are retained indefinitely, so you can always revert to an old version of a file if you need it.

CodeCommit is useful for teams of people who need to collaborate on the same set of files, such as developers who collaborate on a shared source code base for an application. CodeCommit allows users to check out code by copying or cloning it locally to their machine. They make changes to it locally and then check it back in to the repository.

Command Documents

Command documents are scripts that run once or periodically that get the system into the state you want.

Using Command documents, you can install software on an instance, install the latest security patches, or take inventory of all software on an instance. You can use the same Bash or PowerShell commands you'd use with a Linux or Windows instance. Systems Manager can run these periodically, or on a trigger, such as a new instance launch. Systems Manager requires an agent to be installed on the instances that you want it to manage.

Configuration Compliance

Configuration Compliance is a feature of Systems Manager that can show you whether instances are in compliance with your desired configuration state—whether it's having a certain version of an application installed or being up-to-date on the latest operating system security patches.

Automation Documents

In addition to providing configuration management for instances, Systems Manager lets you perform many administrative AWS operations you would otherwise perform using the AWS Management Console or the AWS CLI. These operations are defined using Automation documents. For example, you can use Systems Manager to automatically create a snapshot of an Elastic Block Store (EBS) volume, launch or terminate an instance, create a CloudFormation stack, or even create an AMI from an existing EBS volume.

Distributor

Using Systems Manager Distributor, you can deploy installable software packages to your instances. You create a .zip archive containing installable or executable software packages that your operating system recognizes, put the archive in an S3 bucket, and tell Distributor where to find it. Distributor takes care of deploying and installing the software. Distributor is especially useful for deploying a standard set of software packages that already come as installers or executables.

OpsWorks

OpsWorks is a set of three different services that let you take a declarative approach to configuration management. As explained earlier in this chapter, using a declarative approach requires some intelligence to translate declarative code into imperative operations. OpsWorks uses two popular configuration management platforms that fulfill this requirement: Chef (`https://www.chef.io`) and Puppet Enterprise (`https://puppet.com`).

Puppet and Chef are popular configuration management platforms that can configure operating systems, deploy applications, create databases, and perform just about any configuration task you can dream of, all using code. Both Puppet and Chef are widely used for on-premises deployments, but OpsWorks brings their power to AWS as the OpsWorks managed service.

Just as with automation in general, configuration management is not an all-or-nothing decision. Thankfully, OpsWorks comes in three different flavors to meet any configuration management appetite.

AWS OpsWorks for Puppet Enterprise and AWS OpsWorks for Chef Automate are robust and scalable options that let you run managed Puppet or Chef servers on AWS. This is good if you want to use configuration management across all your instances.

AWS OpsWorks Stacks provides a simple and flexible approach to using configuration management just for deploying applications. Instead of going all-in on configuration management, you can just use it for deploying and configuring applications. OpsWorks Stacks takes care of setting up the supporting infrastructure.

AWS OpsWorks for Puppet Enterprise and AWS OpsWorks for Chef Automate

The high-level architectures of AWS OpsWorks for Puppet Enterprise and Chef Automate are similar. They consist of at least one Puppet master server or Chef server to communicate with your managed nodes—EC2 or on-premises instances—using an installed agent.

You define the configuration state of your instances—such as operating system configurations applications—using Puppet modules or Chef recipes. These contain declarative code, written in the platform's domain-specific language, that specifies the resources to provision. The code is stored in a central location, such as a Git repository like CodeCommit or an S3 bucket.

OpsWorks manages the servers, but you're responsible for understanding and operating the Puppet or Chef software, so you'll need to know how they work and how to manage them. No worries, though, because both Chef and Puppet have large ecosystems brimming with off-the-shelf code that can be used for a variety of common scenarios.

AWS OpsWorks Stacks

If you like the IaC concept for your applications, but you aren't familiar with managing Puppet or Chef servers, you can use AWS OpsWorks Stacks.

OpsWorks Stacks lets you build your application infrastructure in stacks. A *stack* is a collection of all the resources your application needs: EC2 instances, databases, application load balancers, and so on. Each stack contains at least one layer, which is a container for some component of your application.

To understand how you might use OpsWorks Stacks, consider a typical database-backed application that consists of three layers:

- An application layer containing EC2 instances or containers
- A database layer consisting of self-hosted or relational database service (RDS) database instances
- An application load balancer that distributes traffic to the application layer

There are two basic types of layers that OpsWorks uses: OpsWorks layers and service layers.

Despite the name, an OpsWorks stack is not the same as a CloudFormation stack and doesn't use CloudFormation templates.

OpsWorks layers An OpsWorks layer is a template for a set of instances. It specifies instance-level settings such as security groups and whether to use public IP addresses. It also includes an auto-healing option that automatically re-creates your instances if they fail. OpsWorks can also perform load-based or time-based auto scaling, adding more EC2 instances as needed to meet demand.

An OpsWorks layer can provision Linux or Windows EC2 instances, or you can add existing Linux EC2 or on-premises instances to a stack. OpsWorks Stacks supports Amazon Linux, Ubuntu Server, CentOS, and Red Hat Enterprise Linux.

To configure your instances and deploy applications, OpsWorks uses the same declarative Chef recipes as the Chef Automate platform, but it doesn't provision a Chef server. Instead, OpsWorks Stacks performs configuration management tasks using the Chef Solo client that it installs on your instances automatically.

Service layers A stack can also include service layers to extend the functionality of your stack to include other AWS services. Service layers include the following layers:

Relational Database Service (RDS) Using an RDS service layer, you can integrate your application with an existing RDS instance.

Elastic Load Balancing (ELB) If you have multiple instances in a stack, you can create an application load balancer to distribute traffic to them and provide high availability.

Elastic Container Service (ECS) Cluster If you prefer to deploy your application to containers instead of EC2 instances, you can create an ECS cluster layer that connects your OpsWorks stack to an existing ECS cluster.

Summary

If there's one thing that should be clear, it's that AWS provides many different ways to automate the same task. The specific services and approaches you should use are architectural decisions beyond the scope of this book, but you should at least understand how each of the different AWS services covered in this chapter can enable automation.

Fundamentally, automation entails defining a task as code that a system carries out. This code can be written as imperative commands that specify the exact steps to perform the task. The most familiar type of example is the Bash or PowerShell script system administrators write to perform routine tasks. AWS Systems Manager, CodeBuild, and CodeDeploy use an imperative approach. Even the Userdata scripts that you use with EC2 Auto Scaling are imperative.

Code can also be written in a more abstract, declarative form, where you specify the end result of the task. The service providing the automation translates those declarative statements into step-by-step operations that it carries out. CloudFormation, OpsWorks for Puppet Enterprise, OpsWorks for Chef Automate, and OpsWorks Stacks use declarative languages to carry out automation.

At the end of the day, both the imperative and declarative approaches result in nothing more than a set of commands that must be carried out sequentially. It's important to understand that the imperative and declarative approaches are not opposites. Rather, the declarative approach abstracts away from you the step-by-step details in favor of a more results-oriented, user-friendly paradigm. The difference comes down to what approach you prefer and what the service requires.

Configuration management is a form of automation that emphasizes configuration consistency and compliance. Naturally, the focus on achieving and maintaining a certain state lends itself to a declarative approach, so many configuration management platforms such as Puppet and Chef use declarative language. However, AWS Systems Manager also provides configuration management, albeit by using an imperative approach.

Exam Essentials

Know what specific tasks AWS services can automate. CloudFormation can automatically deploy, change, and even delete AWS resources in one fell swoop.

The AWS Developer Tools—CodeCommit, CodeBuild, CodeDeploy, and CodePipeline—can help automate some or all of the software development, testing, and deployment process.

EC2 Auto Scaling automatically provisions a set number of EC2 instances. You can optionally have it scale in or out according to demand or a schedule.

Systems Manager Command documents let you automate tasks against your instance operating systems, such as patching, installing software, enforcing configuration settings, and collecting inventory. Automation documents let you automate many administrative AWS tasks that would otherwise require using the management console or CLI.

OpsWorks for Puppet Enterprise and OpsWorks for Chef Automate also let you configure your instances and deploy software but do so using the declarative language of Puppet modules or Chef recipes.

OpsWorks Stacks can automate the build and deployment of an application and its supporting infrastructure.

Understand the benefits of automation and infrastructure as code. Automation allows common, repetitive tasks to be executed faster than doing them manually and reduces the risk of human error. When you automate infrastructure builds using code, the code simultaneously serves as *de facto* documentation. Code can be placed into version control, making it easy to track changes and even roll back when necessary.

Be able to explain the concepts of continuous integration and continuous delivery. The practice of continuous integration involves developers regularly checking in code as they create or change it. An automated process performs build and test actions against it. This immediate feedback loop allows developers to fix problems quickly and early.

Continuous delivery expands upon continuous integration but includes deploying the application to production after a manual approval. This effectively enables push-button deployment of an application to production.

Review Questions

1. Which of the following is an advantage of using CloudFormation?
   - **A.** It uses the popular Python programming language.
   - **B.** It prevents unauthorized manual changes to resources.
   - **C.** It lets you create multiple separate AWS environments using a single template.
   - **D.** It can create resources outside of AWS.
2. What formats do CloudFormation templates support? (Select TWO.)
   - **A.** XML
   - **B.** YAML
   - **C.** HTML
   - **D.** JSON
3. What's an advantage of using parameters in a CloudFormation template?
   - **A.** Allow customizing a stack without changing the template.
   - **B.** Prevent unauthorized users from using a template.
   - **C.** Prevent stack updates.
   - **D.** Allow multiple stacks to be created from the same template.
4. Why would you use CloudFormation to automatically create resources for a development environment instead of creating them using AWS CLI commands? (Select TWO.)
   - **A.** Resources CloudFormation creates are organized into stacks and can be managed as a single unit.
   - **B.** CloudFormation stack updates help ensure that changes to one resource won't break another.
   - **C.** Resources created by CloudFormation always work as expected.
   - **D.** CloudFormation can provision resources faster than the AWS CLI.
5. What are two features of CodeCommit? (Select TWO.)
   - **A.** Versioning
   - **B.** Automatic deployment
   - **C.** Differencing
   - **D.** Manual deployment
6. In the context of CodeCommit, what can differencing accomplish?
   - **A.** Allowing reverting to an older version of a file
   - **B.** Understanding what code change introduced a bug
   - **C.** Deleting duplicate lines of code
   - **D.** Seeing when an application was last deployed

7. What software development practice regularly tests new code for bugs but doesn't do anything else?
   - **A.** Differencing
   - **B.** Continuous deployment
   - **C.** Continuous delivery
   - **D.** Continuous integration

8. Which CodeBuild build environment compute types support Windows operating systems? (Select TWO.)
   - **A.** build.general2.large
   - **B.** build.general1.medium
   - **C.** build.general1.small
   - **D.** build.general1.large
   - **E.** build.windows1.small

9. What does a CodeBuild environment always contain? (Select TWO.)
   - **A.** An operating system
   - **B.** A Docker image
   - **C.** The Python programming language
   - **D.** .NET Core
   - **E.** The PHP programming language

10. Which of the following can CodeDeploy do? (Select THREE.)
    - **A.** Deploy an application to an on-premises Windows instance.
    - **B.** Deploy a Docker container to the Elastic Container Service.
    - **C.** Upgrade an application on an EC2 instance running Red Hat Enterprise Linux.
    - **D.** Deploy an application to an Android smartphone.
    - **E.** Deploy a website to an S3 bucket.

11. What is the minimum number of actions in a CodePipeline pipeline?
    - **A.** 1
    - **B.** 2
    - **C.** 3
    - **D.** 4
    - **E.** 0

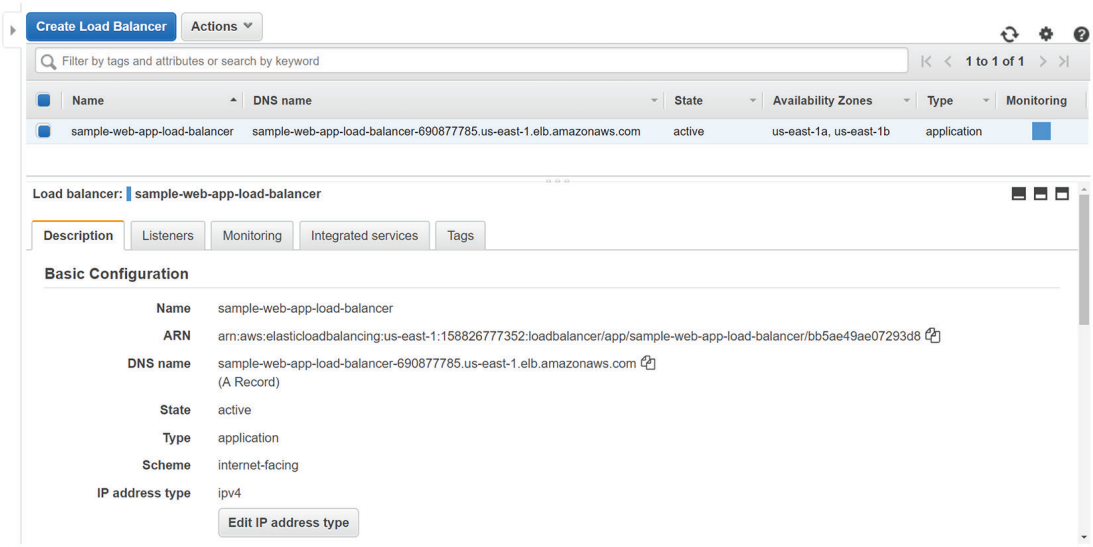
12. You want to predefine the configuration of EC2 instances that you plan to launch manually and using Auto Scaling. What resource must you use?
    - **A.** CloudFormation template
    - **B.** Instance role
    - **C.** Launch configuration
    - **D.** Launch template

EXERCISE 12.2 *(continued)*

17. For Protocol and Port, select HTTP and 80, respectively.
18. Under Health Checks, make sure Protocol and Path are HTTP and /, respectively.
19. Select the button titled Next: Register Targets.
20. The Auto Scaling group that you'll create will add instances to the target group, so there's no need to do that manually here. Select the button titled Next: Review.
21. Review your settings, and select the Create button. It may take a few minutes for your load balancer to become active.

Once AWS has provisioned the load balancer, you should be able to view its publicly resolvable DNS name and other details, as shown in Figure 12.5. Make a note of the DNS name because you'll need it later.

FIGURE 12.5 Application load balancer details



Creating a Launch Template

Before creating the Auto Scaling group, you need to create a launch template that Auto Scaling will use to launch the instances and install and configure the Apache web server software on them when they're launched. Because creating the launch template by hand would be cumbersome, you'll instead let CloudFormation create it by deploying the CloudFormation template at `https://s3.amazonaws.com/aws-ccp/launch-template.yaml`. The launch template that CloudFormation will create will install Apache on each instance that Auto Scaling provisions. You can also create a custom launch template for your own application. Complete Exercise 12.3 to get some practice with CloudFormation and create the launch template.

EXERCISE 12.3

Create a Launch Template

In this exercise, you'll use CloudFormation to deploy an EC2 launch template that Auto Scaling will use to launch new instances.

1. Browse to the CloudFormation service console. Make sure you're in the AWS Region where you want your instances created.
2. Select the Create Stack button.
3. Under Choose A Template, select the radio button titled Specify An Amazon S3 Template URL.
4. In the text field, enter **https://s3.amazonaws.com/aws-ccp/launch-template.yaml**.
5. Select the Next button.
6. In the Stack Name field, enter **sample-app-launch-template**.
7. From the drop-down list, select the instance type you want to use, or stick with the default t2.micro instance type.
8. Select the Next button.
9. On the Options screen, stick with the defaults and select the Next button.
10. Review your settings, and select the Create button.

CloudFormation will take you to the Stacks view screen. Once the stack is created, the status of the sample-app-launch-template stack will show as `CREATE_COMPLETE`, indicating that the EC2 launch template has been created successfully.

Creating an Auto Scaling Group

EC2 Auto Scaling is responsible for provisioning and maintaining a certain number of healthy instances. By default, Auto Scaling provides reliability by automatically replacing instances that fail their EC2 health check. You'll reconfigure Auto Scaling to monitor the ELB health check and replace any instances on which the application has failed.

To achieve performance efficiency and make this configuration cost-effective, you'll create a dynamic scaling policy that will scale the size of the Auto Scaling group in or out between one and three instances, depending on the average aggregate CPU utilization of the instances. If the CPU utilization is greater than 50 percent, it indicates a heavier load, and Auto Scaling will scale out by adding another instance. On the other hand, if the utilization drops below 50 percent, it indicates that you have more instances than you need, so Auto Scaling will scale in. This is called a *target tracking policy*.

EC2 reports these metrics to CloudWatch, where you can graph them to analyze your usage patterns over time. CloudWatch stores metrics for up to 15 months.

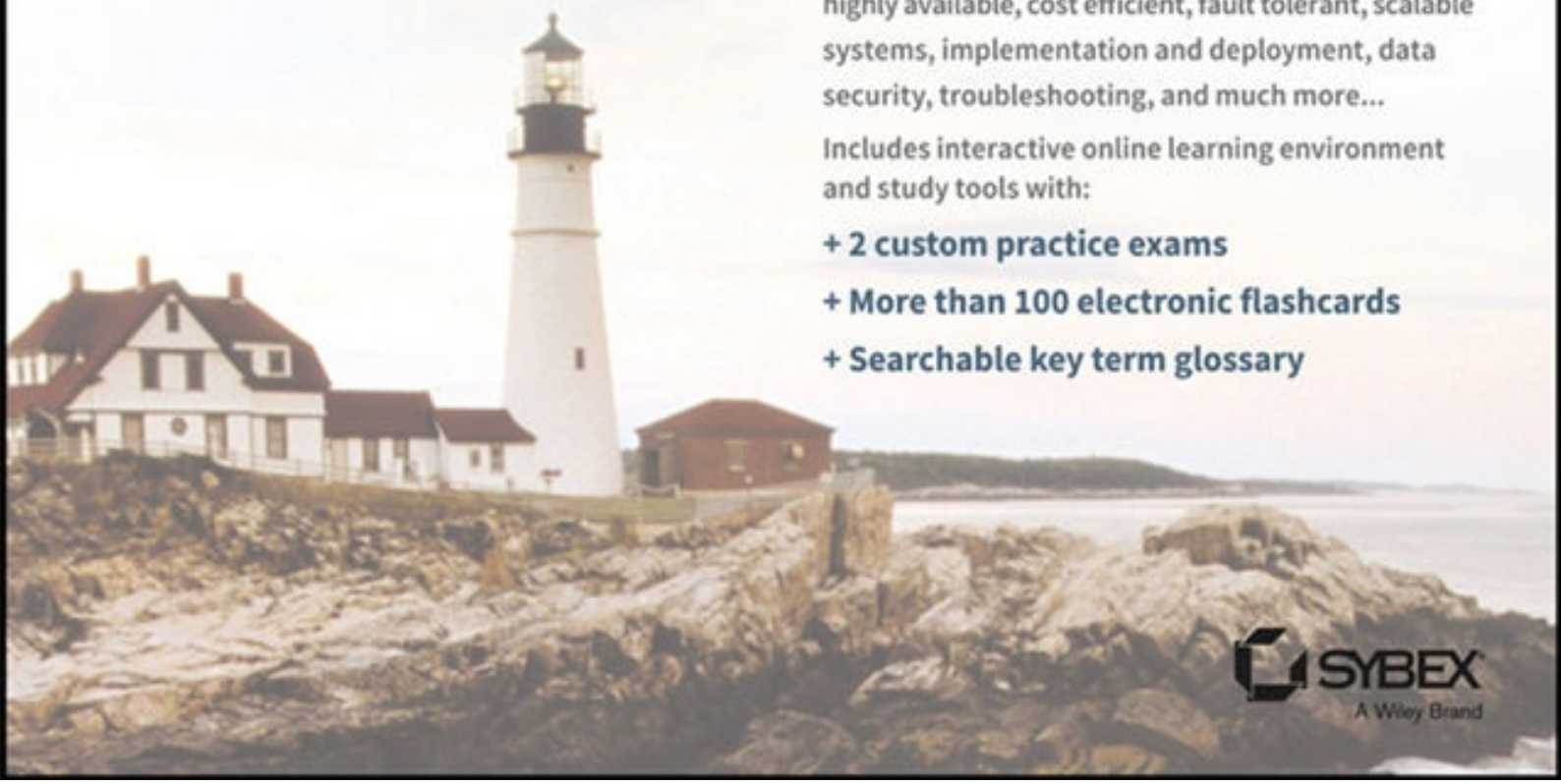
amazon web services

Joe Baron, Hisham Baz, Tim Bixler, Biff Gaut,
Kevin E. Kelly, Sean Senior, John Stamper

AWS Certified Solutions Architect

OFFICIAL STUDY GUIDE

ASSOCIATE EXAM



Covers exam objectives, including designing highly available, cost efficient, fault tolerant, scalable systems, implementation and deployment, data security, troubleshooting, and much more...

Includes interactive online learning environment and study tools with:

+ 2 custom practice exams
+ More than 100 electronic flashcards
+ Searchable key term glossary

SYBEX
A Wiley Brand

columnar storage technology that improves I/O efficiency and parallelizing queries across multiple nodes, Amazon Redshift is able to deliver fast query performance. The Amazon Redshift architecture allows organizations to automate most of the common administrative tasks associated with provisioning, configuring, and monitoring a cloud data warehouse.

Amazon ElastiCache

Amazon ElastiCache is a web service that simplifies deployment, operation, and scaling of an in-memory cache in the cloud. The service improves the performance of web applications by allowing organizations to retrieve information from fast, managed, in-memory caches, instead of relying entirely on slower, disk-based databases. As of this writing, Amazon ElastiCache supports Memcached and Redis cache engines.

Management Tools

AWS provides a variety of tools that help organizations manage your AWS resources. This section provides an overview of the management tools that AWS provides to organizations.

Amazon CloudWatch

Amazon CloudWatch is a monitoring service for AWS Cloud resources and the applications running on AWS. It allows organizations to collect and track metrics, collect and monitor log files, and set alarms. By leveraging Amazon CloudWatch, organizations can gain system-wide visibility into resource utilization, application performance, and operational health. By using these insights, organizations can react, as necessary, to keep applications running smoothly.

AWS CloudFormation

AWS CloudFormation gives developers and systems administrators an effective way to create and manage a collection of related AWS resources, provisioning and updating them in an orderly and predictable fashion. AWS CloudFormation defines a JSON-based templating language that can be used to describe all the AWS resources that are necessary for a workload. Templates can be submitted to AWS CloudFormation and the service will take care of provisioning and configuring those resources in appropriate order (see Figure 1.4).

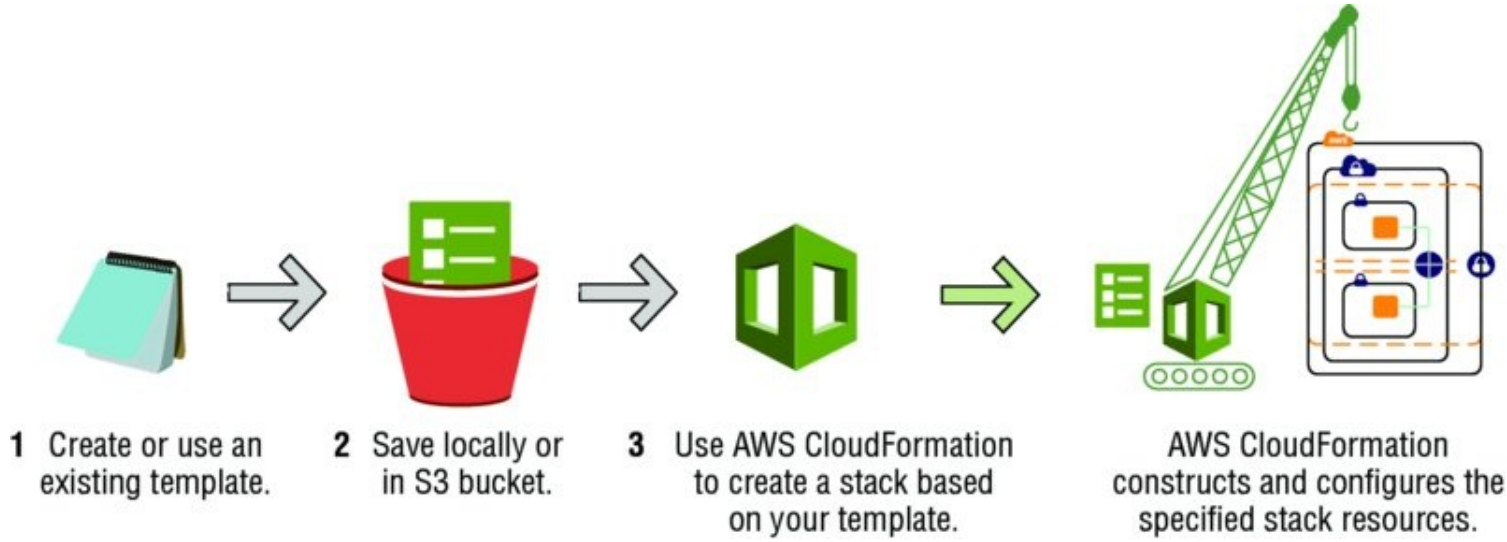


FIGURE 1.4 AWS CloudFormation workflow summary

deploying applications, and running scripts. One of the key AWS OpsWorks features is a set of lifecycle events that automatically run a specified set of recipes at the appropriate time on each instance.

An instance represents a single computing resource, such as an Amazon EC2 instance. It defines the resource's basic configuration, such as operating system and size. Other configuration settings, such as Elastic IP addresses or Amazon EBS volumes, are defined by the instance's layers. The layer's recipes complete the configuration by performing tasks, such as installing and configuring packages and deploying applications.

You store applications and related files in a repository, such as an Amazon S3 bucket or Git repo. Each application is represented by an *app*, which specifies the application type and contains the information that is needed to deploy the application from the repository to your instances, such as the repository URL and password. When you deploy an app, AWS OpsWorks triggers a Deploy event, which runs the Deploy recipes on the stack's instances.

Using the concepts of stacks, layers, and apps, you can model and visualize your application and resources in an organized fashion.

Finally, AWS OpsWorks sends all of your resource metrics to Amazon CloudWatch, making it easy to view graphs and set alarms to help you troubleshoot and take automated action based on the state of your resources. AWS OpsWorks provides many custom metrics, such as CPU idle, memory total, average load for one minute, and more. Each instance in the stack has detailed monitoring to provide insights into your workload.

Use Cases

AWS OpsWorks supports many DevOps efforts, including, but not limited to:

Host Multi-Tier Web Applications AWS OpsWorks lets you model and visualize your application with layers that define how to configure a set of resources that are managed together. Because AWS OpsWorks uses the Chef framework, you can bring your own recipes or leverage hundreds of community-built configurations.

Support Continuous Integration AWS OpsWorks supports DevOps principles, such as continuous integration. Everything in your environment can be automated.

AWS CloudFormation

AWS CloudFormation is a service that helps you model and set up your AWS resources so that you can spend less time managing those resources and more time focusing on your applications that run in AWS. AWS CloudFormation allows organizations to deploy, modify, and update resources in a controlled and predictable way, in effect applying version control to AWS infrastructure the same way one would do with software.

Overview

AWS CloudFormation gives developers and systems administrators an easy way to create and manage a collection of related AWS resources, provisioning and updating them in an orderly

and predictable fashion. When you use AWS CloudFormation, you work with *templates* and *stacks*.

You create AWS CloudFormation templates to define your AWS resources and their properties. A *template* is a text file whose format complies with the JSON standard. AWS CloudFormation uses these templates as blueprints for building your AWS resources.

When you use AWS CloudFormation, you can reuse your template to set up your resources consistently and repeatedly. Just describe your resources once, and then provision the same resources over and over in multiple regions.

When you use AWS CloudFormation, you manage related resources as a single unit called a stack. You create, update, and delete a collection of resources by creating, updating, and deleting stacks. All of the resources in a stack are defined by the stack's AWS CloudFormation template. Suppose you created a template that includes an Auto Scaling group, Elastic Load Balancing load balancer, and an Amazon RDS database instance. To create those resources, you create a stack by submitting your template that defines those resources, and AWS CloudFormation handles all of the provisioning for you. After all of the resources have been created, AWS CloudFormation reports that your stack has been created. You can then start using the resources in your stack. If stack creation fails, AWS CloudFormation rolls back your changes by deleting the resources that it created.

Often you will need to launch stacks from the same template, but with minor variations, such as within a different Amazon VPC or using AMIs from a different region. These variations can be addressed using parameters. You can use parameters to customize aspects of your template at runtime, when the stack is built. For example, you can pass the Amazon RDS database size, Amazon EC2 instance types, database, and web server port numbers to AWS CloudFormation when you create a stack. By leveraging template parameters, you can use a single template for many infrastructure deployments with different configuration values. For example, your Amazon EC2 instance types, Amazon CloudWatch alarm thresholds, and Amazon RDS read-replica settings may differ among AWS regions if you receive more customer traffic in the United States than in Europe. You can use template parameters to tune the settings and thresholds in each region separately and still be sure that the application is deployed consistently across the regions.

Figure 11.8 depicts the AWS CloudFormation workflow for creating stacks.

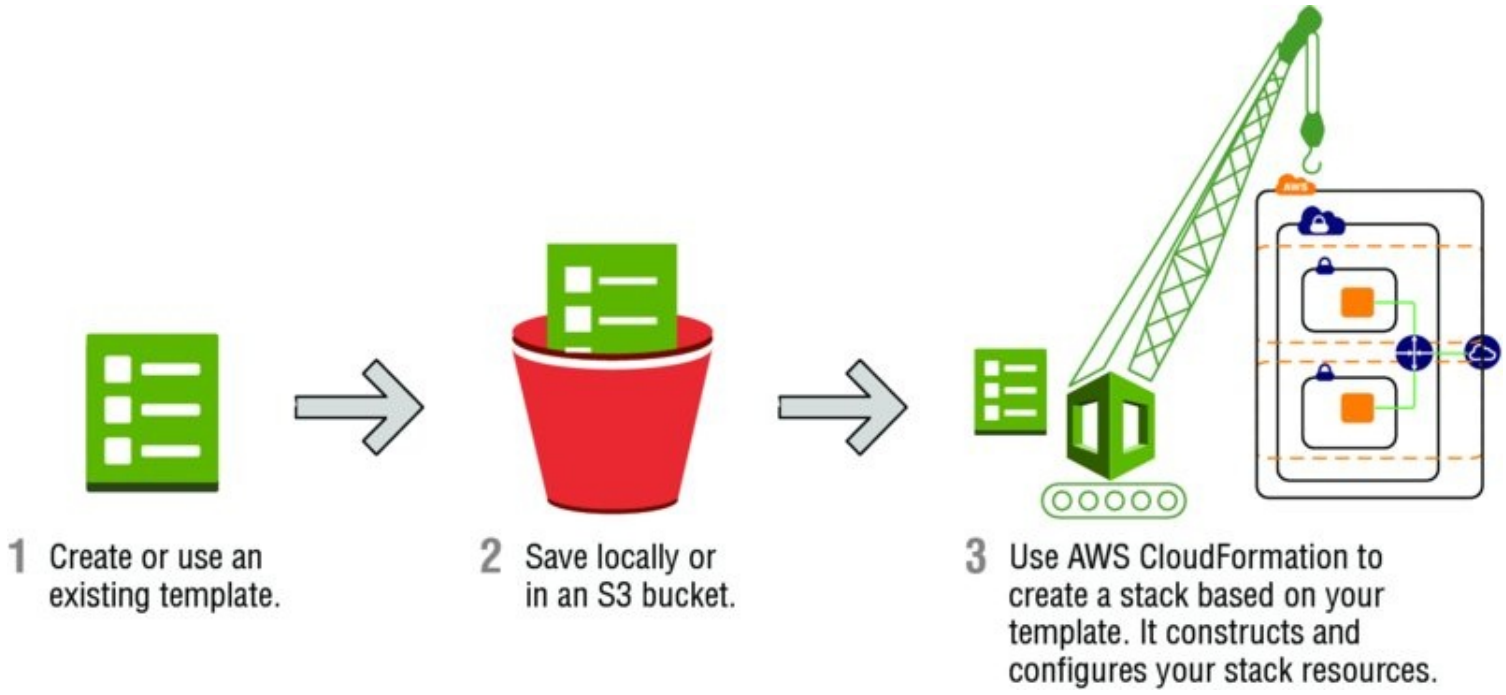


FIGURE 11.8 Creating a stack workflow

Because environments are dynamic in nature, you inevitably will need to update your stack's resources from time to time. There is no need to create a new stack and delete the old one; you can simply modify the existing stack's template. To update a stack, create a *change set* by submitting a modified version of the original stack template, different input parameter values, or both. AWS CloudFormation compares the modified template with the original template and generates a change set. The change set lists the proposed changes. After reviewing the changes, you can execute the change set to update your stack. Figure 11.9 depicts the workflow for updating a stack.

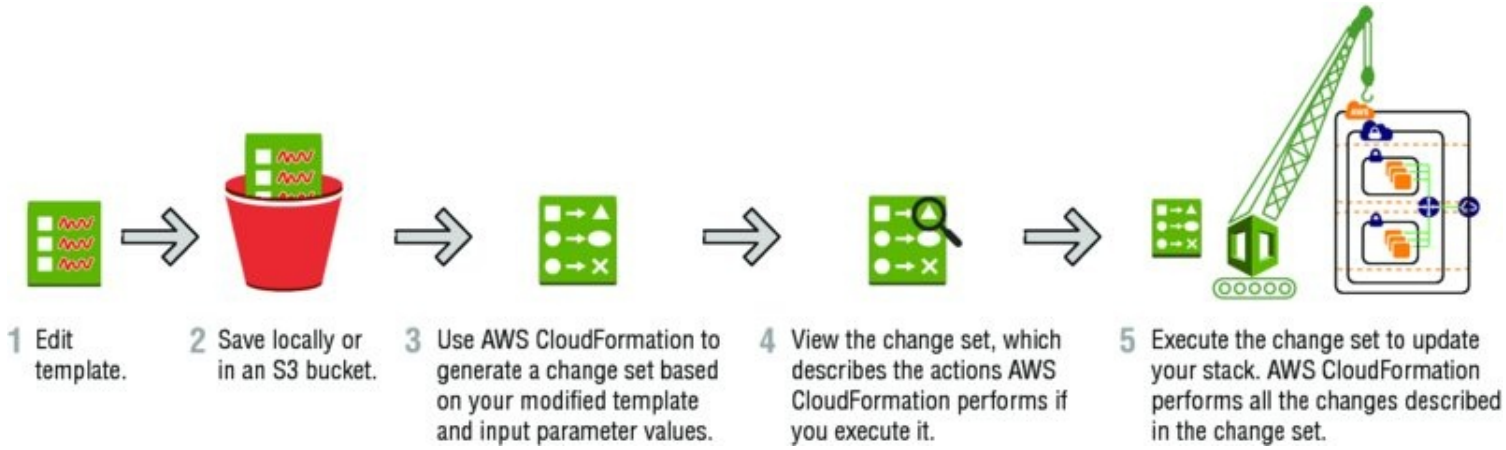


FIGURE 11.9 Updating a stack workflow

When the time comes and you need to delete a stack, AWS CloudFormation deletes the stack and all of the resources in that stack.

If you want to delete a stack but still retain some resources in that stack, you can use a deletion policy to retain those resources. If a resource has no deletion policy, AWS CloudFormation deletes the resource by default.

After all of the resources have been deleted, AWS CloudFormation signals that your stack has been successfully deleted. If AWS CloudFormation cannot delete a resource, the stack will not be deleted. Any resources that haven't been deleted will remain until you can successfully delete the stack.

Use Case

By allowing you to replicate your entire infrastructure stack easily and quickly, AWS CloudFormation enables a variety of use cases, including, but not limited to:

Quickly Launch New Test Environments AWS CloudFormation lets testing teams quickly create a clean environment to run tests without disturbing ongoing efforts in other environments.

Reliably Replicate Configuration Between Environments Because AWS CloudFormation scripts the entire environment, human error is eliminated when creating new stacks.

Launch Applications in New AWS Regions A single script can be used across multiple regions to launch stacks reliably in different markets.

AWS Elastic Beanstalk

AWS Elastic Beanstalk is the fastest and simplest way to get an application up and running on AWS. Developers can simply upload their application code, and the service automatically handles all of the details, such as resource provisioning, load balancing, Auto Scaling, and monitoring.

Overview

AWS comprises dozens of building block services, each of which exposes an area of functionality. While the variety of services offers flexibility for how organizations want to manage their AWS infrastructure, it can be challenging to figure out which services to use and how to provision them. With AWS Elastic Beanstalk, you can quickly deploy and manage applications on the AWS cloud without worrying about the infrastructure that runs those applications. AWS Elastic Beanstalk reduces management complexity without restricting choice or control.

There are key components that comprise AWS Elastic Beanstalk and work together to provide the necessary services to deploy and manage applications easily in the cloud. An *AWS Elastic Beanstalk application* is the logical collection of these AWS Elastic Beanstalk components, which includes environments, versions, and environment configurations. In AWS Elastic Beanstalk, an application is conceptually similar to a folder.

An *application version* refers to a specific, labeled iteration of deployable code for a web application. An application version points to an Amazon S3 object that contains the deployable code. Applications can have many versions and each application version is unique. In a running environment, organizations can deploy any application version they already uploaded to the application, or they can upload and immediately deploy a new application version. Organizations might upload multiple application versions to test differences between one version of their web application and another.

AWS®

Certified Developer

Official Study Guide

Associate (DVA-C01) Exam

AWS CodeDeploy *AWS CodeDeploy* automates code deployments to any instance. It handles the complexity of updating your applications, which avoids downtime during application deployment. It deploys to Amazon EC2 or on-premises servers, in any language and on any operating system. It also integrates with third-party tools and AWS.

Deploying Highly Available and Scalable Applications

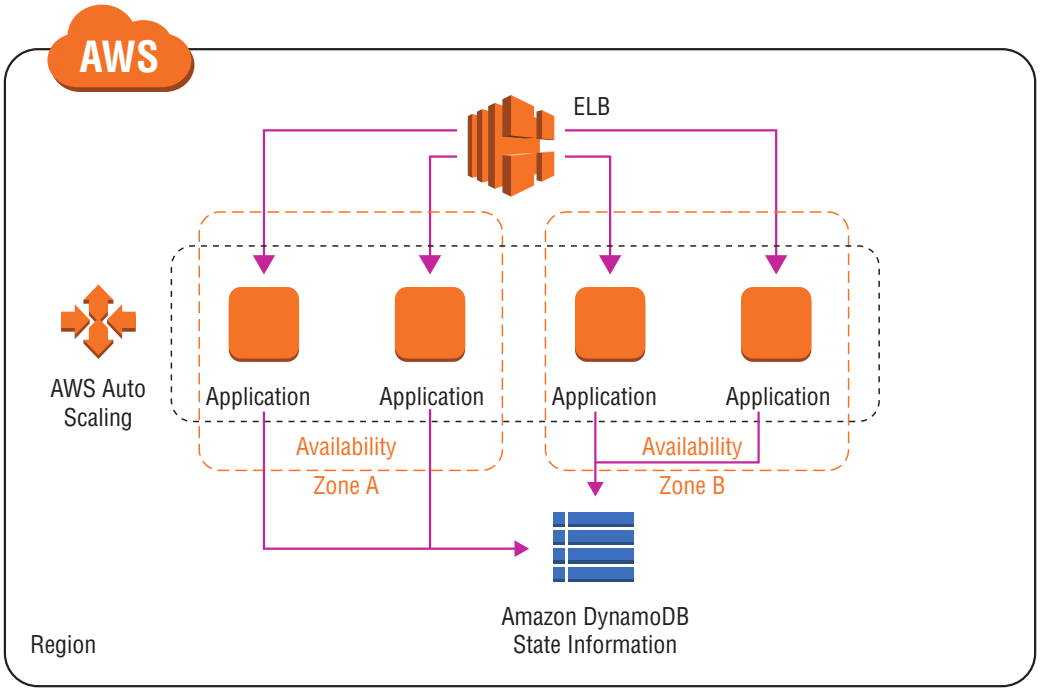
Load balancing is an integral part to directing and managing traffic among your instances. As you launch applications in your environments, you will want them to have high performance and high availability for your users. To enable both of these features, a load balancer will be necessary.

Elastic Load Balancing (ELB) supports three types of load balancers: Application Load Balancers, Network Load Balancers, and Classic Load Balancers. You select a load balancer based on your application needs.

- The *Application Load Balancer* provides advanced request routing targeted at delivery of modern application architectures, including microservices and container-based applications. It simplifies and improves the security of your application by ensuring that the latest Secure Sockets Layer (SSL)/Transport Layer Security (TLS) ciphers and protocols are used at all times. The Application Load Balancer operates at the request level (Layer 7) to route HTTP/HTTPS traffic to its targets: Amazon EC2 instances, containers, and IP addresses based on the content of the request. It is ideal for advanced load balancing of HTTP and HTTPS traffic.
- The *Network Load Balancer* operates at the connection level (Layer 4) to route TCP traffic to targets: Amazon EC2 instances, containers, and IP addresses based on IP protocol data. It is the best option for load balancing of TCP traffic because it's capable of handling millions of requests per second while maintaining ultra-low latencies. Network Load Balancer is optimized to handle sudden and volatile traffic patterns while using a single static IP address per Availability Zone. It is integrated with other popular AWS services, such as AWS Auto Scaling, Amazon Elastic Container Service (Amazon ECS), and AWS CloudFormation. Amazon ECS provides management for deployment, scheduling, and scaling, and management of containerized applications.
- The *Classic Load Balancer* provides basic load balancing across multiple Amazon EC2 instances and operates at both the request level and the connection level. The Classic Load Balancer is intended for applications that were built within the EC2-Classic network. When you're using Amazon Virtual Private Cloud (Amazon VPC), AWS recommends the Application Load Balancer for Layer 7 and Network Load Balancer for Layer 4).

Figure 6.4 displays the flow for deploying highly available and scalable applications.

FIGURE 6.4 Deploying highly available and scalable applications



The flow for deploying highly available and scalable applications includes the following components:

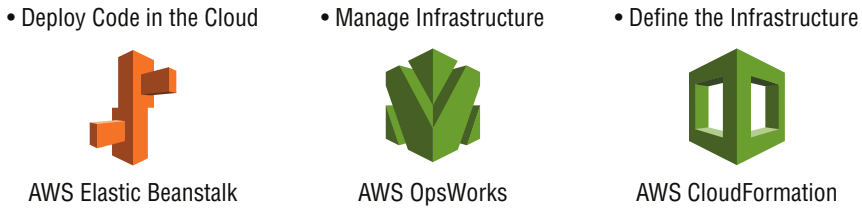
- Multiple Availability Zones and AWS Regions.
- Health check and failover mechanism.
- Stateless application that stores the session state in a cache server or database.
- AWS services that help you to achieve your goal. For example, Auto Scaling helps you maintain high availability and scalability.

Elastic Load Balancing and Auto Scaling are designed to work together.

Deploying and Maintaining Applications

AWS provides several services to manage your application and resources, as shown in Figure 6.5.

FIGURE 6.5 Deployment and maintenance services



With AWS Elastic Beanstalk, you do not have to worry about managing the infrastructure for your application. You deploy your application, such as a Ruby application, in a Ruby container, and Elastic Beanstalk takes care of scaling and managing it.

AWS OpsWorks is a configuration and deployment management tool for your Chef or Puppet resource stacks. Specifically, *OpsWorks for Chef Automate* enables you to manage the lifecycle of your application in layers with Chef recipes. It provides custom Chef cookbooks for managing many different types of layers so that you can write custom Chef recipes to manage any layer that AWS does not support.

AWS CloudFormation is infrastructure as code. The service helps you model and set up AWS resources so that you can spend less time managing them. It is a template-based tool, with formatted text files in JSON or YAML. You can create templates to define what AWS infrastructure you want to build and any relationships that exist among the parts of your AWS infrastructure.

Use AWS CloudFormation templates to provision and configure your stack resources.

Automatically Adjust Capacity

Use AWS Auto Scaling to monitor the AWS resources that are part of your application. The service automatically adjusts capacity to maintain steady, predictable performance. You can build scaling plans to manage your resources, including Amazon EC2 instances and Spot Fleets, Amazon Elastic Container Registry (Amazon ECR) tasks, Amazon DynamoDB tables and indexes, and Amazon Aurora Replicas.

AWS Auto Scaling makes scaling simple, with recommendations that allow you to optimize performance, costs, or balance between them. If you are already using EC2 Auto Scaling to scale your Amazon EC2 instances dynamically, you can now combine it with AWS Auto Scaling to scale additional resources for other AWS services. With AWS Auto Scaling, your applications have the right resources at the right time.

Auto Scaling Groups

An Auto Scaling group contains a collection of Amazon EC2 instances that share similar characteristics. This collection is treated as a logical grouping to manage the scaling of instances. For example, if a single application operates across multiple instances, you might want to increase the number of instances in that group to improve the performance of the application or decrease the number of instances to reduce costs when demand is low.

AWS® Certified Developer Official Study Guide
By Nick Alteen, Jennifer Fisher, Casey Gerena, Wes Gruver, Asim Jalis, Heiwad Osman, Marife Pagan, Santosh Patlolla and Michael Roth

Chapter 8

Infrastructure as Code

THE AWS CERTIFIED DEVELOPER – ASSOCIATE EXAM TOPICS COVERED IN THIS CHAPTER MAY INCLUDE, BUT ARE NOT LIMITED TO, THE FOLLOWING:

Domain 1: Deployment

✓ **1.1 Infrastructure as Code (IaC).**

✓ **1.2 Use AWS CloudFormation to Deploy Infrastructure.**

Domain 5: Monitoring and Troubleshooting

✓ **5.1 Custom Resource Success/Failure.**

Introduction to Infrastructure as Code

Chapter 7 covered deployment tools, processes, and methodologies in AWS services. These services can leverage and be read by *AWS CloudFormation* to provision and manage AWS infrastructure from Amazon Elastic Compute Cloud (Amazon EC2) instances to Amazon API Gateway REST APIs. For all intents and purposes, if you provision and update code with an AWS API, you can use AWS CloudFormation to move this process entirely to template code updates.

If you create an AWS Auto Scaling group of instances with the AWS Management Console, you must perform a number of steps. You can launch and test multiple instances of the user data script with Amazon EC2 launch configurations, you can use Amazon CloudWatch alarms to scale your application, and finally you can implement the AWS Auto Scaling group itself. A better solution is to use AWS CloudFormation to create and manage all of the aforementioned resources over time with a simple, declarative template syntax.

Infrastructure as Code

Using an *infrastructure as code* (IaC) model, instead of manually provisioning or using scripting languages, helps remove the dependency on human intervention when you create and manage infrastructure over time. You can use tools such as AWS CloudFormation to deploy infrastructure from a declarative template syntax. For example, a typical provisioning script that uses the *AWS Command Line Interface* (AWS CLI) includes many procedural steps that are prone to error because of invalid inputs, incorrect command syntax, and resource dependency conflicts. AWS CloudFormation templates provide the ability to validate inputs and automatically detect dependencies between resources.

Provisioning infrastructure with AWS CloudFormation templates provides some built-in benefits, such as the ability to track changes with a "source of truth," such as a Git-based repository. Since repositories track changes over time, you can roll back an undesired change by resubmitting the last working version of the template(s). This can significantly reduce the time needed to roll back undesired changes.

You can view users' resources with appropriate permissions within an AWS account. An issue can arise where, as your infrastructure grows over time, it can be difficult to determine what resources belong to what functional group, application, team, and so on. Use of tags can alleviate this somewhat, but this is not possible for resources that do not yet support tags. AWS CloudFormation organizes resources into stacks, which you describe in the AWS Management Console, the AWS CLI, or AWS software development kits (AWS SDKs). AWS CloudFormation stacks provide a comprehensive list of any infrastructures in a functional group.

Using AWS CloudFormation to Deploy Infrastructure

AWS CloudFormation provides a common language for you to describe and provision all of the infrastructure resources in your cloud environment.

AWS CloudFormation allows you to use a simple text file to model and provision, in an automated and secure manner, all of the resources for your applications across all regions and accounts. This file serves as the single source of truth for your cloud environment.

AWS CloudFormation is available at no additional charge, and you pay only for the AWS resources required to run your applications.

What Is AWS CloudFormation?

Before you deploy any application code, the first requirement is that infrastructure exists where you will deploy the code. AWS CloudFormation aims to alleviate previous deployment issues with the use of a service that allows you to describe your infrastructure with standardized JSON or YAML template syntax. The template contains the infrastructure that AWS will deploy and all the related configuration properties. When you submit this template to the AWS CloudFormation service, it creates a stack, which is a logical group of resources that the template describes.

When you manually create resources with the AWS Management Console or AWS CLI or AWS SDK, you cannot easily define relationships between resources.

If you manually create an AWS Auto Scaling Group (ASG) and attach this to an Elastic Load Balancing (ELB) load balancer, it requires several API calls or console actions—one for each resource and one to attach the ASG to the ELB. With AWS CloudFormation, you define the resources and any relationships in one location for easy deployment and updates over time.

Two key benefits of AWS CloudFormation over procedural scripting or manual console actions are that your infrastructure is now *repeatable* and that it is *versionable*.

Any template that you deploy one time in an account you can deploy again (either in the same account and/or region or in others). This offers you an opportunity for dynamically provisioning short-lived environments to test or roll over to a new production environment (blue/green deployment). Since templates describe your infrastructure, you check the templates themselves into a source code repository. With this, you can track changes over time, and updates roll back when they revert commits and redeploy the previous template(s). Over time, this creates self-documenting infrastructure that shows changes over the lifecycle of an environment.

AWS CloudFormation Concepts

This section details AWS CloudFormation concepts, such as stacks, change sets, permissions, templates, and instinct functions.

Stacks

A *stack* represents a collection of resources to deploy and manage by AWS CloudFormation. When you submit a template, the resources you configure are provisioned and then make up the stack itself. Any modifications to the stack affect underlying resources. For example, if you remove an `AWS::EC2::Instance` resource from the template and update the stack, AWS CloudFormation causes the referred instance to terminate.

AWS CloudFormation manages all of the resources you declare in a stack when the stack updates. If you manually update the resource outside of AWS CloudFormation, the result will be inconsistencies between the state AWS CloudFormation expects and the actual resource state. This can cause future stack operations to fail.

Change Sets

There may be times where you would like to see what changes will occur to resources when you update a template, before the update occurs. Instead of submitting the update directly, you can generate a change set. A *change set* is a description of the changes that will occur to a stack, should you submit the template. If the changes are acceptable, the change set itself can execute on the stack and implement the proposed modifications. This is especially important in situations where there is a potential for data loss.

Amazon Relational Database Service Instances

There are several properties in Amazon Relational Database Service (Amazon RDS) instances that AWS CloudFormation modifies and requires replacement in the underlying database instance resource. If backups are not being taken, data loss will occur. You use a change set to preview the replacement event, make the necessary backups, and take the required precautions before you update the resources.

Permissions

AWS CloudFormation, unless otherwise specified, functions within the context of the IAM user or AWS role to invoke a stack action. This means that if you submit a template that creates an Amazon EC2 instance (or instances), AWS CloudFormation will fail unless your IAM user or AWS role has permissions to create instances. Any action that AWS CloudFormation performs is done on your behalf, with your authorizations. With this, you can control what stack actions perform (create, update, or delete) and what actions are performed on the underlying resources.

If there is a need to restrict what permissions a single IAM user or AWS role can have, you can provide a service role the stack uses for the create, update, or delete actions. When the role passes to AWS CloudFormation, it will use the role's credentials to determine what operations it performs. To create an AWS CloudFormation service role, make sure that the role as a trust policy allows `cloudformation.amazonaws.com` to assume the role.

As a user, your IAM credentials will need to include the ability to pass the role to AWS CloudFormation, using the `iam:PassRole` permission. An additional benefit when you use a service role is that it will extend the default timeout for stack create, update, and delete actions. This is especially important when you work with resources that take a longer time because of their size or distribution. Certain services can time out in AWS CloudFormation, returning a `Resource failed to stabilize` error.

- `AWS::AutoScaling::AutoScalingGroup`
- `AWS::CertificateManager::Certificate`
- `AWS::CloudFormation::Stack`
- `AWS::ElasticSearch::Domain`
- `AWS::RDS::DBCluster`
- `AWS::RDS::DBInstance`
- `AWS::Redshift::Cluster`

After a service role passes to AWS CloudFormation, other users with the ability to perform updates will be able to do so with the same role, regardless of whether they have the ability to pass it. Make sure that the service role follows least-privilege practices.

When you assign permissions for IAM users or AWS roles, you have the ability to specify conditions to control whether policies are in effect. For example, you can allow your users to create stacks only with certain names. However, do not use the `aws:SourceIp` condition. This is because AWS CloudFormation actions originate from AWS IP addresses, not the IP address of the request.

When you create a stack, you can submit a template from a local file or via a URL that points to an object in Amazon S3. If you submit the template as a local file, it uploads to Amazon S3 on your behalf. Because of this, you must add these permissions to create a stack:

- `cloudformation:CreateUploadBucket`
- `s3:PutObject`
- `s3:ListBucket`
- `s3:GetObject`
- `s3:CreateBucket`

Template Structure

AWS CloudFormation uses specific template syntax in JSON or YAML. (The primary difference is YAML's support of comments using the # symbol.) The high-level structure of a template is as follows:

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "String Description",
  "Metadata": { },
  "Parameters": { },
  "Mappings": { },
  "Conditions": { },
  "Transform": { },
  "Resources": { },
  "Outputs": { }
}
```

Of the previous properties, *AWS CloudFormation requires only the `Resources` section*. Each property can be in any order, with the exception that `Description` must follow the `AWSTemplateFormatVersion` command.

AWSTemplateFormatVersion

`AWSTemplateFormatVersion` corresponds to the template version to which this template adheres. Do not confuse this with an API version or the version of the developer's template draft. Currently, AWS CloudFormation only supports the value `"2010-09-09"`, which you must provide as a literal string.

Description

The *`Description`* section allows you to provide a text explanation of the template's purpose or other arbitrary information. The maximum length of the `Description` field is 1,024 bytes. Similar to the `AWSTemplateFormatVersion` section, `Description` supports only literal text.

Metadata

The *Metadata* section of a template allows you to provide structured details about the template. For example, you can provide Metadata about the overall infrastructure to deploy and which sections correspond to certain environments, functional groups, and so on. The Metadata you provide is made available to AWS CloudFormation for reference in other sections of a template or on Amazon EC2 instances being provisioned by AWS CloudFormation.

Updating the Metadata Section of a Template

You cannot update template metadata by itself; you must perform an update to one or more resources when you update the Metadata section of a template.

```
"Metadata": {
  "ApplicationLayer": {
    "Description": "Information about resources in the app layer."
  },
  "DatabaseLayer": {
    "Description": "Information about resources in the DB layer."
  }
}
```

In the Metadata section of the template, you have the ability to specify properties that affect the behavior of different components of the AWS CloudFormation service, such as how template parameters display in the AWS CloudFormation console.

Parameters

You can use *Parameters* to provide inputs into your template, which allows for more flexibility in how this template behaves when you deploy it. Parameter values can be set either when you create the stack or when you perform updates.

The Parameters section must include a unique logical ID (in the next example, InstanceTypeParameter). A parameter must include a value, either a default or one that you provide. Lastly, you cannot reference parameters outside a single template.

AllowedValues Error

This example defines a String parameter named InstanceTypeParameter with a default value of t2.micro. The parameter allows t2.micro, m1.small, or m1.large. The AllowedValues section specifies what options you can select for this parameter in the AWS CloudFormation console. AWS CloudFormation will throw an error if you add a value not in AllowedValues.

```
"Parameters": {
  "InstanceTypeParam": {
    "Type": "String",
```

(continued)

(continued)

```
    "Default": "t2.micro",
    "AllowedValues": [ "t2.micro", "m1.small", "m1.large" ],
    "Description": "Enter t2.micro, m1.small, or m1.large. Default is t2.micro."
  }
}
```

Once you specify a parameter, you can use it within the template using the `Ref` intrinsic function. When AWS CloudFormation evaluates it, the `Ref` statement converts it to the value of the parameter.

```
"EC2Instance": {
  "Type": "AWS::EC2::Instance”,
  "Properties": {
    "InstanceType": { "Ref": "InstanceTypeParam" },
    "ImageId": "ami-12345678"
  }
}
```

AWS CloudFormation supports the following parameter types:

- String
- Number
- List of numbers
- Comma-delimited list
- AWS parameter types
- AWS Systems Manager Parameter Store (Systems Manager) parameter types

If a parameter value is sensitive, you can add the `NoEcho` property. When this is set, the parameter value displays as asterisks (`***`) for any `cloudformation:Describe*` calls. Within the template itself, the value will resolve to the actual input when making `Ref` calls.

AWS parameter types When you use *AWS parameter types*, AWS CloudFormation automatically queries existing properties and values within your AWS account. This can include information such as Amazon EC2 key pair names, IDs of resources, AWS regions/availability zones, or other properties of your account. These input values must exist in your account and are validated to ensure that they are correct. For example, you can use the `AWS::EC2::KeyPair::KeyName` parameter type to require a valid Amazon EC2 key pair. This way, there is reduced risk that a user will input an incorrect value that results in improper stack behavior.

AWS System Manager parameter types *AWS Systems Manager parameter types* can reference parameters that exist in the AWS Systems Manager Parameter Store. If you specify a parameter key, AWS CloudFormation will search your Systems Manager Parameter Store for the correct value and input this into the stack. When you perform stack updates, AWS CloudFormation queries the same key again and could result in a new value for the AWS CloudFormation parameter.

Mappings

You can use the *Mappings* section of a template to create a rudimentary lookup tables that you can reference in other sections of your template when you create the stack.

A common example of mappings usage is to look up Amazon EC2 instance AMI IDs based on the region and architecture type. Note in the following example that mappings entries may contain only string values. (Mappings does not support parameters, conditions, or intrinsic functions.)

```
"Mappings" : {
  "RegionMap" : {
    "us-east-1"      : { "32" : "ami-6411e20d", "64" : "ami-7a11e213" },
    "us-west-1"      : { "32" : "ami-c9c7978c", "64" : "ami-cfc7978a" },
    "eu-west-1"      : { "32" : "ami-37c2f643", "64" : "ami-31c2f645" },
    "ap-southeast-1" : { "32" : "ami-66f28c34", "64" : "ami-60f28c32" },
    "ap-northeast-1" : { "32" : "ami-9c03a89d", "64" : "ami-a003a8a1" }
  }
}
```

After you declare the Mappings section, you can query the values within the mapping with the Fn::FindInMap intrinsic function. The example shows an Fn::FindInMap call that queries the AMI ID based on region and architecture type (32- or 64-bit). If the region was us-east-1, for example, the previous template snippet would resolve to ami-6411e20d.

Pseudo Parameter: AWS::Region

The AWS::Region reference is a pseudoparameter; that is, it's a parameter that AWS defines automatically on your behalf. The AWS::Region parameter, for example, resolves to the region code where the stack is being deployed (such as us-east-1).

```
"Resources" : {
  "myEC2Instance" : {
    "Type" : "AWS::EC2::Instance",
    "Properties" : {
      "ImageId" : { "Fn::FindInMap" : [ "RegionMap", { "Ref" : "AWS::Region" },
"32"]},
      "InstanceType" : "m1.small"
    }
  }
}
```

Conditions

You can use Conditions in AWS CloudFormation templates to determine when to create a resource or when a property of a resource is defined (either in the Resources or Outputs section of the stack). Conditional statements make use of intrinsic functions to evaluate multiple inputs against one other.

A common use case for this would be to conditionally set an Amazon EC2 instance to use a larger instance type if the environment to which you deploy is `prod` versus `dev`. The environment type is input as a template parameter, `EnvType`, which the conditional statement, `CreateProdResources`, uses. The conditional statement decides whether to create an additional Amazon Elastic Block Store (Amazon EBS) volume and mount it to the instance with the `Condition` property of the resource.

A single condition can reference input parameters, mappings, or other conditions to determine whether the final value is true or false.

```
{
  "AWSTemplateFormatVersion" : "2010-09-09",
  "Mappings" : {
    "RegionMap" : {
      "us-east-1"      : { "AMI" : "ami-7f418316", "TestAz" : "us-east-1a" },
      "us-west-1"      : { "AMI" : "ami-951945d0", "TestAz" : "us-west-1a" },
      "us-west-2"      : { "AMI" : "ami-16fd7026", "TestAz" : "us-west-2a" }
    }
  },
  "Parameters" : {
    "EnvType" : {
      "Description" : "Environment type.",
      "Default" : "test",
      "Type" : "String",
      "AllowedValues" : ["prod", "test"]
    }
  },
  "Conditions" : {
    "CreateProdResources" : {"Fn::Equals" : [{"Ref" : "EnvType"}, "prod"]}
  },
  "Resources" : {
    "EC2Instance" : {
      "Type" : "AWS::EC2::Instance",
      "Properties" : {
        "ImageId" : { "Fn::FindInMap" : [ "RegionMap", { "Ref" : "AWS::Region"
}, "AMI" ]}
      }
    },
    "MountPoint" : {
      "Type" : "AWS::EC2::VolumeAttachment",
      "Condition" : "CreateProdResources",
```

```
      "Properties" : {
        "InstanceId" : { "Ref" : "EC2Instance" },
        "VolumeId"  : { "Ref" : "NewVolume" },
        "Device" : "/dev/sdh"
      }
    },
    "NewVolume" : {
      "Type" : "AWS::EC2::Volume",
      "Condition" : "CreateProdResources",
      "Properties" : {
        "Size" : "100",
        "AvailabilityZone" : { "Fn::GetAtt" : [ "EC2Instance",
"AvailabilityZone" ]}
      }
    }
  }
}
```

You can also use `Conditions` to declare different resource properties based on whether the condition evaluates to true with the `Fn::If` intrinsic function. The following example uses the `UseDBSnapshot` condition to determine whether to pass a value to the `DBSnapshotIdentifier` property of an `AWS::RDS::DBInstance` resource. You use the `AWS::NoValue` pseudoparameter in place of a `null` value in AWS CloudFormation templates. When you provide it as a value to a resource property, `AWS::NoValue` removes that property declaration.

```
"MyDB" : {
  "Type" : "AWS::RDS::DBInstance",
  "Properties" : {
    "AllocatedStorage" : "5",
    "DBInstanceClass" : "db.m1.small",
    "Engine" : "MySQL",
    "EngineVersion" : "5.5",
    "MasterUsername" : { "Ref" : "DBUser" },
    "MasterUserPassword" : { "Ref" : "DBPassword" },
    "DBParameterGroupName" : { "Ref" : "MyRDSParamGroup" },
    "DBSnapshotIdentifier" : {
      "Fn::If" : [
        "UseDBSnapshot",
        {"Ref" : "DBSnapshotName"},
        {"Ref" : "AWS::NoValue"}
      ]
    }
  }
}
```

Transforms

As templates grow in size and complexity, there may be situations where you use certain components repeatedly across multiple templates, such as common resources or mappings. Transforms allow you to simplify the template authoring process through a powerful set of macros you use to reduce the amount of time spent in the authoring process. AWS CloudFormation transforms first create a change set for the stack. Transforms are applied to the template during the change set creation process.

Once a change set is complete, the template updates with output of the executed macros. The finalized template deploys to AWS CloudFormation, not the original with the transform declarations. This can cause confusion, as the original template will not be available via the console or AWS CLI or AWS SDK actions.

There are two types of supported transforms.

`AWS::Include Transform` `AWS::Include Transform` acts as a tool to import template snippets from Amazon S3 buckets into the template being developed. When the template is evaluated, a change set is created, and the template snippet is copied from its location and is added to the overall template structure. You can use this transform anywhere in a template, except the `Parameters` and `AWSTemplateFormatVersion` sections.

When you use the `AWS::Include Transform` at the top level of a template, the syntax must match the example. (Note that the transform is declared as `Transform`.) This is especially useful if there is a set of common mappings that you use across multiple teams or template authors, as they can share this set and update it in one location.

```
{
  "Transform" : {
    "Name" : "AWS::Include",
    "Parameters" : {
      "Location" : "s3://MyAmazonS3BucketName/MyFileName.json"
    }
  }
}
```

When you use a transform in nested sections of a template, such as the `Properties` section of an `AWS::EC2::Instance` resource, use the following syntax. (Note that this is now an intrinsic function call.)

```
{
  "Fn::Transform" : {
    "Name" : "AWS::Include",
    "Parameters" : {
      "Location" : "s3://MyAmazonS3BucketName/MyFileName.json"
    }
  }
}
```

When you process stack updates, the template snippets you reference in any transforms pull from their Amazon S3 locations. This means that if a snippet updates without your knowledge, the updated snippet will import into the template. We recommend that you create change sets first so that any accidental updates can be caught before you deploy.

AWS CloudFormation does not support nested transforms. If the snippet being imported into a template includes an additional transform declaration, the stack creation or update will fail.

`AWS::Serverless Transform` You can use the `AWS::Serverless Transform` to convert AWS Serverless Application Model (AWS SAM) templates to valid AWS CloudFormation templates for deployment. AWS SAM uses an abbreviated template syntax to deploy serverless applications with AWS Lambda, Amazon API Gateway, and Amazon DynamoDB.

The following example creates a function that uses the serverless transform. When AWS CloudFormation evaluates the transform, the transform expands the template to include an AWS Lambda function and its IAM execution role.

```
Transform: AWS::Serverless-2016-10-31
Resources:
  MyServerlessFunctionLogicalID:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs4.3
      CodeUri: 's3://testBucket/mySourceCode.zip'
```

Resources

The *`Resources`* section of an AWS CloudFormation template declares the actual AWS resources to be provisioned and their properties. *AWS CloudFormation requires this template section when you create stacks*. The `Resources` section follows a standard syntax, where a logical ID acts as the resource key and type/properties subkeys define the actual type of resource to deploy and what properties it should have.

The *logical ID* of the resource allows it to be referenced in other parts of a template. You can refer to `Resources` in other sections of a template, build relationships between interdependent resources, output property values of the resources, perform other useful functions. The `Resource Type` defines the actual type of resource being managed. For example, an Amazon S3 bucket type is `AWS::S3::Bucket`. There are too many resource types available to list in this book, and they are updated regularly. Check the AWS CloudFormation documentation for available resource types.

```
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-
resource-type-ref.html
```

The resource properties section defines what configuration a resource should have. In the same example, the `AWS::S3::Bucket` resource has an optional property called `BucketName`, which defines the name of the bucket to create.

```
{
  "Resources": {
    "MyBucket": {
      "Type": "AWS::S3::Bucket",
      "Properties": {
        "BucketName": "MyBucketName1234"
      }
    }
  }
}
```

Resource properties are either optional or required and may be any of the following types:

- String
- List of strings
- Boolean
- References to parameters or pseudoparameters
- Intrinsic functions

Outputs

`Outputs` are values that can be made available to use outside a single stack. You can reference these values in a number of different ways, such as cross-stack references, nested stacks, describe-stack API calls, or in the AWS CloudFormation console. Outputs are useful in providing meaningful information after a stack has been created or updated successfully. For example, it would be helpful to output an Elastic Load Balancing load balancer URL to the user when a web application stack deploys successfully.

The basic structure for AWS CloudFormation outputs follows. Similar to resources, outputs must have a logical ID so that AWS CloudFormation can reference them. The `Description` field provides a friendly explanation of the purpose of the output, which can be useful to users of your template. The value being returned can be produced using intrinsic functions, or it can be a static string value. Lastly, the `Export` key (optional) creates cross-stack references.

Here is an example of outputting the ELB load balancer URL:

```
"Outputs" : {
  "BackupLoadBalancerDNSName" : {
    "Description": "The DNSName of the backup load balancer",
    "Value" : { "Fn::GetAtt" : [ "BackupLoadBalancer", "DNSName" ]}
  }
}
```

Intrinsic Functions

Situations can occur where values input into a template cannot be determined until the stack or change set actually is created. If you create an Amazon RDS instance, which is referenced in a configuration file added to an Amazon EC2 instance in the same template, the actual database connection string cannot be determined until the database instance is created. Other attributes, settings, or values may need to be calculated from several inputs at once.

Intrinsic functions aim to resolve this issue by adding dynamic functionality into AWS CloudFormation templates. Multiple intrinsic functions are available to add significant power and flexibility to your templates.

Fn::Base64

The *`Fn::Base64`* intrinsic function converts an input string into its `Base64` equivalent. The primary purpose of this function is to pass instructions written in string format to an Amazon EC2 instance's `UserData` property.

```
{ "Fn::Base64": valueToEncode }
```

Fn::Cidr

When you create Amazon VPCs and subnets, you must provide Classless Inter-Domain Routing (CIDR) blocks to map a group of IP addresses to the resource being created. The *`Fn::Cidr`* intrinsic function allows you to convert an IP address block, subnet count, and size mask (optional) into valid CIDR notation.

```
{ "Fn::Cidr": [ ipBlock, count, sizeMask ] }
```

Fn::FindInMap

After you create mappings in AWS CloudFormation, you use the *`Fn::FindInMap`* intrinsic function to query information stored within the mapping table. Note that mappings have two key levels, and thus top-level and second-level keys must be supplied as inputs, along with the mapping name itself.

```
{ "Fn::FindInMap": [ "MapName", "TopLevelKey", "SecondLevelKey" ] }
```

Consider the following `Mappings` section. The `Fn::FindInMap` call would return `ami-c9c7978c`.

```
"Mappings" : {
  "RegionMap" : {
    "us-east-1" : { "32" : "ami-6411e20d", "64" : "ami-7a11e213" },
    "us-west-1" : { "32" : "ami-c9c7978c", "64" : "ami-cfc7978a" },
    "eu-west-1" : { "32" : "ami-37c2f643", "64" : "ami-31c2f645" }
  }
}

. . .

{ "Fn::FindInMap" : [ "RegionMap", { "Ref" : "AWS::Region" }, "32" ] }
```

Fn::GetAtt

Resources you create in AWS CloudFormation contain information that you can query in other parts of the same template. For example, if you create an IAM role to use when log in to AWS CloudTrail events to Amazon CloudWatch Logs, you must provide the Amazon Resource Name (ARN) of the AWS role to the trail configuration. Since the ARN is not returned when you use the `Ref` intrinsic function (this returns the role name), you can use *`Fn::GetAtt`* to query additional resource properties. In this case, you would be able to use this intrinsic function to determine the ARN of the role.

```
{ "Fn::GetAtt" : [ "logicalIDOfResource", "attributeName" ] }
```

Fn::GetAZs

For each AWS region, different availability zones (with different names) are available. The specific availability zones will not always match between different accounts (in fact, two accounts with the same availability zone by name may not use the same physical location). Because of this, it is not easy to determine which availability zones are usable when you create a stack. The *`Fn::GetAZs`* intrinsic function returns a list of availability zones for the account in which the stack is being created.

```
{ "Fn::GetAZs" : "region" }
```

Only availability zones where a default subnet exists will be returned by `Fn::GetAZ`.

To increase flexibility further and remove the need to hard-code a region in the template, you can use the `AWS::Region` pseudoparameter to return the list of availability zones for the region in which the stack is being created.

```
{ "Fn::GetAZs" : { "Ref": "AWS::Region" } }
```

Fn::Join

In some situations, string values must be concatenated from multiple input strings, as is the case when building Java Database Connectivity (JDBC) connection strings. AWS CloudFormation supports string concatenation with the *`Fn::Join`* intrinsic function. You can join string values with a predefined delimiter, which you supply to the function along with a list of strings to join.

When you define the `UserData` for an `AWS::EC2::Instance` resource, it may be required that you add various parameters to commands being run on the instance.

Fn::Join Appending Data Dynamically

This example shows how you can use `Fn::Join` to append various data dynamically to create complex commands.

```
"Resources" : {
    "Ec2Instance" : {
      "Type" : "AWS::EC2::Instance",
```

```
      "Properties" : {
        "ImageId" : "ami-12345678",
        "Tags" : [ {"Key" : "Role", "Value" : "Test Instance"}],
        "UserData" : { "Fn::Base64" : { "Fn::Join" : [ "", [
          "#!/bin/bash -ex", "\n",
          "echo deploying into region: ", { "Ref": "AWS::Region" }, "\n",
          "\n", "yum install ec2-net-utils -y", "\n",
          "ec2ifup eth1", "\n",
          "service httpd start" ] ] }
      }
    }
  }
}
```

Fn::Select

If you pass a list of values into your template, there needs to be a way to select an item from the list based on what position (index) it is in the list. *Fn::Select* allows you to choose an item in a list based on the zero-based index.

The Fn::Select intrinsic function does not check for issues such as whether an index is out of bounds or whether the values in a list equal null. You need to verify that the input list does not contain null values and has a known length.

```
{ "Fn::Select" : [ index, listOfObjects ] }
```

Fn::Split

Counter to the Fn::Join intrinsic function, you use *Fn::Split* to create a list of strings by separating a single string by a known delimiter. You can use Fn::Select to access the output list of strings and pass them to an index to select from different substrings.

```
{ "Fn::Split" : [ "delimiter", "source string" ] }
```

Fn::Sub

If you need to build an input string with multiple variables determined at runtime, use the *Fn::Sub* function to populate a template string with input variables from a variable map.

This intrinsic function can also use parameters, resources, and resource attributes already present in your template. Note in the following example that two template values are present in the string, but only one mapping value is provided. This is because the

AWS::AccountId pseudoparameter will automatically resolve to the account ID where the stack is being created, and AWS::Region automatically resolves to the region ID.

```
{
  "Fn::Sub": [ "arn:aws:ec2:${AWS::Region}:${AWS::AccountId}:vpc/${vpc}", {
    "vpc": { "Ref": "MyVPC" }
  }
}
```

Ref

You will use the *Ref* intrinsic function a lot within your template, especially when multiple resources have dependencies and relationships between one another (such as if you create an AWS::EC2::VPC resource with two AWS::EC2::Subnet resources). The behavior of the Ref function can differ slightly depending on the resource type being referenced. In some cases, such as with AWS::S3::Bucket or AWS::AutoScaling::AutoScalingGroup resources, you use Ref to return the resource name (in this situation, either the bucket or AWS Auto Scaling group name). In other cases, different properties such as the resource ARN or physical ID returns. Make sure to check the documentation for the resource type being referenced to verify what data returns.

```
{ "Ref" : "logicalName" }
```

Condition Functions

Condition functions are special intrinsic functions for which you can optionally create resources or set resource properties, depending on whether the condition evaluates to true or false. Other than Fn::If, you must use all other condition functions within the Conditions section of a template. The Fn::If intrinsic function allows you to pass different data to resource properties depending on the state of the referenced condition.

FN::AND

Returns true only if all contained conditions evaluate to true; otherwise, false returns.

```
"Fn::And": [{condition}, {...}]
```

FN::EQUALS

Returns true if both compared values are equal; otherwise, false returns.

```
"Fn::Equals" : ["value_1", "value_2"]
```

FN::IF

Returns one of two values, depending on whether the specified condition evaluates to true or false. If you would like to return a null value, pass a reference the AWS::NoValue pseudoparameter with the Ref intrinsic function.

```
"Fn::If": [condition_name, value_if_true, value_if_false]
```

FN::NOT

Acts as a negation, returning the opposite of the evaluated condition.

```
"Fn::Not": [{condition}]
```

FN::OR

Returns `true` if any of the provided conditions are `true`. Otherwise, `false` returns.

```
"Fn::Or": [{condition}, {...}]
```

Built-in Metadata Keys

This section details built-in metadata keys for `AWS::CloudFormation:Init`, `AWS::CloudFormation::Interface`, and `AWS::CloudFormation::Designer`.

AWS::CloudFormation::Init

This section defines what operations the `cfn-init` helper script performs on Amazon EC2 instances being provisioned by AWS CloudFormation (either as stand-alone instances or in AWS Auto Scaling groups). This metadata key allows you to develop a more declarative infrastructure configuration, instead of having to procedurally script every individual action (such as installing packages, which can vary based on the instance's operating system).

This `Metadata` section is organized by config keys, which contain a list of configurations to apply.

AWS::CloudFormation::Init: Resource Metadata

Unless otherwise specified, AWS CloudFormation will look for `config` wherever the `AWS::CloudFormation::Init Metadata` section appears.

```
"Resources": {
  "MyInstance": {
    "Type": "AWS::EC2::Instance",
    "Metadata" : {
      "AWS::CloudFormation::Init" : {
        "config" : {
          "packages" : { },
          "groups" :   { },
          "users" :    { },
          "sources" :  { },
          "files" :    { },
          "commands" : { },
          "services" : { }
        }
      }
    },
    "Properties": { }
  }
}
```

PACKAGES

The packages key allows installation of arbitrary packages on the system. Packages must be available to one of the supported package managers (yum, apt, python, and others). Packages nest under the supported package manager and include a package name followed by an optional version string (or list of versions). If you do not provide a version, the version installs. If the package is not available in the package manager repository, you must include a download URL.

```
"packages": {
  "rpm" : {
    "epel" : "http://download.fedoraproject.org/pub/epel/5/i386/
epel-release-5-4.noarch.rpm"

  },
  "yum" : {
    "httpd" : [],
    "php" : [],
    "wordpress" : []
  }
}
```

On Windows systems, the packages key only supports MSI installers.

GROUPS

Use the groups key to generate Linux/UNIX groups on the target system. The name of the group is derived from the key name, and you can provide an optional group ID. For example, you create two groups with the following syntax. The first group, groupOne, randomly generates the gid value. The second group, groupTwo, will be assigned a gid of 45.

```
"groups" : {
    "groupOne" : {},
    "groupTwo" : { "gid" : "45" }
}
```

Windows systems do not support the groups key.

USERS

The users key allows you to create Linux/UNIX users on your instance. By default, users you create with this key are noninteractive system users, and their default shell is set to /sbin/nologon. If you want to modify this behavior, you will have to issue a separate command on the system after the user generates.

```
"users" : {
    "myUser" : {
        "groups" : ["groupOne", "groupTwo"],
        "uid" : "50",
        "homeDir" : "/tmp"
    }
}
```

Windows systems do not support the `users` key.

SOURCES

Similar in operation to the `files` key, you use the `sources` key to download files from remote locations. However, the `sources` key supports unpacking archives into target directories on the instance. For example, to download and unpack an archive hosted in a public Amazon S3 bucket, use this snippet:

```
"sources" : {
  "/etc/myapp" : "https://s3.amazonaws.com/mybucket/myapp.tar.gz"

}
```

FILES

The `files` key creates files based on either inline content in the template or content from a remote location (URL). An example of inline file content written to `/tmp/setup.mysql` is as follows:

```
"files" : {
  "/tmp/setup.mysql" : {
    "content" : { "Fn::Join" : [ "", [
      "CREATE DATABASE ", { "Ref" : "DBName" }, ";\n",
      "CREATE USER '", { "Ref" : "DBUsername" }, "'@'localhost' IDENTIFIED BY
'",
      { "Ref" : "DBPassword" }, "';\n",
      "GRANT ALL ON ", { "Ref" : "DBName" }, ".* TO '", { "Ref" : "DBUsername"
},
      "'@'localhost';\n",
      "FLUSH PRIVILEGES;\n"
    ]]},
    "mode"  : "000644",
    "owner" : "root",
    "group" : "root"
  }
}
```

Additional file options, such as `symlinks` and `mustache` templates, are also supported.

COMMANDS

The commands key allows the execution of arbitrary commands on an Amazon EC2 instance, such as calling a custom application or script file.

Command Order of Execution

The commands section processes commands in alphabetical order based on the command name key. In this snippet, the command test would be called before test2.

```
"commands" : {
  "test" : {
    "command" : "echo \"$MAGIC\" > test.txt",
    "env" : { "MAGIC" : "I come from the environment!" },
    "cwd" : "~",
    "test" : "test ! -e ~/test.txt",
    "ignoreErrors" : "false"
  },
  "test2" : {
    "command" : "echo \"$MAGIC2\" > test2.txt",
    "env" : { "MAGIC2" : "I come from the environment!" },
    "cwd" : "~",
    "test" : "test ! -e ~/test2.txt",
    "ignoreErrors" : "false"
  }
}
```

SERVICES

The services key defines which services are enabled or disabled on the instance being configured. Linux systems utilize sysvinit to support the services key, while Windows systems use Windows Service Manager. Additionally, you can configure services to restart when dependencies update, such as files and packages. The following example enables nginx, configures it to run when the instance starts, and restarts whenever /var/www/html updates on the instance.

```
"services" : {
  "sysvinit" : {
    "nginx" : {
      "enabled" : "true",
      "ensureRunning" : "true",
      "files" : ["/etc/nginx/nginx.conf"],
      "sources" : ["/var/www/html"]
    }
  }
}
```

CONFIGSETS

You can organize config keys into *configSets*, which allow you to call groups of configurations at different times during an instance's setup process and change the order in which configurations are applied. The following example shows two configSets, which reverse the order in which configurations execute.

```
"AWS::CloudFormation::Init" : {
    "configSets" : {
        "ascending" : [ "config1" , "config2" ],
        "descending" : [ "config2" , "config1" ]
    },
    "config1" : {
        "commands" : {
            "test" : {
                "command" : "echo \"$CFNTEST\" > test.txt",
                "env" : { "CFNTEST" : "I come from config1." },
                "cwd" : "~"
            }
        }
    },
    "config2" : {
        "commands" : {
            "test" : {
                "command" : "echo \"$CFNTEST\" > test.txt",
                "env" : { "CFNTEST" : "I come from config2" },
                "cwd" : "~"
            }
        }
    }
}
```

ENFORCING AWS::CLOUDFORMATION::INIT METADATA

To enforce the AWS::CloudFormation::Init metadata, instances being provisioned in your template must call the cfn-init helper script as part of UserData execution (either in the AWS::EC2::Instance UserData property or in the same property of an AWS::AutoScaling::LaunchConfiguration resource). When doing so, you must provide the stack name and resource logical ID. Optionally, you can execute a configSet or list of configSets in the call.

You must pass UserData to instances in Base64 format. Thus, you call the Fn::Base64 function to convert the text-based script to a Base64 encoding.

```
"UserData" : { "Fn::Base64" :
  { "Fn::Join" : ["", [
     "#!/bin/bash -xe\n",
```

```
      "# Install the files and packages from the metadata\n",
      "/opt/aws/bin/cfn-init -v ",
      "         --stack ", { "Ref" : "AWS::StackName" },
      "         --resource WebServerInstance ",
      "         --configsets InstallAndRun ",
      "         --region ", { "Ref" : "AWS::Region" }, "\n"
  ]]}
}
```

AWS::CloudFormation::Interface

This section details how to modify the ordering and presentation of parameters in the AWS CloudFormation console. Without this section, parameters display alphabetically without any additional clarification. This is especially useful when providing templates to other groups who are not familiar with the purpose of each input parameter.

This `Metadata` section is only for the visual appearance of parameters in the AWS CloudFormation console. `metadata` does not have change templates that you submit via the AWS CLI or AWS SDK.

The `AWS::CloudFormation::Interface` metadata key uses two child keys, `ParameterGroups` and `ParameterLabels`.

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

PARAMETERGROUPS

You use the `ParameterGroups` section to organize sets of parameters into logical groupings, which are then separated by horizontal lines in the console. Each entry in `ParameterGroups` is defined as an object with a `Label` key and `Parameters` key. The `Label` key contains a friendly text name for each grouping of parameters. The `Parameters` key contains a list of logical IDs for each parameter in the group.

```
"ParameterGroups" : [
    {
        "Label" : { "default" : "Network Configuration" },
        "Parameters" : [ "VPCID", "SubnetId", "SecurityGroupID" ]
    },
    {
```

```
        "Label" : { "default":"Amazon EC2 Configuration" },
        "Parameters" : [ "InstanceType", "KeyName" ]
    }
]
```

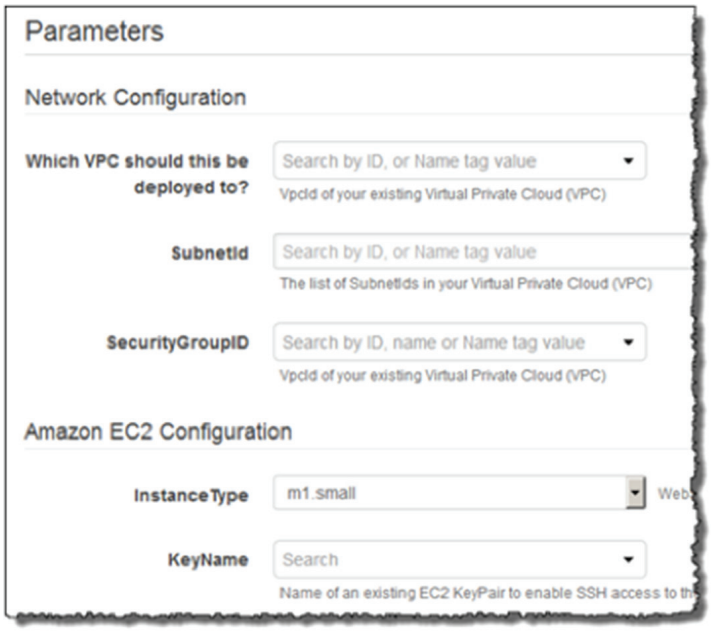
PARAMETERLABELS

The `ParameterLabels` section lets you define friendly names for parameters in the console. A logical ID such as `BastionSecurityGroupName` may be confusing to consumers of your template, especially if the template is shared outside your organization or team. By providing a more human-readable name, template portability is increased. The `ParameterLabels` key takes a list of parameter logical IDs, each of which has a friendly description as a subkey.

```
"ParameterLabels" : {
    "VPCID" : { "default" : "Which VPC should this be deployed to?" }
}
```

The inclusion of the `AWS::CloudFormation::Interface` definition results in an easy-to-understand list of parameters that you can complete, as shown in Figure 8.1.

FIGURE 8.1 AWS CloudFormation parameters



AWS::CloudFormation::Designer

This `Metadata` section specifies the visual layout and representation of resources when you design templates in the AWS CloudFormation Designer. Since it is used by Designer, we do not recommend that you manually modify this section.

AWS CloudFormation Designer

AWS CloudFormation Designer is a web-based graphical interface used to design and deploy AWS CloudFormation templates. You can design templates with a drag-and-drop interface of resource objects. You can create connections to make relationships between resources, which automatically update dependencies between them. When you are ready to deploy, you can submit the template directly to AWS CloudFormation or download it in JSON or YAML format.

AWS CloudFormation Designer keeps track of resource positions and relationships with metadata information in `AWS::CloudFormation::Designer`. Since no other service or component uses this information, it is safe to leave as is within your template.

Custom Resources

Sometimes custom provisioning logic is required when creating resources in AWS. Common examples of this include managing resources not currently supported by AWS CloudFormation, interacting with third-party tools, or other situations where more complexity is involved in the provisioning process.

AWS CloudFormation uses *custom resource providers* to handle the provisioning and configuration of custom resources. Custom resource providers may be AWS Lambda functions or Amazon Simple Notification Service (Amazon SNS) topics. When you create, update, or delete a custom resource, either the AWS Lambda function is invoked or a message is sent to the Amazon SNS topic you configure in the resource declaration.

In the custom resource declaration, you must provide a service token along with any optional input parameters. The *service token* acts as a reference to where custom resource requests are sent. This can be either an AWS Lambda function or Amazon SNS topic. Any input parameters you include are sent with the request body. After the resource provider processes the request, a `SUCCESS` or `FAILED` result is sent to the presigned Amazon S3 URL you included in the request body. AWS CloudFormation monitors this bucket location for a response, which it processes once it is sent by the provider. Custom resources can provide outputs back to AWS CloudFormation, which are made accessible as properties of the custom resource. You can access these properties with the `Fn::GetAtt` intrinsic function to pass the logical ID of the resource and the attribute you desire to query.

AWS Lambda Backed Custom Resources

Custom resources that are backed by AWS Lambda invoke functions whenever create, update, or delete actions are sent to the resource provider. This resource type is incredibly useful to reference other AWS services and resources that may not support AWS CloudFormation. Also, you can use them to look up data from other resources, such as Amazon EC2 instance IDs or entries in an Amazon DynamoDB table.

You can include the AWS Lambda function, which acts as a resource provider in the same AWS CloudFormation template that creates the custom resource and adds additional flexibility for stack update events. In this case, you can define the code for the

AWS Lambda function itself inline in the template or store it in a separate location such as Amazon S3. The following example demonstrates a custom resource, `AMIInfo`, which makes use of an AWS Lambda function, `AMIInfoFunction`, as the resource provider. Two additional properties, `Region` and `OSName`, provide inputs to the resource provider.

```
"AMIInfo": {
  "Type": "Custom::AMIInfo",
  "Properties": {
    "ServiceToken": { "Fn::GetAtt" : ["AMIInfoFunction", "Arn"] },
    "Region": { "Ref": "AWS::Region" },
    "OSName": { "Ref": "WindowsVersion" }
  }
}
```

For the AWS Lambda function to execute successfully, you must supply it with an IAM role. If the function will interact with other AWS services, you need the following permissions at minimum:

- `logs:CreateLogGroup`
- `logs:CreateLogStream`
- `logs:PutLogEvents`

Custom Resources Associated with Amazon SNS

Although AWS Lambda functions are incredibly powerful and versatile, they have a limit of 5 minutes of execution time, at which point the function will exit prematurely. This may not be desirable, especially when custom resources take a long time to provision or update. In these situations, use *custom resources associated with Amazon SNS*.

With this resource type, notifications are sent to an Amazon SNS topic any time the custom resource triggers. As the developer you are responsible for managing the system that receives notifications and performs processing. For instance, transcoding of long video files may take a longer time than AWS Lambda allows. In these situations, you subscribe an Amazon EC2 instance to the Amazon SNS topic to listen for requests, consume the input request object, perform the transcoding work, and place an appropriate response.

Custom Resource Success/Failure

For a custom resource to be successful in AWS CloudFormation, the resource provider must return a success response to the presigned Amazon S3 URL that you provide in the request. If you do not provide a response, the custom resource will eventually time out. This is especially important with regard to update and delete actions. The custom resource provider will need to respond appropriately to every action type (create, update, and delete) for both successful and unsuccessful attempts. If you do not provide a response to an update action, for example, the entire stack update will fail after the custom resource times out, and this results in a stack rollback.

Resource Relationships

By default, AWS CloudFormation will track most dependencies between resources. There are, however, some exceptions to this process. For example, an application server may not function properly until the backend database is up and running. In this case, you can add a `DependsOn` attribute to your template to specify the order of creation. The *`DependsOn`* attribute specifies that creation of a resource should not begin until another completes. A resource can have a dependency on one or more other resources in a stack. The following code demonstrates that the resource `EC2Instance` has a dependency on `MyDB`, which means the instance resource will not begin creation until the database resource is in a `CREATE_COMPLETE` state.

```
{
    "Resources" : {
        "Ec2Instance" : {
            "Type" : "AWS::EC2::Instance",
            "Properties" : {
                "ImageId" : {
                   "Fn::FindInMap" : [ "RegionMap", { "Ref" : "AWS::Region" }, "AMI" ]
                }
            },
            "DependsOn" : "myDB"
        },
        "myDB" : {
            "Type" : "AWS::RDS::DBInstance",
            "Properties" : {
               "AllocatedStorage" : "5",
               "DBInstanceClass" : "db.m1.small",
               "Engine" : "MySQL",
               "EngineVersion" : "5.5",
               "MasterUsername" : "MyName",
               "MasterUserPassword" : "MyPassword"
            }
        }
    }
}
```

Creation Policies

There may be situations where a dependency is not enough, such as when you install and configure applications on an instance before you attach it to an elastic load balancer. In this case, you can use a `CreationPolicy`. A *`CreationPolicy`* instructs AWS CloudFormation not to mark a resource as `CREATE_COMPLETE` until the resource itself signals back to the service.

You can configure the creation policy to require a specific number of signals in a certain amount of time; otherwise, the resource will show `CREATE_FAILED`. Signals sent to a resource are visible events in the AWS CloudFormation stack logs.

You can define creation policies with this syntax. When you configure creation policies for AWS Auto Scaling groups, you must specify the `MinSuccessfulInstancesPercent` property so that a certain percentage of instances in the group setup successfully complete before the group itself shows `CREATE_COMPLETE`. You can also configure creation policies to require a certain number of signals (`Count`) in a certain amount of time (`Timeout`). The following code example displays an AWS Auto Scaling group resource with a creation policy. This policy specifies that at least three signals must be received in 15 minutes for the group to create successfully.

```
"AutoScalingGroup": {
  "Type": "AWS::AutoScaling::AutoScalingGroup",
  "Properties": {
    "AvailabilityZones": { "Fn::GetAZs": "" },
    "LaunchConfigurationName": { "Ref": "LaunchConfig" },
    "DesiredCapacity": "3",
    "MinSize": "1",
    "MaxSize": "4"
  },
  "CreationPolicy": {
    "ResourceSignal": {
      "Count": "3",
      "Timeout": "PT15M"
    }
  }
}
```

Wait Conditions

You can use the *`WaitCondition`* property to insert arbitrary pauses until resources complete. If you require additional tracking of stack creation, you can use the `WaitCondition` property to add pauses to wait for external configuration tasks. An example of this would be if you create an Amazon DynamoDB table with a custom resource associated with AWS Lambda to load data into the table and then install software on an Amazon EC2 instance that reads data from the table. You can insert a `WaitCondition` into this template to prevent the creation of the instance until the custom resource function signals that data has been successfully loaded.

For Amazon EC2 instances and AWS Auto Scaling groups, we recommend that you use creation policies instead of wait conditions.

Wait conditions consist of two resources in a template, an `AWS::CloudFormation::WaitCondition` (wait condition) and an `AWS::CloudFormation::WaitConditionHandle` (wait condition handle).

The first resource, the wait condition, is similar to a creation policy. It requires a signal count and timeout value. However, it also requires a reference to a wait condition handle. The wait condition handle acts as a reference to a presigned URL where signals are sent to AWS CloudFormation, which it monitors.

Wait condition handles should never be reused between stack creation and subsequent updates, as it may result in signals from previous stack actions being evaluated. Instead, create new wait conditions for each stack action.

In the following example, the `WebServerGroup` resource creates an AWS Auto Scaling group with a count equal to the `WebServerCapacity` parameter. The example also creates a wait condition and wait condition handle, where the wait condition handle expects a number of signals equal to the `WebServerCapacity` parameter.

```
"WebServerGroup" : {
   "Type" : "AWS::AutoScaling::AutoScalingGroup",
   "Properties" : {
      "AvailabilityZones" : { "Fn::GetAZs" : "" },
      "LaunchConfigurationName" : { "Ref" : "LaunchConfig" },
      "MinSize" : "1",
      "MaxSize" : "5",
      "DesiredCapacity" : { "Ref" : "WebServerCapacity" },
      "LoadBalancerNames" : [ { "Ref" : "ElasticLoadBalancer" } ]
   }
},
"WaitHandle" : {
   "Type" : "AWS::CloudFormation::WaitConditionHandle"
},
"WaitCondition" : {
   "Type" : "AWS::CloudFormation::WaitCondition",
   "DependsOn" : "WebServerGroup",
   "Properties" : {
      "Handle"  : { "Ref" : "WaitHandle" },
      "Timeout" : "300",
      "Count"   : { "Ref" : "WebServerCapacity" }
   }
}
```

With this approach, you need to ensure that the signal is sent to the wait condition handle. This is done in the Amazon EC2 instance's user data, which you define in the launch configuration for AWS Auto Scaling groups. In this case, the `LaunchConfig` resource must include a signal to the wait condition handle. To do this, you reference the wait condition handle within the launch configuration's `UserData` script.

```
"UserData" : {
   "Fn::Base64" : {
       "Fn::Join" : [ "", ["SignalURL=", { "Ref" : "myWaitHandle" } ] ]
   }
}
```

Within `UserData`, you can use a `curl` command to send the success signal back to AWS CloudFormation.

```
curl -T /tmp/a "WAIT_CONDITION_HANDLE_URL"
```

The file `/tmp/a` must be in the following format:

```
{
   "Status" : "SUCCESS",
   "Reason" : "Configuration Complete",
   "UniqueId" : "ID1234",
   "Data" : "Application has completed configuration."
}
```

The `Data` section of the JSON response can include arbitrary data about the signal. You can make it accessible in the AWS CloudFormation template with the `Fn::GetAtt` intrinsic function.

```
"Outputs": {
    "WaitConditionData" : {
        "Value" : { "Fn::GetAtt" : [ "mywaitcondition", "Data" ]},
        "Description" : "The data passed back as part of signalling the
WaitCondition"
    }
}
```

Stack Create, Update, and Delete Statuses

Whenever you perform an action on an AWS CloudFormation stack, the end result will bring the stack into one of three possible statuses: Create, Update, and Delete. These statuses are visible in the AWS CloudFormation console, or if you use the `DescribeStacks` action.

CREATE_COMPLETE

The stack has created successfully.

CREATE_IN_PROGRESS

The stack is currently undergoing creation. No error has been detected.

CREATE_FAILED

One or more resources has failed to create successfully, causing the entire stack creation to fail. Review the stack failure messages to determine which resource(s) failed to create.

DELETE_COMPLETE

The stack has deleted successfully and will remain visible for 90 days.

DELETE_IN_PROGRESS

The stack is currently deleting.

DELETE_FAILED

The stack delete action has failed because of one or more underlying resources failing to delete. Review the stack output events to determine which resource(s) failed to delete. There you can manually delete the resource to prevent the stack delete from failing again.

ROLLBACK_COMPLETE

If a stack creation action fails to complete, AWS CloudFormation will automatically attempt to roll the stack back and delete any created resources. This status is achieved when the resources have been removed.

ROLLBACK_IN_PROGRESS

The stack has failed to create and is currently rolling back.

ROLLBACK_FAILED

If AWS CloudFormation is not able to delete resources that were provisioned during a failed stack create action, the stack will enter `ROLLBACK_FAILED`. The remaining resources will not be deleted until the error condition is corrected. Other than attempting to continue deleting the stack, no other actions can be performed on the stack itself. To resolve this, review the stack events to determine which resource(s) failed to delete.

UPDATE_COMPLETE

The stack has updated successfully.

UPDATE_IN_PROGRESS

The stack is currently performing an update.

UPDATE_COMPLETE_CLEANUP_IN_PROGRESS

When AWS CloudFormation updates certain resources, the type of update may require a replacement of the original physical resource. In these situations, AWS CloudFormation will first create the replacement resource and verify that the provision was successful. After all resources update, the stack will enter this phase and remove any previous resources. For example, when you update

the `Name` property of an `AWS::S3::Bucket` resource, AWS CloudFormation will create a bucket with the new name value and then delete the previous bucket during the cleanup phase.

UPDATE_ROLLBACK_COMPLETE

If a stack update fails, AWS CloudFormation will attempt to roll the stack back to the last working state. Once complete, the stack will enter the `UPDATE_ROLLBACK_COMPLETE` state.

UPDATE_ROLLBACK_IN_PROGRESS

After a stack update fails, AWS CloudFormation begins to roll back any changes to bring the stack back to the last working state.

UPDATE_ROLLBACK_COMPLETE_CLEANUP_IN_PROGRESS

A failed rollback will require a cleanup of any newly created resources that would have originally replaced existing ones. During this phase, replacement resources are deleted in place of the originals.

UPDATE_ROLLBACK_FAILED

If the stack update fails and the rollback is unable to return it to a working state, it will enter `UPDATE_ROLLBACK_FAILED`. You can delete the entire stack. Otherwise, you can review to determine what failed to roll back and continue the update rollback again.

Stack Updates

You do not need to re-create stacks any time you need to update an underlying resource. You can modify and resubmit the same template, and AWS CloudFormation will parse it for changes and apply the modifications to the resources. This can include the ability to add new resources or modify and delete existing ones. You can perform stack updates when you create a new template or parameters directly, or you can create a change set with the updates.

Some template sections, such as `Metadata`, require you to modify one or more resources when the stack updates, as you cannot change them on their own. You can change parameters without modifying the stack's template.

When performing a stack action, such as an update, one or more stack events are created. The event contains information such as the resource being modified, the action being performed, and resource IDs. One critical piece of information in the stack event is the `ClientRequestToken`.

All events triggered by a single stack action are assigned the same token value. For example, if a stack update modifies an Amazon S3 bucket and Amazon EC2 instance, the corresponding Amazon S3 and Amazon EC2 API calls will contain the same request token. This lets you easily track what API activity corresponds to particular stack actions. This API activity can be tracked in AWS CloudTrail and stored in Amazon S3 for later review.

When you update a stack, underlying resources can exhibit one of several behaviors. This depends on the update to the resource property or properties. Resource property changes can cause one of update types to occur, as shown in Table 8.1.

TABLE 8.1 AWS CloudFormation Update Types

Update Type	Resource Downtime	Resource Replacement
Update with No Interruption	No	No
Update with Some Interruption	Yes	No
Replacing Update	Yes	Yes

For resource properties that require replacement, the resource's physical ID will change.

Some resource properties do not support updates. In these cases, you must create new resources first. After this, you can remove the original resource from the stack.

Update Policies

You use the AWS CloudFormation *`UpdatePolicy`* to determine how to respond to changes to `AWS::AutoScaling::AutoScalingGroup` and `AWS::Lambda::Alias` resources.

For AWS Auto Scaling group update policies, there are policies that you can enforce. These depend on the type of change you make and whether you configure the AWS Auto Scaling scheduled actions. Table 8.2 displays the types of policies that take effect under each scenario.

TABLE 8.2 AWS Auto Scaling Update Types in AWS CloudFormation

AWS Auto Scaling Update Type	Change AWS Auto Scaling group Launch Configuration	Change AWS Auto Scaling group `VPCZoneIdentifier` Property	AWS Auto Scaling group Has a Scheduled Action
`AutoScalingReplacingUpdate`	X	X	
`AutoScalingRollingUpdate`	X	X	
`AutoScalingScheduledAction`			X

You can configure the `WillReplace` property for an `UpdatePolicy` to `true` and give precedence to the `AutoScalingReplacingUpdate` settings.

The `AutoScalingReplacingUpdate` policy defines how to replace updates. You can replace the entire AWS Auto Scaling group or only instances inside.

```
"UpdatePolicy" : {
  "AutoScalingReplacingUpdate" : {
    "WillReplace" : Boolean
  }
}
```

The `AutoScalingRollingUpdate` policy allows you to define the update process for instances in an AWS Auto Scaling group. This lets you configure the group to update instances all at once, in batches, or with an additional batch.

SuspendProcesses Attribute

The `SuspendProcesses` attribute can define whether to suspend AWS Auto Scaling scheduled actions or those you invoke by alarms, which can otherwise cause the update to fail.

```
"UpdatePolicy" : {
  "AutoScalingRollingUpdate" : {
    "MaxBatchSize" : Integer,
    "MinInstancesInService" : Integer,
    "MinSuccessfulInstancesPercent" : Integer
    "PauseTime" : String,
    "SuspendProcesses" : [ List of processes ],
    "WaitOnResourceSignals" : Boolean
  }
}
```

Lastly, for AWS Auto Scaling groups, the `AutoScalingScheduledAction` property defines whether to adhere to the group sizes (minimum, maximum, and desired counts) you define in your template. If your AWS Auto Scaling group has enabled scheduled actions, there is a possibility that the actual group sizes no longer reflect those in the template. If you run an update without this policy set, it can cause the group to be reverted to its original size. If you configure the `IgnoreUnmodifiedGroupSizeProperties` property to `true`, it will cause

AWS CloudFormation to ignore different group sizes when it compares the template to the actual AWS Auto Scaling group.

```
"UpdatePolicy" : {
  "AutoScalingScheduledAction" : {
    "IgnoreUnmodifiedGroupSizeProperties" : Boolean
  }
}
```

For changes to an `AWS::Lambda::Alias` resource, you can define the `CodeDeployLambdaAliasUpdate` policy. This controls whether a deployment is made with AWS CodeDeploy whenever it detects version changes.

```
"UpdatePolicy" : {
  "CodeDeployLambdaAliasUpdate" : {
    "AfterAllowTrafficHook" : String,
    "ApplicationName" : String,
    "BeforeAllowTrafficHook" : String,
    "DeploymentGroupName" : String
  }
}
```

In the previous examples, you only require the `ApplicationName` and `DeploymentGroupName` properties. These refer to the AWS CodeDeploy application and deployment group, which should update when the alias changes.

Deletion Policies

When you delete a stack, by default all underlying stack resources are also deleted. If this behavior is not desirable, apply the *`DeletionPolicy`* to resources in the stack to modify their behavior when the stack is deleted. You use deletion policies to preserve resources when you delete a stack (set `DeletionPolicy` to `Retain`). Some resources can instead have a snapshot or backup taken before you delete the resource (set `DeletionPolicy` to `Snapshot`). The following resource types support snapshots:

- `AWS::EC2::Volume`
- `AWS::ElastiCache::CacheCluster`
- `AWS::ElastiCache::ReplicationGroup`
- `AWS::RDS::DBInstance`
- `AWS::RDS::DBCluster`
- `AWS::Redshift::Cluster`

The following template example creates an Amazon S3 bucket with a deletion policy set to retain the bucket when you delete the stack.

```
{
  "AWSTemplateFormatVersion" : "2010-09-09",
  "Resources" : {
    "myS3Bucket" : {
      "Type" : "AWS::S3::Bucket",
      "DeletionPolicy" : "Retain"
    }
  }
}
```

Exports and Nested Stacks

Since AWS CloudFormation enforces limits on how large templates can grow and how many resources, outputs, and parameters you can declare in one template, situations can arise where you will need to manage more infrastructure than a single stack will allow. There are two approaches to manage relationships between multiple stacks. You use *stack exports* to share information between separate stacks or manage AWS CloudFormation stacks themselves as resources in a "parent" or "master" stack (a *nested stack* relationship).

Export and Import Stack Outputs

You can export stack output values to import them into other stacks in the same account and region. This allows you to share data that generates in one stack out to other stacks in your account. If, for example, you create a networking infrastructure such as an Amazon VPC in one stack, you can export the IDs of such resources from this stack and import them into others at a later date.

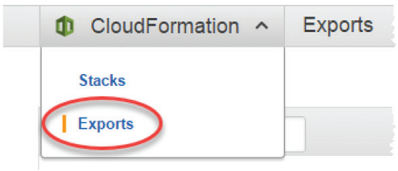
To export a stack value, update the `Outputs` section to include an `Export` declaration for every output you want to share.

```
"Outputs" : {
  "Logical ID" : {
    "Description" : "Information about the value",
    "Value" : "Value to return",
    "Export" : {
      "Name" : "Value to export"
    }
  }
}
```

Export values must have a unique name within the AWS account and AWS region.

After you declare the export and the stack creates or updates, it displays in the AWS CloudFormation console on the Exports tab, as shown in Figure 8.2.

FIGURE 8.2 AWS CloudFormation Exports tab



To import this value into another stack, you use the *Fn::ImportValue* intrinsic function. This intrinsic function requires only the export name as an input parameter (the name present in the AWS CloudFormation console).

You cannot change export values after you import them into another stack. You must first modify the import stack so that it no longer uses the export. To list stacks that import an exported output, use the `ListImports` API action.

```
https://cloudformation.us-east-1.amazonaws.com/
 ?Action=ListImports
 &ExportName=SampleStack-MyExportedValue
 &Version=2010-05-15
 &X-Amz-Algorithm=AWS4-HMAC-SHA256
 &X-Amz-Credential=[Access key ID and scope]
 &X-Amz-Date=20160316T233349Z
 &X-Amz-SignedHeaders=content-type;host
 &X-Amz-Signature=[Signature]
```

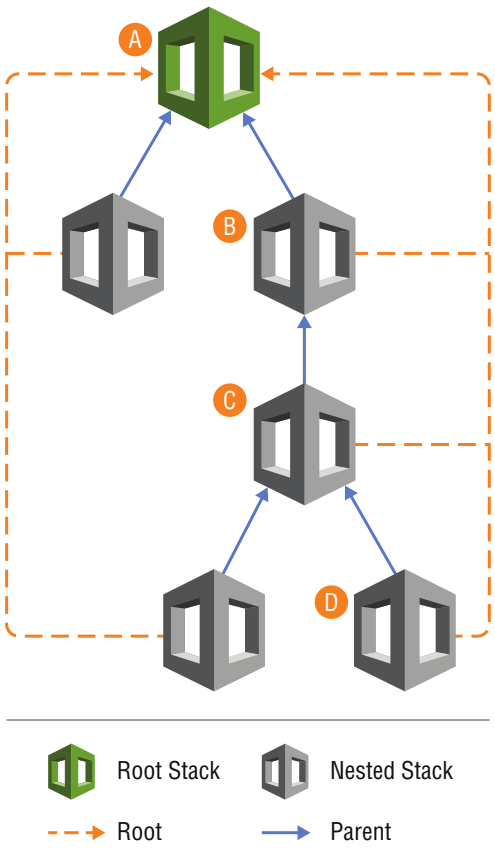
Nesting with the AWS::CloudFormation::Stack Resource

You can manage stacks as resources within the service in AWS CloudFormation. A single parent stack can create one or more `AWS::CloudFormation::Stack` resources, which act as child stacks that the parent manages. The direct benefits of this are as follows:

- You can work around template limits that AWS CloudFormation imposes.
- It provides the ability to separate resources into logical groups, such as network, database, and web application.
- It lets you separate duties. (Each team is responsible only for maintaining their respective child stack.)

You can increase the nesting levels, as shown in Figure 8.3, with the `AWS::CloudFormation::Stack` resources.

FIGURE 8.3 Nested stack structure



From a workflow perspective, the "topmost" parent stack should manage all updates to child stacks. In Figure 8.3, if you need to update stack D, you perform the update on stack A, the topmost parent, to accomplish this.

You can share data from each nested stack if you use a combination of stack outputs and the `Fn::GetAtt` function calls. If there is an output value from a nested stack that you would like to access from its parent, the following syntax will let you access stack outputs.

```
{ "Fn::GetAtt" : [ "logicalNameOfChildStack", "Outputs.attributeName" ] }
```

Outputs from stacks created by a nested stack (such as to access outputs in stack C from stack A, as shown in Figure 8.3) can be accessed from the parent stack(s). First, you will need to output the value in the originating stack and then its parent and finally access the output from the parent. To clarify, the output would originate in stack C and be added as an output to stack B, and then stack A references it.

Stack Policies

Though you can assign resources to create, update, and delete policies to stacks directly, there may be situations where you will want to prevent certain types of updates to stacks themselves. By default, anyone with permissions to modify stacks can perform updates to all underlying stack resources (if they have permissions to modify the resources themselves, or the AWS CloudFormation service role attached to the stack has these permissions). You can assign a *stack policy* to a stack to allow or deny access to modify certain stack resources, which you can filter by the type of update. Stack policies apply to all users, regardless of their IAM permissions.

```
{
  "Statement" : [
    {
      "Effect" : "Allow",
      "Action" : "Update:*",
      "Principal": "*",
      "Resource" : "*"
    },
    {
      "Effect" : "Deny",
      "Action" : "Update:*",
      "Principal": "*",
      "Resource" : "LogicalResourceId/ProductionDatabase"
    }
  ]
}
```

Stack policies are not a replacement for appropriate access control from an IAM policy. Stack policies are an additional fail-safe to prevent accidental updates to critical resources.

Stack policies protect all resources by default with an implicit deny. To allow access to actions on stack resources, you must apply explicit allow statements to the policy. In the previous example, an explicit allow specifies that you can perform all updates on all resources in the stack. However, the explicit deny for the `ProductionDatabase` resource prevents update actions to this specific resource. You can specify allow and deny actions for

either resource logical IDs or generic resource types. To specify policies for generic resource types, use a condition statement as follows:

```
{
  "Statement" : [
    {
      "Effect" : "Deny",
      "Principal" : "*",
      "Action" : "Update:*",
      "Resource" : "*",
      "Condition" : {
        "StringEquals" : {
          "ResourceType" : ["AWS::EC2::Instance", "AWS::RDS::DBInstance"]
        }
      }
    }
  ]
}
```

Once you apply a stack policy, you cannot remove it. During future updates, the policy must be temporarily replaced.

You can allow or deny specific types of updates for resources in your stack. Action types include the following:

`Update:Modify` Update actions where resources will experience some or no interruption

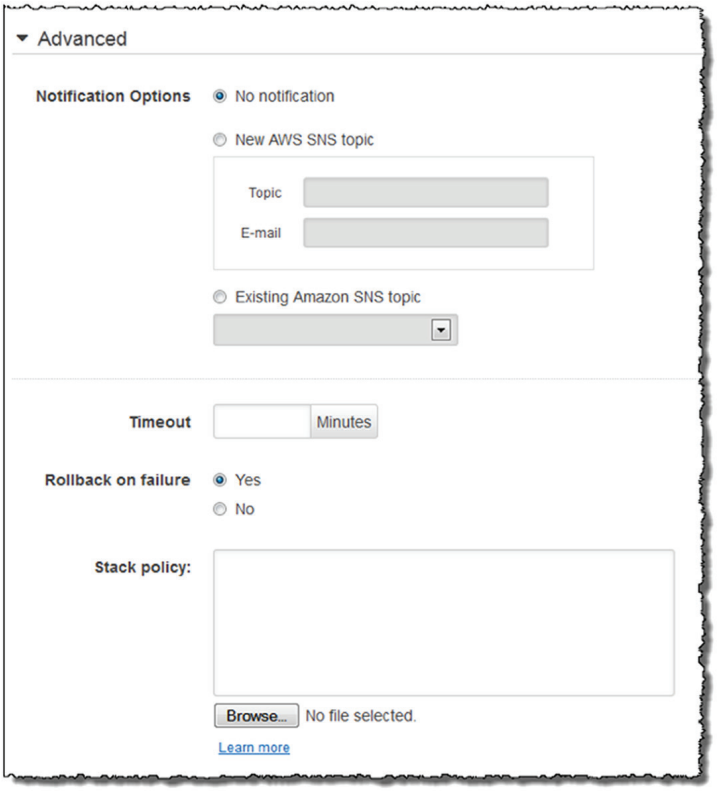
`Update:Replace` Update actions where replacement resources create (the physical ID of the resource changes)

`Update:Delete` Update actions where resources delete from the stack

`Update:*` All update actions

Once a stack policy has been set, it will need to be overridden during updates to protected resources. To do so, you supply a new, temporary stack policy. You add this stack policy in the console under the `Stack policy` property, as shown in Figure 8.4.

FIGURE 8.4 AWS CloudFormation Stack Policy field



When you supply a stack policy during an update, it only modifies the policy for the duration of the update after which the original policy reinstates.

AWS CloudFormation Command Line Interface

AWS CloudFormation provides several utility functions apart from the standard API-based component of the AWS CloudFormation CLI.

Packaging Local Dependencies

When you develop templates locally, you may require additional files for your infrastructure that you do not want to define inline as part of the template syntax. For example, you may need to place configuration files on Amazon EC2 instances or AWS Lambda function code. You can use the `aws cloudformation package` command to upload local files and convert local references in your template to Amazon S3 URIs. Consider the following example:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
```

```
Resources:
  MyFunction:
    Type: 'AWS::Serverless::Function'
    Properties:
      Handler: index.handler
      Runtime: nodejs4.3
      CodeUri: /home/user/code/lambdafunction
```

The `CodeUri` property refers to a local path on the user's workstation (`/home/user/code/lambdafunction`). To prepare this for deployment, you can run the following command:

```
aws cloudformation package --template /path_to_template/template.json --s3-bucket
mybucket --output json > packaged-template.json
```

When you execute this command, the AWS CLI will package the contents of `/home/user/code/lambdafunction` into a `.zip` archive and upload it to the Amazon S3 bucket you specify in the `--s3-bucket` parameter. After doing so, the template updates to refer to the Amazon S3 URI for the archive file and generates the following:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Resources:
  MyFunction:
    Type: 'AWS::Serverless::Function'
    Properties:
      Handler: index.handler
      Runtime: nodejs4.3
      CodeUri: s3://mybucket/lambdafunction.zip
```

Deploy Templates with Transforms

Any time that you want to deploy an AWS CloudFormation template that contains transforms, you must first create a change set. The *change set* is responsible for executing the transform to generate a final template that you can deploy. If you would like to reduce this to a one-step process, the `aws cloudformation deploy` command will generate and execute the change set on your behalf. This is especially useful for rapid testing, as it eliminates the need to approve change sets manually.

When you use this command, you can override default parameters with the `--parameter-overrides` property.

```
aws cloudformation deploy --template /path_to_template/my-template.json --stack-
name my-new-stack --parameter-overrides Key1=Value1 Key2=Value2
```

AWS CloudFormation Helper Scripts

When you execute custom scripts on Amazon EC2 instances as part of your `UserData`, AWS CloudFormation provides several important helper scripts. You can use these to interact with the stack to query metadata, notify a `CreationPolicy` or `WaitCondition`, and process scripts when AWS CloudFormation detects metadata updates.

cfn-init

You use this helper script to read `AWS::CloudFormation::Init` metadata from the `AWS::EC2::LaunchConfiguration` or `AWS::EC2::Instance` resource being declared. It is responsible for installing packages, adding files, creating users and groups, and any other configuration you specify in your `AWS::CloudFormation::Init` metadata.

`AWS::CloudFormation::Init` metadata is not enforced automatically. You must call the `cfn-init` helper script in your instances' `UserData`. The following example demonstrates a `cfn-init` call on an instance in an AWS CloudFormation stack. In this case, the `InstallAndRun` configuration set executes on the instance.

```
"UserData" : { "Fn::Base64" :
  { "Fn::Join" : ["", [
     "#!/bin/bash -xe\n",
     "# Install the files and packages from the metadata\n",
     "/opt/aws/bin/cfn-init -v ",
     "         --stack ", { "Ref" : "AWS::StackName" },
     "         --resource WebServerInstance ",
     "         --configsets InstallAndRun ",
     "         --region ", { "Ref" : "AWS::Region" }, "\n"
  ]]}
}
```

cfn-signal

After `cfn-init` has been called and the `AWS::CloudFormation::Init` metadata has been enforced successfully (or unsuccessfully), you can use *`cfn-signal`* to notify AWS CloudFormation that the instance has completed its configuration. For example, if your template contains a `CreationPolicy` or `WaitCondition` to prevent the setup of an `AWS::ElasticLoadBalancing::LoadBalancer` resource until instances in your `AWS::AutoScaling::AutoScalingGroup` have configured a custom application, `cfn-signal` performs the notification. The following `UserData` example demonstrates how to pass the result of `cfn-init` to `cfn-signal`:

```
"UserData": {
  "Fn::Base64": {
    "Fn::Join": [
      "",
      [
        "#!/bin/bash -x\n",
        "# Install the files and packages from the metadata\n",
```

```
        "/opt/aws/bin/cfn-init -v ",
        "         --stack ", { "Ref": "AWS::StackName" },
        "         --resource MyInstance ",
        "         --region ", { "Ref": "AWS::Region" },
        "\n",
        "# Signal the status from cfn-init\n",
        "/opt/aws/bin/cfn-signal -e $? ",
        "         --stack ", { "Ref": "AWS::StackName" },
        "         --resource MyInstance ",
        "         --region ", { "Ref": "AWS::Region" },
        "\n"
      ]
    ]
  }
}
```

cfn-get-metadata

If your template contains arbitrary metadata, use `cfn-get-metadata` to fetch this information for use on your instance(s). You can use this helper script to query either an entire metadata block or a subtree. AWS CloudFormation supports only top-level keys.

cfn-hup

Since AWS CloudFormation executes `UserData` only on resource creation, instances will not detect changes to `AWS::CloudFormation::Init` metadata automatically. Unlike other helper scripts, you can configure *`cfn-hup`* to run as a daemon on instances. This script checks for changes to resource metadata, can execute custom scripts whenever they are detected, and allows you to perform configuration updates on instances in a stack.

The `cfn-hup` helper script requires you to perform several configuration steps before it detects updates.

DAEMON CONFIGURATION FILE

You must create the `cfn-hup.conf` configuration file on the instance, and it needs to contain the stack name. You can also use `cfn-hup.conf` to contain AWS credentials the daemon requires, though it can also leverage IAM instance profiles. Here's an example:

```
[main]
stack=<stack-name-or-id>
```

HOOKS CONFIGURATION FILE

Whenever AWS CloudFormation detects changes to instance metadata, user-defined actions are called based on settings in the `hooks.conf` configuration file. You can configure hooks to run on one or more resource actions (add, update, or remove) and can execute arbitrary commands. If there are scripts you want to call, you must add the scripts to the instance

before you execute the hook. If you require more than one configuration file, you can add `/etc/cfn/hooks.d/` on Linux instances. The `hooks.conf` file structure is as follows:

```
[hookname]
triggers=post.add or post.update or post.remove
path=Resources.<logicalResourceId> (.Metadata or
.PhysicalResourceId)(.<optionalMetadatapath>)
action=<arbitrary shell command>
runas=<runas user>
```

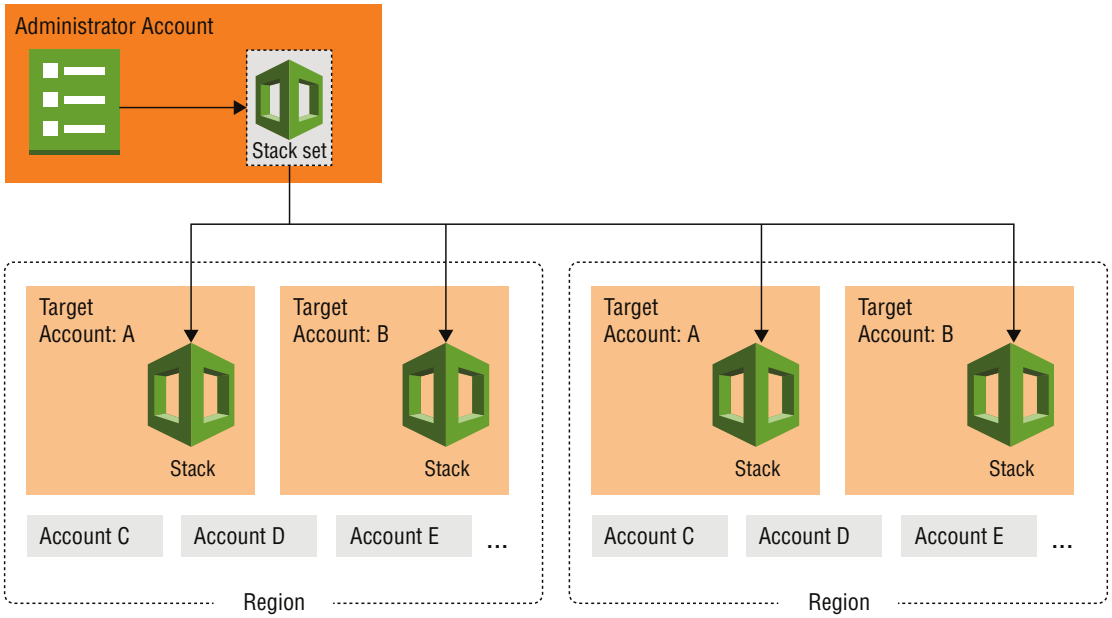
This template snippet demonstrates how to add a `cfn-hup` hook file to instances in an `AWS::AutoScaling::LaunchConfiguration` resource. This hook file will detect updates to the `LaunchConfig` resource and execute the `wordpress_install` config set you specify in the `AWS::CloudFormation::Init` metadata.

```
[hookname]
triggers=post.add or post.update or post.remove
path=Resources.<logicalResourceId> (.Metadata or
.PhysicalResourceId)(.<optionalMetadatapath>)
action=<arbitrary shell command>
runas=<runas user>
"LaunchConfig": {
  "Type" : "AWS::AutoScaling::LaunchConfiguration",
  "Metadata" : {
    "AWS::CloudFormation::Init" : {
      ...
      "/etc/cfn/hooks.d/cfn-auto-reloader.conf": {
        "content": { "Fn::Join": [ "", [
          "[cfn-auto-reloader-hook]\n",
            "triggers=post.update\n",
            "path=Resources.LaunchConfig.Metadata.AWS::CloudFormation::Init\n",
            "action=/opt/aws/bin/cfn-init -v ",
            "         --stack ", { "Ref" : "AWS::StackName" },
            "         --resource LaunchConfig ",
            "         --configsets wordpress_install ",
            "         --region ", { "Ref" : "AWS::Region" }, "\n",
            "runas=root\n"
          ]]},
          "mode"  : "000400",
          "owner" : "root",
          "group" : "root"
}
```

AWS CloudFormation StackSets

AWS CloudFormation *StackSets* gives users the ability to control, provision, and manage stacks across multiple accounts, as shown in Figure 8.5. From a centralized administrator account, you can develop a template as the basis for provisioning similar stacks across a fleet of accounts.

FIGURE 8.5 AWS CloudFormation StackSets structure



Stack Set

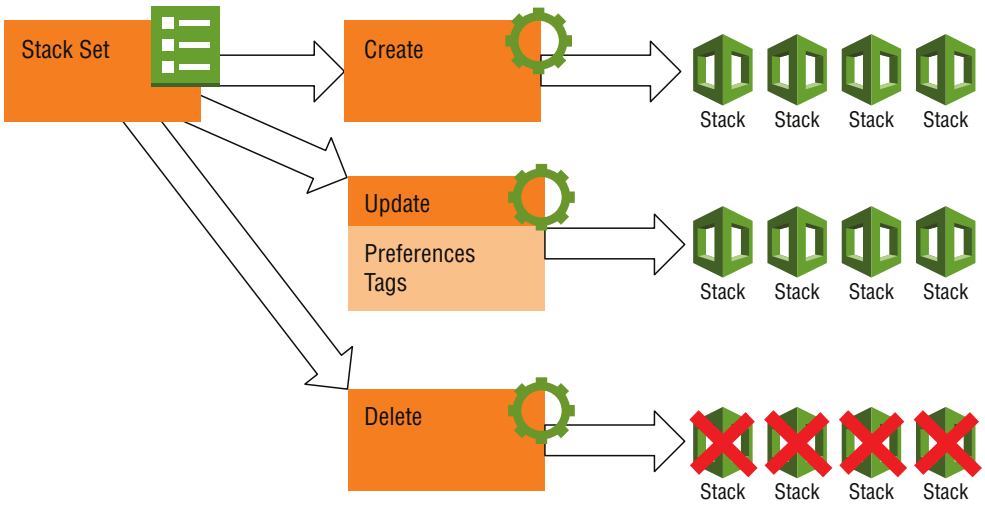
A *stack set* acts as a logical container for stack information in an administrator account. Each stack set will contain information about the stacks you deploy to a single target account in one or more regions. You can configure stack sets to deploy to regions in a specific order and how many unsuccessful deployments are required to fail the entire deployment.

Though a stack set allows you to deploy stacks to multiple regions, the stack set itself exists in one region, and you must manage it there.

Stack Instance

Stack instances allow you to manage stacks in a target account, as shown in Figure 8.6. For example, if a stack set deploys to four regions in a target account, you create four stack instances. An update to a stack set propagates to all stack instances in all accounts and regions.

FIGURE 8.6 AWS CloudFormation StackSet actions



Stack Set Operations

When you perform operations on stack sets, you can configure how to control the flow of updates across accounts and regions. You can specify a maximum number or percentage of target accounts for concurrent deployment. Additionally, you can specify a maximum number of percentage of failures (per region). Lastly, you can configure delete operations to remove only the stack instances and stack set and leave the stack itself present in the target account. This option is useful when removing control from an administrator account for resources that need to remain operational in the target account.

If you specify a maximum number of failures per region, stack updates will not progress to the next region when this threshold is breached. The stack set operation will stop completely.

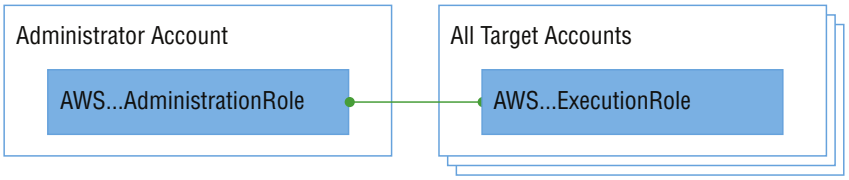
Stack Set Permissions

For an administrator account to deploy to any target accounts, you must create a trust relationship between the accounts. To do this, you create an IAM role in each account.

The administrator account requires an IAM service role with permissions to execute stack set operations and assume an execution role in any target accounts. This service role must have a trust policy that allows `cloudformation.amazonaws.com`.

Any target accounts will require an execution role that you create in the administrator account, which the service role can assume. This execution role will require AWS CloudFormation permissions and permissions to manage any resources you define in the template being deployed by the stack set, as shown in Figure 8.7.

FIGURE 8.7 AWS CloudFormation StackSets permissions



Target Account Gate

Before you create or update a stack set, evaluate potential blockers in target accounts. If a certain resource type is not available in different regions, for example, this can cause the stack set operation to fail. You can use a *target account gate* to perform evaluation tasks with AWS Lambda functions in the target account. Depending on the return value of the function, the stack set operation will either continue or stop. You can configure this so that account gate failures count toward the stack set's configured tolerance settings.

AWS CloudFormation Service Limits

Important service limits for AWS CloudFormation are listed in Table 8.3. *You cannot raise template-specific limits through a support request*. You can raise some limits such as the number of stacks per account.

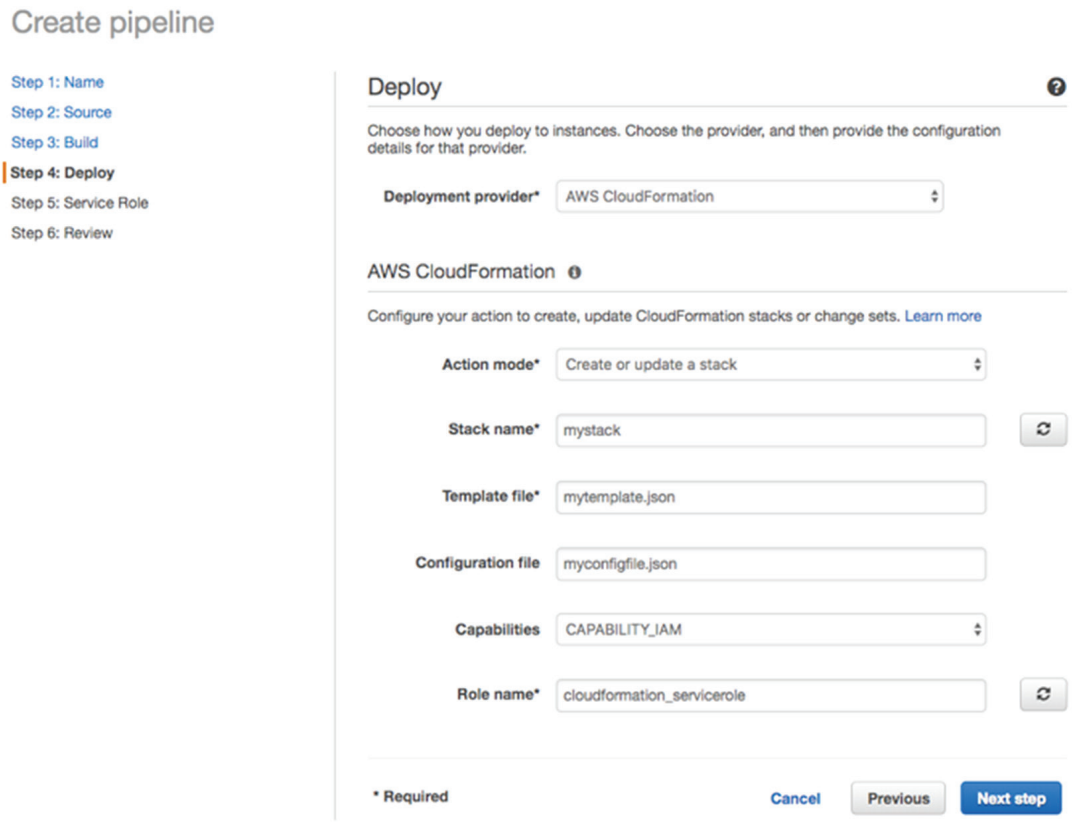
TABLE 8.3 AWS CloudFormation Service Limits

Limit	Value
Mappings per template	100
Outputs per template	60
Parameters per template	60
Resources per template	200
Stacks per account	200
Template body size	51,200 B (local file) 460,800 B (S3)

Using AWS CloudFormation with AWS CodePipeline

AWS CloudFormation has built-in integrations with AWS CodePipeline as a deployment provider. Refer to Figure 8.8. When a template revision passes through a pipeline, AWS CloudFormation can reference input parameters, stack policies, and other configuration data in the AWS CodePipeline deployment.

FIGURE 8.8 CloudFormation as a deployment provider



Deployment Configuration Properties

This section details deployment configuration properties including the following: Action Mode, Stack or Change Set Name, Templates, Template Configurations, Capabilities, Role Names, Output File Names, and Parameter Overrides.

Action Mode

You can use change sets in a pipeline to include a manual review step to ensure that the changes you deploy are valid and desired before they actually execute. AWS CodePipeline supports the following AWS CloudFormation actions:

- Create or replace a change set
- Create or update a stack
- Delete a stack
- Execute a change set
- Replace a failed stack

Stack or Change Set Name

These refer to the new or existing stack or change set to be created, updated, or deleted.

Template

This is the location of the template file to submit. Since AWS CodePipeline uses artifacts to pass files between stages, you must define this file within the artifact with the following:

```
ArtifactName::TemplateFileName
```

Template Configuration

The template configuration is where you specify properties such as template parameters and the stack policy.

Do not commit sensitive information to your repository. If this file contains information such as passwords, restrict access and pull it into the artifact from another source, such as Amazon S3.

Capabilities

You must specify any templates which create, update, or delete IAM resources with either the `CAPABILITY_IAM` or `CAPABILITY_NAMED_IAM` within this property.

Role Name

Unlike manually provisioned stacks, AWS CodePipeline requires a service role to assume when you perform actions in AWS CloudFormation.

Output File Name

This is an optional output that you can add to the output artifact after the deploy action completes. This will add any stack outputs to the pipeline output artifact.

Parameter Overrides

Though you can define parameters in the template configuration file, the parameter overrides section lets you specify a JSON input file to override any already-specified parameters. You can retrieve data from pipeline artifacts with the `Fn::GetParam` intrinsic function. The following example demonstrates how to specify a parameter override for `ParameterName`.

```
{
  "ParameterName" : {
    "Fn::GetParam" : ["ArtifactName", "config-file-name.json", "ParamName"]
  }
}
```

All parameters you specify in the parameter overrides or template configuration file must already exist in the Parameters section of the template you want to deploy.

Parameter overrides can leverage two intrinsic functions specific to AWS CodePipeline. These functions allow you to specify dynamic pipeline values and data from artifacts being passed through the pipeline.

FN::GETARTIFACTATT

You use *Fn::GetArtifactAtt* to query values of an input artifact attribute, such as the Amazon S3 bucket name where the artifact is stored. This function enables you to gather information about the artifact itself, not data within the artifact.

When you run a pipeline, AWS CodePipeline copies and writes files to the pipeline's artifact store (Amazon S3 bucket). AWS CodePipeline generates the filenames in the artifact store. These filenames are unknown before you run the pipeline. This attribute requires the Amazon S3 bucket name (`BucketName`), artifact object key (`ObjectKey`), and artifact URL (`URL`).

Use the following syntax to retrieve an attribute value of an artifact:

```
{ "Fn::GetArtifactAtt" : [ "artifactName", "attributeName" ] }
```

FN::GETPARAM

Complimentary to `Fn::GetArtifactAtt`, the *Fn::GetParam* function allows you to query information within an artifact. Any files in the artifact that you query must be in valid JSON format. For example, you can add outputs from a stack as a JSON file to the pipeline artifact, which you use `Fn::GetParam` to query.

```
{ "Fn::GetParam" : [ "artifactName", "JSONFileName", "keyName" ] }
```

Summary

In this chapter, you became familiar with provisioning and managing AWS infrastructure using AWS CloudFormation. AWS CloudFormation allows you to describe an entire enterprise's infrastructure as one or more template files, achieving infrastructure as code (IaC) in an environment.

By leveraging AWS CloudFormation in a deployment pipeline, you can dynamically provision and update infrastructure over time by simply committing code to a Git-based repository (AWS CodeCommit). *You can use AWS CodePipeline to reliably automate complex deployment processes*.

AWS CloudFormation uses a declarative language (JSON or YAML template) to describe, model, and provision all infrastructure resources for your applications across all regions and accounts in your cloud environment in an automated and secure manner. This file serves as the single source of truth for your cloud environment. You pay only for the AWS resources you require to run your applications.

The template contains the infrastructure to where AWS will deploy and configuration properties. After you deploy a template in an account, you can redeploy it again in the same or different account and/or region.

A stack is a collection of resources that will be deployed and managed by AWS CloudFormation. When you submit a template, the resources you configure are provisioned and then make up the stack itself. Any modifications to the stack affect underlying resources. Stacks use the IAM user or AWS role authorizations to invoke an action. The template only requires the `Resources` section.

When you create a stack, you can submit a template from a local file or via a URL that points to an object in Amazon S3. If you submit the template as a local file, it uploads to Amazon S3 on your behalf.

Two key benefits of AWS CloudFormation are that your infrastructure is repeatable and that it is versionable.

A change set is a description of the changes that will occur to a stack should you submit the template and/or parameter updates. When you process stack updates, the template snippets you reference in any transforms pull from their Amazon S3 locations. If a snippet updates without your knowledge, the updated snippet will import into the template. Use a change set where there is a potential for data loss.

If values input into a template cannot be determined until the stack or change set is actually created, intrinsic functions resolve this by adding dynamic functionality into AWS CloudFormation templates. Condition functions are intrinsic functions to create resources or set resource properties that evaluate true or false conditions.

AWS CloudFormation Designer is a web-based graphical interface to design and deploy AWS CloudFormation templates. You can create connections to make relationships between resources that automatically update dependencies between them.

AWS CloudFormation uses custom resource providers to handle the provisioning and configuration of custom resources with AWS Lambda functions or Amazon SNS topics. You must provide a service token along with any optional input parameters. *The service token acts as a reference to where custom resource requests are sent*. This can be an AWS Lambda function or Amazon SNS topic. Custom resources can provide outputs back to AWS CloudFormation, which are made accessible as properties of the custom resource.

Custom resources associated with AWS Lambda invoke functions whenever create, update, or delete actions are sent to the resource provider. This resource type is incredibly useful to reference other AWS services and resources that may not support AWS CloudFormation.

You can use custom resources associated with Amazon SNS for any long-running custom resource tasks, such as transcoding a large video file.

By default, AWS CloudFormation will track most dependencies between resources. *A resource can have a dependency on one or more other resources in a stack, in which case you create a resource relationship to control the order of resource creation, updates, and deletion.*

Whenever you perform an action on an AWS CloudFormation stack, the end result will bring the stack into one of several possible statuses. These actions can complete or fail. In the case of a failed event, you can roll back the release based on your update or deletion policies.

To update stacks, you can modify and resubmit the same template or create a change set; AWS CloudFormation will parse it for changes (add, modify, or delete) and apply the modifications to the resources. You use the AWS CloudFormation `UpdatePolicy` to determine how to respond to changes. When you delete a stack, by default all underlying stack resources also delete. *You use deletion policies to preserve resources when you delete a stack*.

AWS Auto Scaling group update policies enforce the behavior that will occur when an update is performed on an AWS Auto Scaling group. This depends on the type of change you make and whether you configure the AWS Auto Scaling scheduled actions. You can replace the entire AWS Auto Scaling group or only instances inside it. When you delete a stack, all underlying stack resources are deleted. You can apply the `DeletionPolicy` to resources in the stack to modify their behavior when the stack deletes.

You use stack exports to share information between separate stacks. Or, you can manage AWS CloudFormation stacks themselves as resources in a nested stack relationship. *You can export stack output values to import them into other stacks in the same account and region*. This allows you to share data that generates in one stack out to other stacks in your account.

You can assign a stack policy to a stack to allow or deny access to modify certain stack resources, which you can filter by the type of update. Stack policies protect all resources by default with an *implicit deny*. To allow access to actions on stack resources, you must apply *explicit allow* statements to the policy.

When you execute custom scripts on Amazon EC2 instances as part of your `UserData`, AWS CloudFormation provides several important helper scripts. You can use these to interact with the stack to query metadata, notify a `CreationPolicy` or `WaitCondition`, and process scripts when AWS CloudFormation detects metadata updates.

AWS CloudFormation *StackSets* give users the ability to control, provision, and manage stacks across multiple accounts and regions. A stack set as a logical container for stack information in an administrator account. Each stack set will contain information about stacks that you deploy to a single target account in one or more regions. Stack instances allow you to manage stacks in a target account. An update to a stack set propagates to all stack instances in all accounts and regions. When you perform operations on stack sets, you can configure how to control the flow of updates across accounts and regions. The administrator account requires an IAM service role with permissions to execute stack set operations and assume an execution role in any target account(s).

You can use a target account gate to perform evaluation tasks with AWS Lambda functions in the target account.

You cannot raise AWS CloudFormation template-specific limits through a support request. You can raise some limits, such as the number of stacks per account.

AWS CloudFormation has built-in integrations with AWS CodePipeline as a deployment provider. When a template revision passes through a pipeline, AWS CloudFormation can reference input parameters, stack policies, and other configuration data in the AWS CodePipeline deployment.

You can use change sets in a pipeline to include a manual review step to ensure that the changes you deploy are valid and desired before they actually execute with the use of the Action Mode.

Exam Essentials

Understand Infrastructure as Code (IaC). You model infrastructure as code to automate the provisioning, maintenance, and retirement of complex infrastructure across an organization. The declarative syntax allows you to describe the resource state you desire, instead of the steps to create it. You can version and maintain IaC with the same development workflow as application and configuration code.

Understand the purpose of change sets. Change sets allow administrators to preview the changes that will take place when a given template deploys to the AWS CloudFormation.

This includes a description of resources that you will update or replace entirely. You create change sets to help prevent stack updates that could accidentally result in the replacement of critical resources, such as databases.

Know the AWS CloudFormation permissions model. When you create, update, or delete stacks, AWS CloudFormation will operate with the same permissions as the IAM user or IAM role that performs the stack action. For example, a user who deletes a stack that contains an Amazon EC2 instance must also have the ability to terminate instances; otherwise, the stack delete fails. AWS CloudFormation also supports service roles, which you can pass to the service when you perform stack actions. This requires that the user or role have permissions to pass the service role to perform the stack action.

Know the AWS CloudFormation template structure. You can use these AWS CloudFormation template properties: `AWSTemplateFormatVersion`, `Description`, `Metadata`, `Parameters`, `Mappings`, `Conditions`, `Transform`, `Resources`, and `Outputs`. Templates only require the `Resources` property, and you must define at least one resource in every template.

Know how to use the intrinsic functions. It is important to understand the AWS CloudFormation templates intrinsic functions.

- `Fn::FindInMap`
- `Fn::GetAtt`
- `Fn::Join`
- `Fn::Split`
- `Ref`

Understand the purpose of `AWS::CloudFormation::Init`. This template section defines the configuration tasks the `cfn-init` helper script will perform on instances that you create individually or as part of AWS Auto Scaling launch configurations. This metadata key allows you to define a more declarative syntax for configuration tasks compared to using procedural steps in the `UserData` property.

Know the use cases for both custom resource types. You can implement custom resources with AWS Lambda functions or Amazon SNS topics. The primary difference between each type is that AWS Lambda-backed custom resources have a maximum execution duration of 5 minutes. This may not work for custom resources that take a long time to provision or update. In those cases, Amazon SNS topics backed by Amazon EC2 instances would allow for long running tasks.

Understand how AWS CloudFormation manages resource relationships. AWS CloudFormation will automatically reorder resource provisioning and update steps based on known dependencies. For example, if a template declares an Amazon VPC and a subnet, the subnet will not create before the Amazon VPC (a subnet requires an Amazon VPC ID during creation). However, AWS CloudFormation is not aware of all possible relationships, so you must manually declare them with the `DependsOn` property. If a template declares an Amazon EC2 instance and AWS DynamoDB table, and the table is referenced inside the instance's `UserData` property, you must declare a `DependsOn` property that states the instance depends on the table.

Understand wait conditions and creation policies. In some cases, resources in a template should wait for other resources to provision and configure before starting their tasks. For example, you may want to prevent creation of a load balancer resource until instances in an AWS Auto Scaling group have installed a web application. In those cases, you can use either wait conditions or creation policies. Wait conditions require you to add two separate resources to the template (`AWS::CloudFormation::WaitCondition` and `AWS::CloudFormation::WaitConditionHandle`). The instance's `UserData` property references the wait condition handle, where a success or failure signal will be sent. A creation policy does not require the additional resources, and it allows for additional options such as timeouts and signal counts.

Understand how stack updates affect resources. When you update a stack, resources may behave differently when properties update. If an Amazon S3 bucket is created as part of a stack and later the bucket policy is updated, the resource will update with no interruption. However, if the bucket name later updates, you must replace the bucket. Resources can undergo one of three types of updates: update with no interruption, update with some interruption, and replace update.

Know how to use exports and nested stacks to share stacks. Stack exports allow you to access stack outputs in other stacks in the same region. Exports, however, come with some limitations. For example, you cannot delete stacks that export values until all other stacks that import the exported value have been modified to no longer include the import. Nested stacks make use for the `AWS::CloudFormation::Stack` resource type. This way, a single stack can create multiple "child" stacks, which can declare their own resources (including other stacks). This is a useful mechanism to work around some service limits such as the number of resources per template (200).

Understand stack policies. To prevent updates to critical stack resources, you implement stack policies. A stack policy declares what resources you can and cannot update and under what circumstances. A stack containing an Amazon RDS instance, for example, can include a stack policy that prevents updates that require replacement of the database instance.

Resources to Review

AWS CloudFormation:

`https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html`

Infrastructure as Code:

`https://d1.awsstatic.com/whitepapers/DevOps/infrastructure-as-code.pdf`

AWS Quick Starts:

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

With Amazon Aurora Serverless, you also get the same high availability as traditional Amazon Aurora, which means that you get six-way replication across three Availability Zones inside of a region in order to prevent against data loss.

Amazon Aurora Serverless is great for infrequently used applications, new applications, variable workloads, unpredictable workloads, development and test databases, and multitenant applications. This is because you can scale automatically when you need to and scale down when application demand is not high. This can help cut costs and save you the heartache of managing your own database infrastructure.

Amazon Aurora Serverless is easy to set up, either through the console or directly with the CLI. To create an Amazon Aurora Serverless cluster with the CLI, you can run the following command:

```
aws rds create-db-cluster --db-cluster-identifier sample-cluster --engine aurora
--engine-version 5.6.10a \
--engine-mode serverless --scaling-configuration
MinCapacity=4,MaxCapacity=32,SecondsUntilAutoPause=1000,AutoPause=true \
--master-username user-name --master-user-password password \
--db-subnet-group-name mysubnetgroup --vpc-security-group-ids sg-c7e5b0d2
-region us-east-1
```

Amazon Aurora Serverless gives you many of the similar benefits as other serverless technologies, such as AWS Lambda, but from a database perspective. Managing databases is hard work, and with Amazon Aurora Serverless, you can utilize a database that automatically scales and you don't have to manage any of the underlying infrastructure.

AWS Serverless Application Model

The *AWS Serverless Application Model* (AWS SAM) allows you to create and manage resources in your serverless application with *AWS CloudFormation* to define your serverless application infrastructure as a SAM template. A *SAM template* is a JSON or YAML configuration file that describes the AWS Lambda functions, API endpoints, tables, and other resources in your application. With simple commands, you upload this template to AWS CloudFormation, which creates the individual resources and groups them into an *AWS CloudFormation stack* for ease of management. When you update your AWS SAM template, you re-deploy the changes to this stack. AWS CloudFormation updates the individual resources for you.

AWS SAM is an extension of AWS CloudFormation. You can define resources by using the AWS CloudFormation in your AWS SAM template. This is a powerful feature, as you can use AWS SAM to create a template of your serverless infrastructure, which you can then build into a DevOps pipeline. For example, examine the following:

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: 'Example of Multiple-Origin CORS using API Gateway and Lambda'
```

```
Resources:
  ExampleRoot:
    Type: 'AWS::Serverless::Function'
    Properties:
      CodeUri: '.'
      Handler: 'routes/root.handler'
      Runtime: 'nodejs8.10'
      Events:
        Get:
          Type: 'Api'
          Properties:
            Path: '/'
            Method: 'get'
  ExampleTest:
    Type: 'AWS::Serverless::Function'
    Properties:
      CodeUri: '.'
      Handler: 'routes/test.handler'
      Runtime: 'nodejs8.10'
      Events:
        Delete:
          Type: 'Api'
          Properties:
            Path: '/test'
            Method: 'delete'
        Options:
          Type: 'Api'
          Properties:
            Path: '/test'
            Method: 'options'

Outputs:
    ExampleApi:
      Description: "API Gateway endpoint URL for Prod stage for API Gateway
Multi-Origin CORS Function"
      Value: !Sub "https://${ServerlessRestApi}.execute-api.${AWS::Region}
.amazonaws.com/Prod/"
    ExampleRoot:
      Description: "API Gateway Multi-Origin CORS Lambda Function (Root) ARN"
      Value: !GetAtt ExampleRoot.Arn
    ExampleRootIamRole:
```

```
      Description: "Implicit IAM Role created for API Gateway Multi-Origin CORS
Function (Root)"
      Value: !GetAtt ExampleRootRole.Arn
    ExampleTest:
      Description: "API Gateway Multi-Origin CORS Lambda Function (Test) ARN"
      Value: !GetAtt ExampleTest.Arn
    ExampleTestIamRole:
      Description: "Implicit IAM Role created for API Gateway Multi-Origin CORS
Function (Test)"
      Value: !GetAtt ExampleTestRole.Arn
```

In the previous code example, you create two AWS Lambda functions and then associate *three* different Amazon API Gateway endpoints to trigger those functions. To deploy this AWS SAM template, download the template and all of the necessary dependencies from here:

```
https://github.com/awslabs/serverless-application-model/tree/develop/
examples/apps/api-gateway-multiple-origin-cors
```

AWS SAM is similar to AWS CloudFormation, with a few key differences, as shown in the second line:

```
Transform: 'AWS::Serverless-2016-10-31'
```

This important line of code transforms the AWS SAM template into an AWS CloudFormation template. Without it, the AWS SAM template will not work.

Similar to the AWS CloudFormation, you also have a `Resources` property where you define infrastructure to provision. The difference is that you provision serverless services with a new `Type` called `AWS::Serverless::Function`. This provisions an AWS Lambda function to define all properties from an AWS Lambda point of view. AWS Lambda includes `Properties`, such as `MemorySize`, `Timeout`, `Role`, `Runtime`, `Handler`, and others.

While you can create an AWS Lambda function with AWS CloudFormation using `AWS::Lambda::Function`, the benefit of AWS SAM lies in a property called `Event`, where you can tie in a trigger to an AWS Lambda function, all from within the `AWS::Serverless::Function` resource. This `Event` property makes it simple to provision an AWS Lambda function and configure it with an Amazon API Gateway trigger. If you use AWS CloudFormation, you would have to declare an Amazon API Gateway separately with `AWS::ApiGateway::Resource`.

To summarize, AWS SAM allows you to provision serverless resources more rapidly with less code by extending AWS CloudFormation.

AWS SAM CLI

Now that we've addressed AWS SAM, let's take a closer look at the AWS SAM CLI. With AWS SAM, you can define templates, in JSON or YAML, which are designed for provisioning serverless applications through AWS CloudFormation.

AWS SAM CLI is a command line interface tool that creates an environment in which you can develop, test, and analyze your serverless-based application, all locally. This allows you to test your AWS Lambda functions before uploading them to the AWS service. AWS SAM CLI also allows you to develop and test your code quickly, and this gives you the ability to test it locally, which allows you to develop it faster. Previously, you would have had to upload your code each time you wanted to test an AWS Lambda function. Now, with the AWS SAM CLI, you can develop faster and get your application out the door more quickly.

To use AWS SAM CLI, you must meet a few prerequisites. You must install Docker, have Python 2.7 or 3.6 installed, have pip installed, install the AWS CLI, and finally install the AWS SAM CLI. You can read more about how to install AWS SAM CLI at `https://github.com/awslabs/aws-sam-cli`.

With AWS SAM CLI, you must define three key things.

- You must have a valid AWS SAM template, which defines a serverless application.
- You must have the AWS Lambda function defined. This can be in any valid language that Lambda currently supports, such as Node.js, Java 8, Python, and so on.
- You must have an event source. An *event source* is simply an `event.json` file that contains all the data that the Lambda function expects to receive. Valid event sources are as follows:
  - Amazon Alexa
  - Amazon API Gateway
  - AWS Batch
  - AWS CloudFormation
  - Amazon CloudFront
  - AWS CodeCommit
  - AWS CodePipeline
  - Amazon Cognito
  - AWS Config
  - Amazon DynamoDB
  - Amazon Kinesis
  - Amazon Lex
  - Amazon Rekognition
  - Amazon Simple Storage Service (Amazon S3)
  - Amazon Simple Email Service (Amazon SES)
  - Amazon Simple Notification Service (Amazon SNS)
  - Amazon Simple Queue Service (Amazon SQS)
  - AWS Step Functions

To generate this JSON event source, you can simply run this command in the AWS SAM CLI:

```
sam local generate-event <service> <event>
```

AWS SAM CLI is a great tool that allows developers to iterate quickly on their serverless applications. You will learn how to create and test an AWS Lambda function locally in the "Exercises" section of this chapter.

AWS Serverless Application Repository

The *AWS Serverless Application Repository* enables you to deploy code samples, components, and complete applications quickly for common use cases, such as web and mobile backends, event and data processing, logging, monitoring, Internet of Things (IoT), and more. Each application is packaged with an AWS SAM template that defines the AWS resources. Publicly shared applications also include a link to the application's source code. There is no additional charge to use the serverless application repository. You pay only for the AWS resources you use in the applications you deploy.

You can also use the serverless application repository to publish your own applications and share them within your team, across your organization, or with the community at large. This allows you to see what other people and organizations are developing.

Serverless Application Use Cases

Case studies on running serverless applications are located at the following URLs:

The Coca-Cola Company:

```
https://aws.amazon.com/blogs/aws/things-go-better-with-step-functions/
```

FINRA:

```
https://aws.amazon.com/solutions/case-studies/finra-data-validation/
```

iRobot:

```
https://aws.amazon.com/solutions/case-studies/irobot/
```

Localytics:

```
https://aws.amazon.com/solutions/case-studies/localytics/
```

Summary

This chapter covered the AWS serverless core services, how to store your static files inside of Amazon S3, how to use Amazon CloudFront in conjunction with Amazon S3, how to integrate your application with user authentication flows using Amazon Cognito, and how to deploy and scale your API quickly and automatically with Amazon API Gateway.

Serverless applications have three main benefits: no server management, flexible scaling, and automated high availability. Without server management, you no longer have to provision or maintain servers. With AWS Lambda, you upload your code, run it, and focus on your application updates. With flexible scaling, you no longer have to disable Amazon EC2 instances to scale them vertically, groups do not need to be auto-scaled, and you do not need to create Amazon CloudWatch alarms to add them to load balancers. With AWS Lambda, you adjust the units of consumption (memory and execution time), and AWS adjusts the rest of the instance appropriately. Finally, serverless applications have built-in availability and fault tolerance. When periods of low traffic occur, you do not spend money on Amazon EC2 instances that do not run at their full capacity.

You can use an Amazon S3 web server to create your presentation tier. Within an Amazon S3 bucket, you can store HTML, CSS, and JavaScript files. JavaScript can create HTTP requests. These HTTP requests are sent to a REST endpoint service called Amazon API Gateway, which allows the application to save and retrieve data dynamically by triggering a Lambda function.

After you create your Amazon S3 bucket, you configure it to use static website hosting in the AWS Management Console and enter an endpoint that reflects your AWS Region.

Amazon S3 allows you to configure web traffic logs to capture information, such as the number of visitors who access your website in the Amazon S3 bucket.

One way to decrease latency and improve your performance is to use Amazon CloudFront with Amazon S3 to move your content closer to your end users. Amazon CloudFront is a serverless service.

The Amazon API Gateway is a fully managed service designed to define, deploy, and maintain APIs. Clients integrate with the APIs using standard HTTPS requests. Amazon API Gateway can integrate with a service-oriented multitier architecture. The Amazon API Gateway provides dynamic data in the logic or app tier.

There are three types of endpoints for Amazon API Gateway: regional endpoints, edge-optimized endpoints, and private endpoints.

In the Amazon API Gateway service, you expose addressable resources as a tree of API Resources entities, with the root resource (/) at the top of the hierarchy. The root resource is relative to the API's base URL, which consists of the API endpoint and a stage name.

You use Amazon API Gateways to help drive down the total response-time latency of your API. Amazon API Gateway uses the HTTP protocol to process these HTTP methods and send/receive data to and from the backend. Serverless data is sent to AWS Lambda to process.

You can use Amazon Route 53 to create a more user-friendly domain name instead of using the default host name (Amazon S3 endpoint). To support two subdomains, you create two Amazon S3 buckets that match your domain name and subdomain.

A stage is a named reference to a deployment, which is a snapshot of the API. Use a stage to manage and optimize a particular deployment. You create stages for each of your environments such as DEV, TEST, and PROD, so you can develop and update your API and applications without affecting production. Use Amazon API Gateway to set up authorizers with Amazon Cognito user pools on an AWS Lambda function. This enables you to secure your APIs.

An Amazon Cognito user pool includes a prebuilt user interface (UI) that you can use inside your application to build a user authentication flow quickly. A user pool is a user directory in Amazon Cognito. With a user pool, your users can sign in to your web or mobile app through Amazon Cognito. Users can also sign in through social identity providers such as Facebook or Amazon and through Security Assertion Markup Language (SAML) identity providers.

Amazon Cognito identity pools allow you to create unique identities and assign permissions for your users to help you integrate with authentication providers. With the combination of user pools and identity pools, you can create a serverless user authentication system.

You can choose how users sign in with a username, an email address, and/or a phone number and to select attributes. Attributes are properties that you want to store about your end users. You can also configure password policies. Multi-factor authentication (MFA) prevents anyone from signing in to a system without authenticating through two different sources, such as a password and a mobile device–generated token. You create an Amazon Cognito role to send Short Message Service (SMS) messages to users.

The AWS Serverless Application Model (AWS SAM) allows you to create and manage resources in your serverless application with AWS CloudFormation as a SAM template. A SAM template is a JSON or YAML file that describes the AWS Lambda function, API endpoints, and other resources. You upload the template to AWS CloudFormation to create a stack. When you update your AWS SAM template, you redeploy the changes to this stack, and AWS CloudFormation updates the resources. You can use AWS SAM to create a template of your serverless infrastructure, which you can then build into a DevOps pipeline.

The `Transform: 'AWS::Serverless-2016-10-31'` code converts the AWS SAM template into an AWS CloudFormation template.

The AWS Serverless Application Repository enables you to deploy code samples, components, and complete applications for common use cases. Each application is packaged with an AWS SAM template that defines the AWS resources.

Additionally, you learned the differences between the standard three-tier web applications and the AWS serverless stack. You learned how to build your infrastructure quickly with AWS SAM and AWS SAM CLI for testing and development purposes.

Exam Essentials

Know serverless applications' three main benefits. The benefits are as follows:

- No server management
- Flexible scaling
- Automated high availability

Know what no server management means. Without server management, you no longer have to provision or maintain servers. With AWS Lambda, you upload your code, run it, and focus on your application updates.

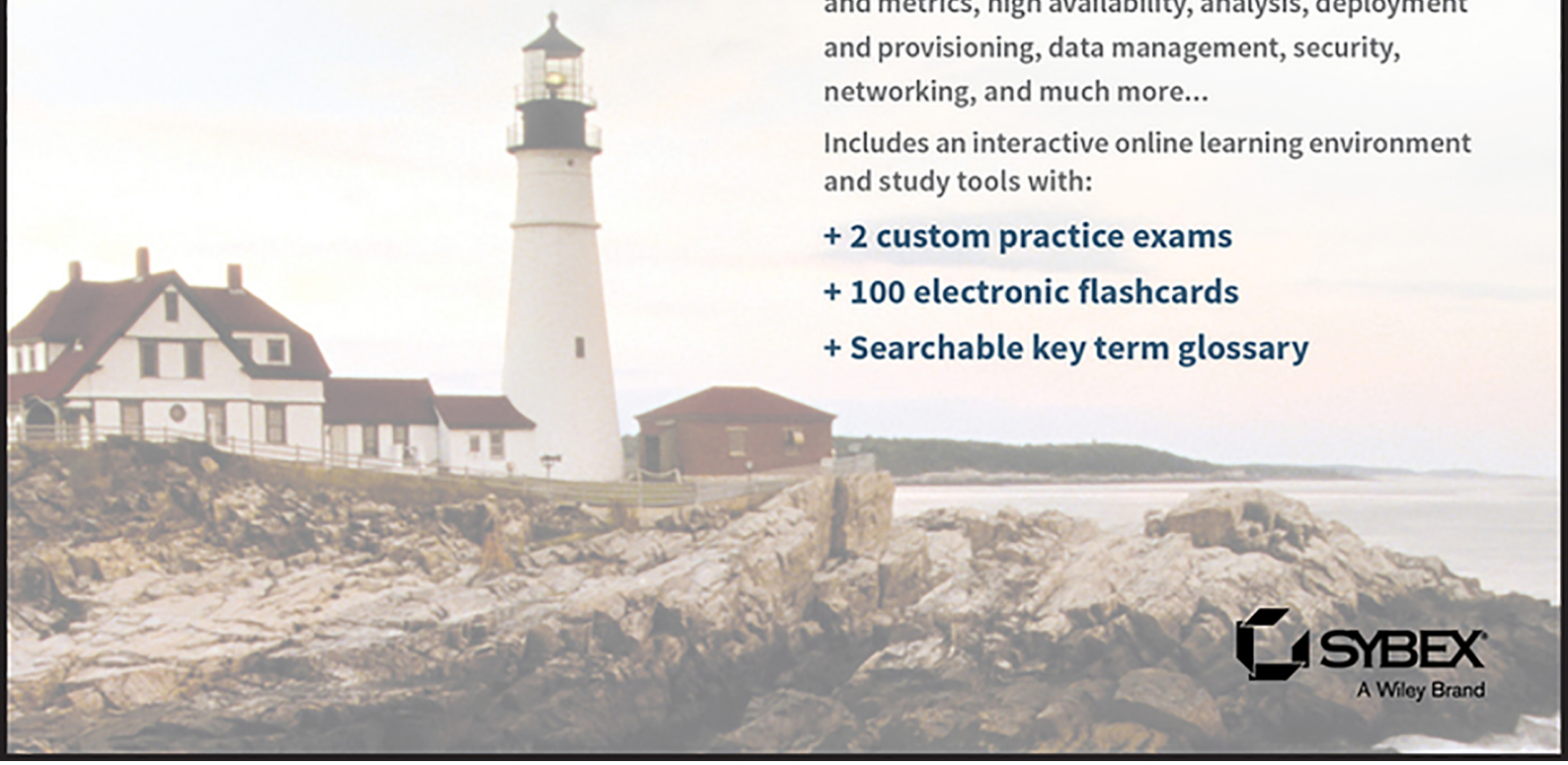
amazon web services

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.

JON BONSO AND KENNETH SAMONTE

AWS CERTIFIED

DEVOPS ENGINEER PROFESSIONAL

Tutorials Dojo
Study Guide and Cheat Sheets

	to AWS SNS or a Lambda function to send results to Slack channel
Need to run an AWS CodePipeline every day for updating the development progress status	Create CloudWatch Events rule to run on schedule every day and set a target to the AWS CodePipeline ARN
Automate deployment of a Lambda function and test for only 10% of traffic for 10 minutes before allowing 100% traffic flow.	Use CodeDeploy and select deployment configuration CodeDeployDefault.LambdaCanary10Percent10Minutes
Deployment of Elastic Beanstalk application with absolutely no downtime. The solution must maintain full compute capacity during deployment to avoid service degradation.	Choose the “Rolling with additional Batch” deployment policy in Elastic Beanstalk
Deployment of Elastic Beanstalk application where the new version must not be mixed with the current version.	Choose the “Immutable deployments” deployment policy in Elastic Beanstalk
Configuration Management and Infrastructure-as-Code	
The resources on the parent CloudFormation stack needs to be referenced by other nested CloudFormation stacks	Use Export on the Output field of the main CloudFormation stack and use Fn::ImportValue function to import the value on the other stacks
On which part of the CloudFormation template should you define the artifact zip file on the S3 bucket?	The artifact file is defined on the AWS::Lambda::Function code resource block
Need to define the AWS Lambda function inline in the CloudFormation template	On the AWS::Lambda::Function code resource block, the inline function must be enclosed inside the ZipFile section.
Use CloudFormation to update Auto Scaling Group and only terminate the old instances when the newly launched instances become fully operational	Set AutoScalingReplacingUpdate : WillReplace property to TRUE to have CloudFormation retain the old ASG until the instances on the new ASG are healthy.
You need to scale-down the EC2 instances at night when there is low traffic using OpsWorks.	Create *Time-based* instances for automatic scaling of predictable workload.

Can't install an agent on on-premises servers but need to collect information for migration	Deploy the Agentless Discovery Connector VM on your on-premises data center to collect information.
Syntax for CloudFormation with an Amazon ECS cluster with ALB	Use the AWS::ECS::Service element for the ECS Cluster, AWS::ECS::TaskDefinition element for the ECS Task Definitions and the AWS::ElasticLoadBalancingV2::LoadBalancer element for the ALB.
Monitoring and Logging	
Need to centralize audit and collect configuration setting on all regions of multiple accounts	Setup an Aggregator on AWS Config.
Consolidate CloudTrail log files from multiple AWS accounts	Create a central S3 bucket with bucket policy to grant cross-account permission. Set this as destination bucket on the CloudTrail of the other AWS accounts.
Ensure that CloudTrail logs on the S3 bucket are protected and cannot be tampered with.	Enable Log File Validation on CloudTrail settings
Need to collect/investigate application logs from EC2 or on-premises server	Install CloudWatch Logs Agent to send the logs to CloudWatch Logs for storage and viewing.
Need to review logs from running ECS Fargate tasks	Enable awslogs log driver on the Task Definition and add the required logConfiguration parameter.
Need to run real-time analysis for collected application logs	Send logs to CloudWatch Logs, create a Lambda subscription filter, Elasticsearch subscription filter, or Kinesis stream filter.
Need to be automatically notified if you are reaching the limit of running EC2 instances or limit of Auto Scaling Groups	Track service limits with Trusted Advisor on CloudWatch Alarms using the ServiceLimitUsage metric.
Policies and Standards Automation	
Need to secure the buildspec.yml file which contains the AWS keys and database password stored in plaintext.	Store these values as encrypted parameter on SSM Parameter Store

What is Infrastructure-as-Code (IaC)?

Infrastructure-as-code (IaC) takes the concept of configuration management to the next level. Imagine your entire AWS infrastructure and resources described inside a YAML or JSON file. Just like how your application source code outputs an artifact, IaC generates a consistent environment when you apply it.

For example, Infrastructure as Code enables DevOps teams to easily and quickly create test environments that are similar to the production environment. IaC allows you to deliver stability rapidly, consistently, and at scale.

Another example is when you need to create a Disaster Recovery site in another region. With IaC, you can quickly create resources on the new region and be assured that the environment is consistent with the current live environment because everything is defined and described on your JSON or YAML code. You can also save your code to repositories and you can version control it to track changes on your infrastructure. AWS CloudFormation is the main service that you can use if you have codified your infrastructure.

AWS CloudFormation gives developers and systems administrators an easy way to create and manage a collection of related AWS resources, provisioning and updating them in an orderly and predictable fashion. You can use templates to describe the AWS resources and any associated dependencies or runtime parameters required to run your application.

Benefits of using CloudFormation

- **Extensibility** - Using the AWS CloudFormation Registry, you can model, provision, and manage third party application resources alongside AWS resources with AWS CloudFormation.
- **Authoring with JSON/YAML** - allows you to model your entire infrastructure in a text file. You can use JSON or YAML to describe what AWS resources you want to create and configure.
- **Safety controls** - automates the provisioning and updating of your infrastructure in a safe and controlled manner. You can use Rollback Triggers to rollback in case errors are encountered during the update.
- **Preview changes to your environment** - Change Sets allow you to preview how proposed changes to a stack might impact your running resources. For example, whether your changes will delete or replace any critical resources.
- **Dependency management** - automatically manages dependencies between your resources during stack management actions. The sequence of creating, updating, or deleting the dependencies and resources are automatically taken care of.
- **Cross account & cross-region management** - AWS StackSets that lets you provision a common set of AWS resources across multiple accounts and regions with a single CloudFormation template.

Sources:
https://aws.amazon.com/opsworks/
https://aws.amazon.com/cloudformation/

CloudFormation Cross-Stack Reference

CloudFormation allows you to reference resources from one CloudFormation stack and use those resources on another stack. This is called **cross-stack reference**. It allows for a layering of stacks, which is useful for separating your resources based on your services. Instead of putting all resources on one stack, you can create resources from one stack and reference those resources on other CloudFormation stacks.

This also allows you to re-use the same CloudFormation stacks so that you can build faster if you need a new environment with minimal changes.

Example:

- Network stack – contains VPC, public and private subnets, and security groups.
- Web server stack – contains webserver and referencing the public subnets and security groups from the network stack
- Database stack – contains your database server and referencing the private subnets and security groups from the network stack

DevOps Exam Notes:

The requirement for cross stack reference is that you need to export the resources that you want to be referenced by other stacks. Use **Export** on the output Field of your main CloudFormation stack to define the resources that you want to expose to other stacks. On the other stacks, use the **Fn::ImportValue** intrinsic function to import the value that was previously exported.

Here's an example of CloudFormation exporting a subnet and a security group, and referencing it on another CloudFormation stack.

Source:
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/walkthrough-crossstackref.html

Lambda Function Artifact from S3 or CloudFormation Inline

S3 bucket on CloudFormation

Using AWS CloudFormation, you can deploy AWS Lambda functions, which is an easy way to reliably reproduce and version your application deployments.

DevOps Exam Notes:

On your CloudFormation template, the **AWS::Lambda::Function** resource creates a Lambda function. To create a function, you need a deployment package and an execution role. The deployment package contains your function code. The artifact file is defined on the **AWS::Lambda::Function Code** resource.

Here is an example of Node.js lambda function that uses an artifact saved on an S3 bucket on CloudFormation (uses JSON Format).

```
            "AMIIDLookup": {
                  "Type": "AWS::Lambda::Function",
                  "Properties": {
                  "Handler": "index.handler",
                  "Role": {
                        "Fn::GetAtt": [
                            "LambdaExecutionRole",
                              "Arn"
                        ]
                  },
                  "Code": {
                        "S3Bucket": "lambda-functions",
                        "S3Key": "amilookup.zip"
                  },
                  "Runtime": "nodejs12.x",
                  "Timeout": 25,
                  "TracingConfig": {
                        "Mode": "Active"
                  }
                  }
            }
```

Note that Changes to a deployment package in Amazon S3 are not detected automatically during stack updates. To update the function code, change the object key or version in the template. Or you can use a

parameter on your CloudFormation template to input the name of the artifact on S3 that contains the latest version of your code.

If you have a zip file on your local machine, you can use the package command to create a template that generates a template that you can use.

```
aws cloudformation package --template /path_to_template/template.json --s3-bucket mybucket
--output json > packaged-template.json
```

This will save the output on a json template that you can upload on CloudFormation.

Inline Lambda functions in CloudFormation

For Node.js and Python functions, you can specify the function code inline in the template.

> **DevOps Exam Notes:**
>
> Note that you need to use the **AWS::Lambda::Function Code** to define that function and it should be enclosed inside the **ZipFile: |** section to ensure that CloudFormation correctly parses your code.

Here is an example of Node.js lambda function inline on CloudFormation template using the YAML format.

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Lambda function with cfn-response.
Resources:
  primer:
      Type: AWS::Lambda::Function
      Properties:
      Runtime: nodejs12.x
      Role: arn:aws:iam::123456789012:role/lambda-role
      Handler: index.handler
      Code:
      ZipFile: |
            var aws = require('aws-sdk')
            var response = require('cfn-response')
            exports.handler = function(event, context) {
            console.log("REQUEST RECEIVED:\n" + JSON.stringify(event))
            // For Delete requests, immediately send a SUCCESS response.
            if (event.RequestType == "Delete") {
                  response.send(event, context, "SUCCESS")
                  return
            }
```

```
                var responseStatus = "FAILED"
                var responseData = {}
                var functionName = event.ResourceProperties.FunctionName
                var lambda = new aws.Lambda()
                lambda.invoke({ FunctionName: functionName }, function(err,
      invokeResult) {
                      if (err) {
                            responseData = {Error: "Invoke call failed"}
                          console.log(responseData.Error + ":\n", err)
                      }
                      else responseStatus = "SUCCESS"
                      response.send(event, context, responseStatus, responseData)
                })
                }
          Description: Invoke a function during stack creation.
          TracingConfig:
          Mode: Active
```

Sources:
https://docs.aws.amazon.com/lambda/latest/dg/deploying-lambda-apps.html
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-cfn-cli-package.html

AutoScalingReplacingUpdate vs AutoScalingRollingUpdate Policy

When using AWS CloudFormation to provision your Auto Scaling groups, you can control how CloudFormation handles updates for your Auto Scaling Group. You need to define the proper **UpdatePolicy** attribute for your ASG depending on your desired behavior during an update.

DevOps Exam Notes:

You can use **AWS::AutoScaling::AutoScalingGroup** resource type on CloudFormation if you want to create an AutoScaling group for your fleet of EC2 instances. Going into the exam, you will need to distinguish the difference between the **AutoScalingReplacingUpdate** and **AutoScalingRollingUpdate** UpdatePolicy attribute, which define how the instances on your group will be updated when you deploy a new revision of your application.

AutoScalingReplacingUpdate - will create a new auto scaling group with new launch configuration. This is more like an immutable type of deployment.

AutoScalingRollingUpdate - will replace the instances on the current auto-scaling group. You can control if instances will be replaced “all-at-once” or use a rolling update by batches. The default behavior is to delete instances first, before creating the new instances.

The **AutoScalingReplacingUpdate** policy specifies how AWS CloudFormation handles replacement updates for an Auto Scaling group. This policy enables you to specify whether AWS CloudFormation replaces an Auto Scaling group with a new one or replaces only the instances in the Auto Scaling group.

```
"UpdatePolicy" : {
  "AutoScalingReplacingUpdate" : {
    "WillReplace" : Boolean
  }
}
```

For example, you can set **AutoScalingReplacingUpdate WillReplace** property to TRUE to have CloudFormation retain the old ASG and the instances it contains. CloudFormation will wait for the successful creation of the new ASG and its instances before it deletes the old ASG. This is helpful when the update fails; CloudFormation can quickly rollback as it will only delete the new ASG. The current ASG and its instances are not affected during the deployment and rollback process.

The **AutoScalingRollingUpdate** policy specifies how AWS CloudFormation handles rolling updates for an Auto Scaling group. Rolling updates enable you to specify whether AWS CloudFormation updates instances that are in an Auto Scaling group in batches or all at once.

```
"UpdatePolicy" : {
  "AutoScalingRollingUpdate" : {
      "MaxBatchSize" : Integer,
      "MinInstancesInService" : Integer,
      "MinSuccessfulInstancesPercent" : Integer,
      "PauseTime" : String,
      "SuspendProcesses" : [ List of processes ],
      "WaitOnResourceSignals" : Boolean
  }
}
```

For example, **AutoScalingRollingUpdate** allows you to specify the **MaxBatchSize** property to set the maximum number of instances that AWS CloudFormation updates at any given time. Or use the **MinInstancesInService** property to ensure that there is a minimum number of instances that are in service while CloudFormation updates the old instances.

Sources:
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-attribute-updatepolicy.html
https://aws.amazon.com/premiumsupport/knowledge-center/auto-scaling-group-rolling-updates/

CloudFormation Template for ECS, Auto Scaling and ALB

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

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

DevOps Exam Notes:

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

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

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

      },

……

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

      "Properties":{

……

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

```
        "Properties":{

……

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

        },
```

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

AWS CloudFormation

- A service that gives developers and businesses an easy way to create a collection of related AWS resources and provision them in an orderly and predictable fashion.

Features

- CloudFormation allows you to model your entire infrastructure in a text file called a **template**. You can use JSON or YAML to describe what AWS resources you want to create and configure. If you want to design visually, you can use *AWS CloudFormation Designer*.
- CloudFormation automates the provisioning and updating of your infrastructure in a safe and controlled manner. You can use **Rollback Triggers** to specify the CloudWatch alarm that CloudFormation should monitor during the stack creation and update process. If any of the alarms are breached, CloudFormation rolls back the entire stack operation to a previously deployed state.
- CloudFormation enables you to build custom extensions to your stack template using AWS Lambda.

Concepts

- **Templates**
  - A JSON or YAML formatted text file.
  - CloudFormation uses these templates as blueprints for building your AWS resources.
- **Stacks**
  - Manage related resources as a single unit.
  - All the resources in a stack are defined by the stack's CloudFormation template.
- **Change Sets**
  - Before updating your stack and making changes to your resources, you can generate a change set, which is a summary of your proposed changes.
  - Change sets allow you to see how your changes might impact your running resources, especially for critical resources, before implementing them.
- With AWS CloudFormation and AWS CodePipeline, you can use continuous delivery to automatically build and test changes to your CloudFormation templates before promoting them to production stacks.
- *CloudFormation artifacts* can include a stack template file, a template configuration file, or both. AWS CodePipeline uses these artifacts to work with CloudFormation stacks and change sets.
  - **Stack Template File** - defines the resources that CloudFormation provisions and configures. You can use YAML or JSON-formatted templates.
  - **Template Configuration File** - a JSON-formatted text file that can specify template parameter values, a stack policy, and tags. Use these configuration files to specify parameter values or a stack policy for a stack.

Stacks

- If a resource cannot be created, CloudFormation rolls the stack back 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.
- Stack update methods
    - Direct update
    - Creating and executing change sets
- **Drift detection** enables you to detect whether a stack's actual configuration differs, or has drifted, from its expected configuration. Use CloudFormation to detect drift on an entire stack, or on individual resources within the stack.
    - A resource is considered to have drifted if any if its actual property values differ from the expected property values.
    - A stack is considered to have drifted if one or more of its resources have drifted.
- To share information between stacks, export a stack's output values. Other stacks that are in the same AWS account and region can import the exported values.
- You can nest stacks.

Templates

- Templates include several major sections. The Resources section is the only required section.
- **CloudFormation Designer** is a graphic tool for creating, viewing, and modifying CloudFormation templates. You can diagram your template resources using a drag-and-drop interface, and then edit their details using the integrated JSON and YAML editor.
- Custom resources enable you to write custom provisioning logic in templates that CloudFormation runs anytime you create, update (if you change the custom resource), or delete stacks.
- Template macros enable you to perform custom processing on templates, from simple actions like find-and-replace operations to extensive transformations of entire templates.

StackSets

- CloudFormation StackSets allow you to roll out CloudFormation stacks over multiple AWS accounts and in multiple Regions with just a couple of clicks. StackSets is commonly used together with AWS Organizations to centrally deploy and manage services in different accounts.
- Administrator and target accounts - An *administrator account* is the AWS account in which you create stack sets. A stack set is managed by signing in to the AWS administrator account in which it was created. A *target account* is the account into which you create, update, or delete one or more stacks in your stack set.
- Stack sets - A *stack set* lets you create stacks in AWS accounts across regions by using a single CloudFormation template. All the resources included in each stack are defined by the stack set's CloudFormation template. A stack set is a regional resource.
- Stack instances - A *stack instance* is a reference to a stack in a target account within a region. A stack instance can exist without a stack, and can be associated with only one stack set.

- Stack set operations - Create stack set, update stack set, delete stacks, and delete stack set.
- Tags - You can add tags during stack set creation and update operations by specifying key and value pairs.
- Drift detection identifies unmanaged changes or changes made to stacks outside of CloudFormation. When CloudFormation performs drift detection on a stack set, it performs drift detection on the stack associated with each stack instance in the stack set. If the current state of a resource varies from its expected state, that resource is considered to have drifted.
- If one or more resources in a stack has drifted then the stack itself is considered to have drifted, and the stack instances that the stack is associated with is considered to have drifted as well.
- If one or more stack instances in a stack set has drifted, the stack set itself is considered to have drifted.