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

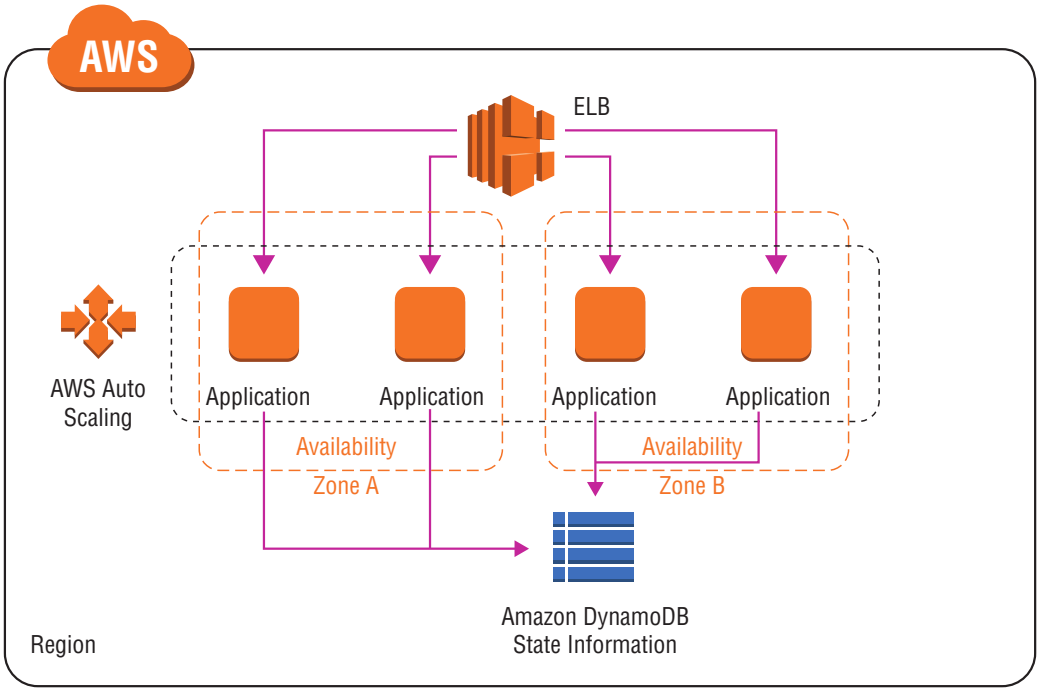
Load balancing is an integral part to directing and managing traffic among your instances. As you launch applications in your environments, you will want them to have high performance and high availability for your users. To enable both of these features, a load balancer will be necessary.

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:

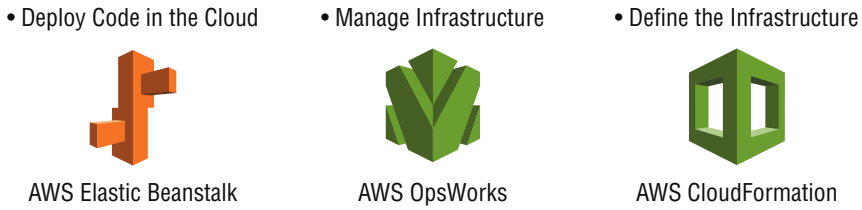
- Multiple Availability Zones and AWS Regions.
- Health check and failover mechanism.
- Stateless application that stores the session state in a cache server or database.
- AWS services that help you to achieve your goal. For example, Auto Scaling helps you maintain high availability and scalability.

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" ]
    }
]
```

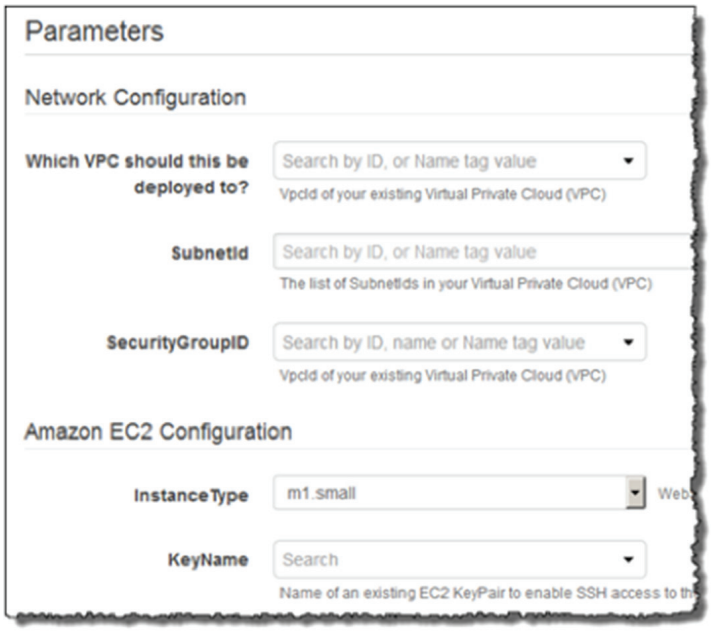
PARAMETERLABELS

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.

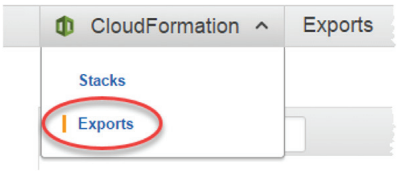
To export a stack value, update the `Outputs` section to include an `Export` declaration for every output you want to share.

```
"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]
```

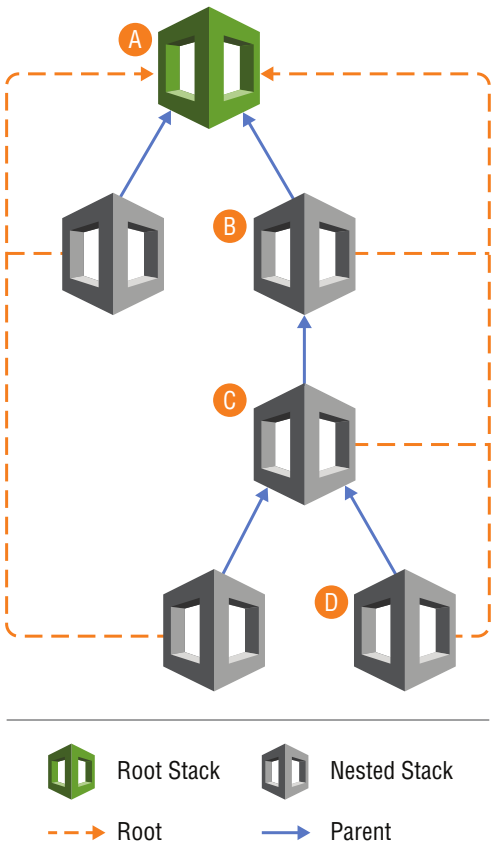
Nesting with the AWS::CloudFormation::Stack Resource

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

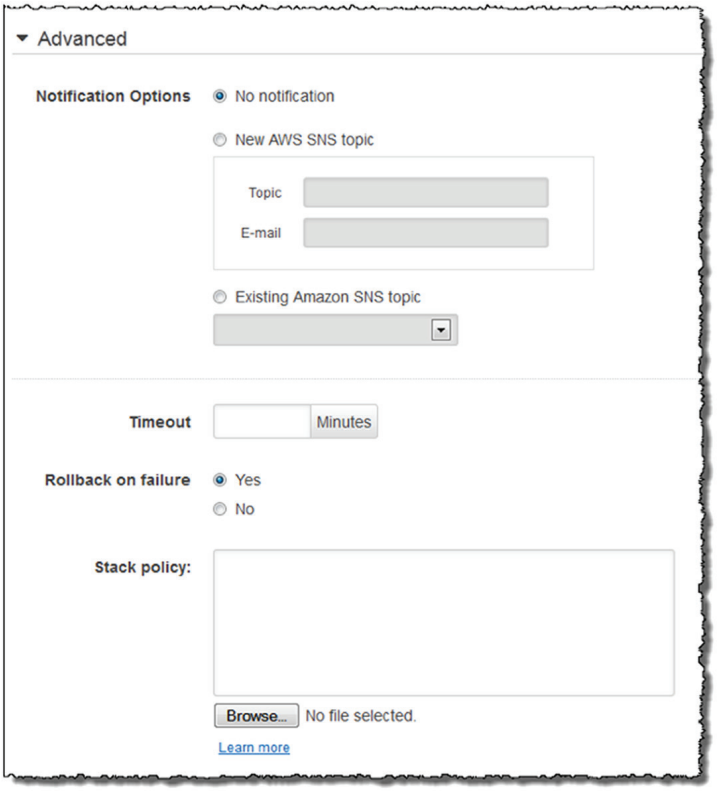
`Update:Replace` Update actions where replacement resources create (the physical ID of the resource changes)

`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>
```

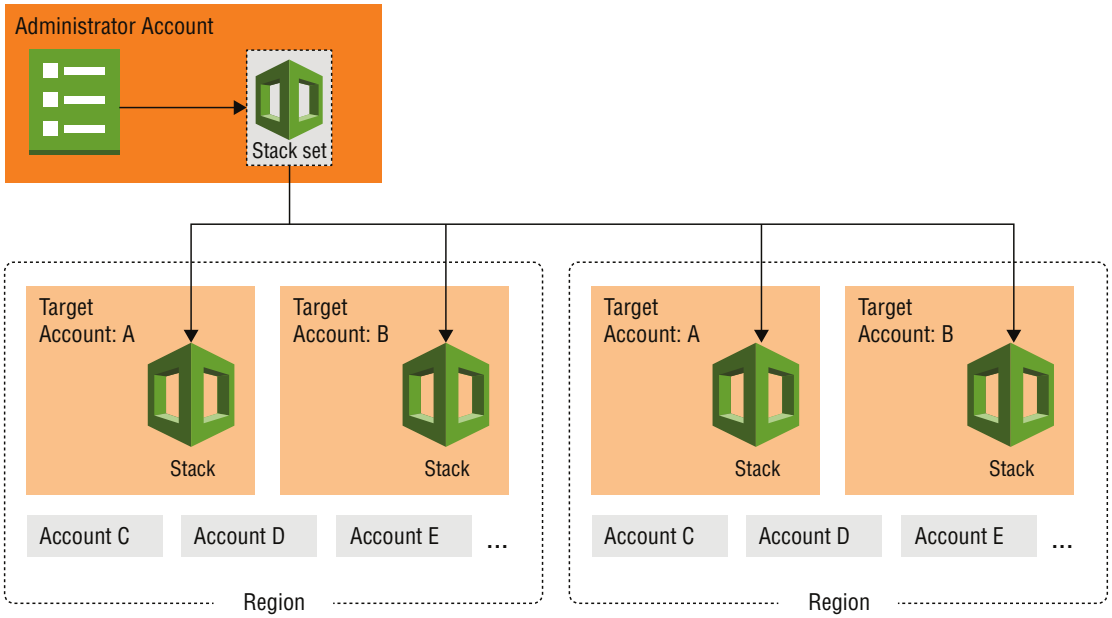
This template snippet demonstrates how to add a `cfn-hup` hook file to instances in an `AWS::AutoScaling::LaunchConfiguration` resource. This hook file will detect updates to the `LaunchConfig` resource and execute the `wordpress_install` config set you specify in the `AWS::CloudFormation::Init` metadata.

```
[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

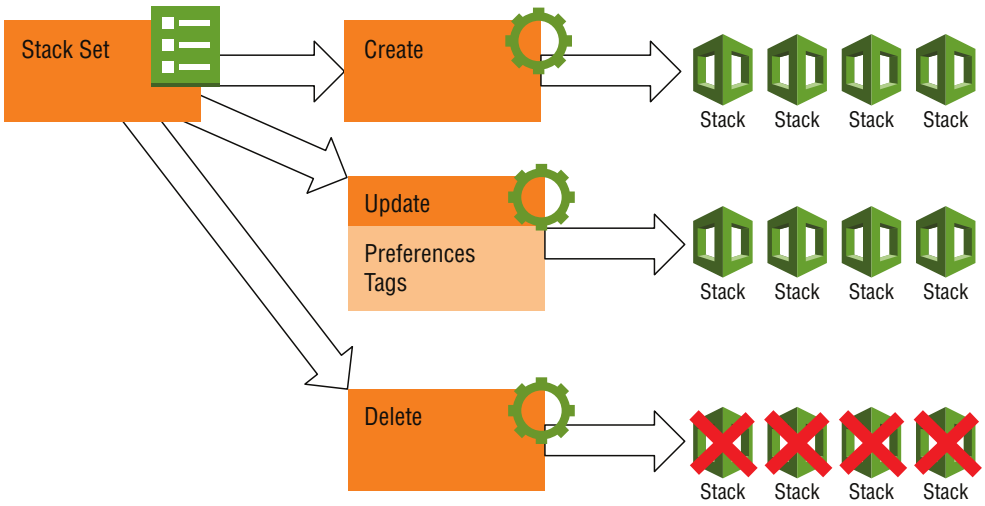
A *stack set* acts as a logical container for stack information in an administrator account. Each stack set will contain information about the stacks you deploy to a single target account in one or more regions. You can configure stack sets to deploy to regions in a specific order and how many unsuccessful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

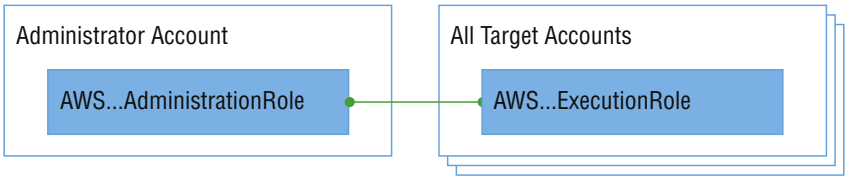
Stack Set Permissions

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.

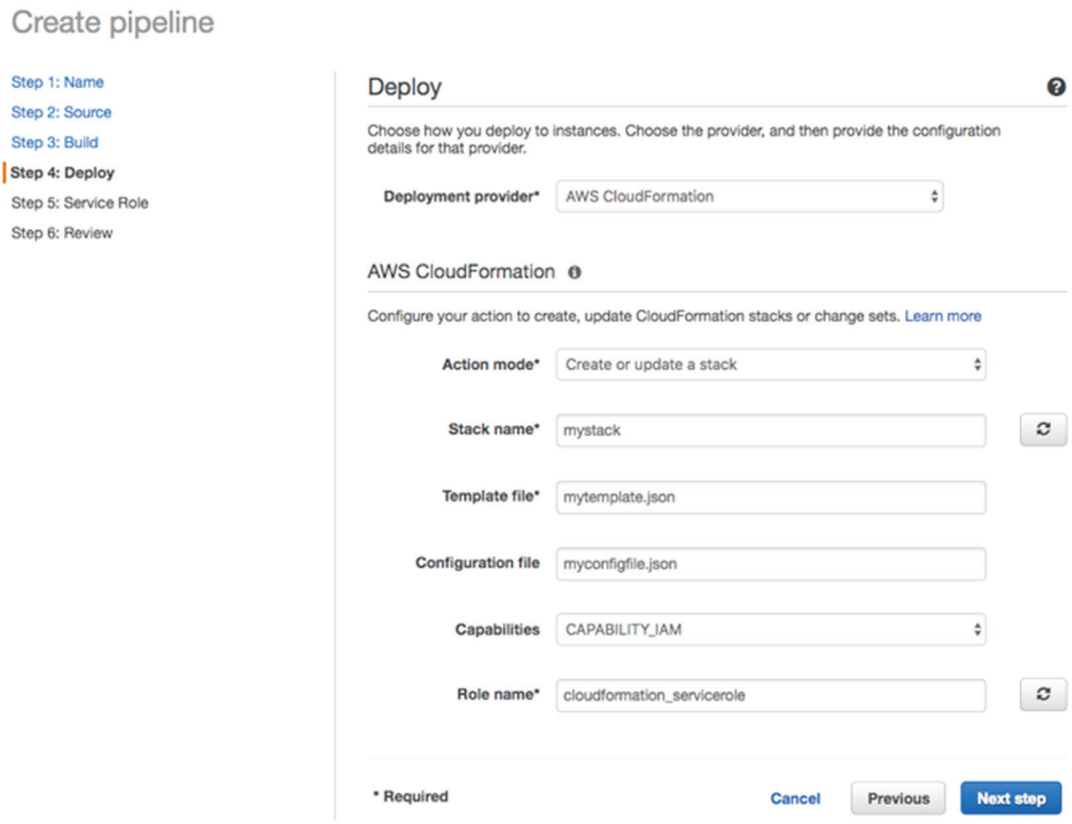
TABLE 8.3 AWS CloudFormation Service Limits

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.