

# Chapter 9



AWS® Certified Developer Official Study Guide  
By Nick Alteen, Jennifer Fisher, Casey Gerena, Wes Gruver, Asim Jalil,  
Heiwad Osman, Marife Pagan, Santosh Patlolla and Michael Roth  
Copyright © 2019 by Amazon Web Services, Inc.

# Configuration as Code

---

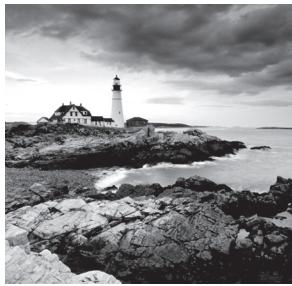
**THE AWS CERTIFIED DEVELOPER –  
ASSOCIATE EXAM TOPICS COVERED IN  
THIS CHAPTER MAY INCLUDE, BUT ARE  
NOT LIMITED TO, THE FOLLOWING:**

## Domain 1: Deployment

- ✓ 1.1 Deploy Serverless Applications.
- ✓ 1.2 Use AWS OpsWorks Stacks to Deploy Applications.
- ✓ 1.3 Use Amazon Elastic Container Service (Amazon ECS) to Deploy Containers.

## Domain 3: Development with AWS Services

- ✓ 3.1 Write code for serverless applications.
- ✓ 3.2 Write code that interacts with AWS services by using APIs, SDKs, and AWS CLI.



## Introduction to Configuration as Code

To expand on the theme of automation, you can add configuration to AWS enterprise as code.

AWS CloudFormation leverages standard AWS APIs to provision and update infrastructure in your account. Though AWS CloudFormation is highly effective at this task, there are some configuration tasks that are either inaccessible from the AWS API or more easily done with standard configuration management tools, such as Chef and Puppet. *AWS OpsWorks Stacks* provides a serverless Chef infrastructure to configure servers with Chef code, known as *recipes*. Much like AWS CloudFormation templates, Chef recipe code is declarative in nature. This means you do not have to rely on the accuracy of procedural steps, as you would with a userdata script you apply to Amazon Elastic Compute Cloud (Amazon EC2) instances or launch configurations. If you separate infrastructure from configuration, you also gain the ability to update each on separate cadences.



If a security vulnerability is found that requires a configuration update, use a recipe update. When you do the same in AWS CloudFormation, this requires a combination of userdata updates and cfn-hup configuration. A specific configuration management tool, however, requires only a new configuration code to submit to the instance, which will digest and apply the changes automatically.

In containerized environments, configuration of the container itself must also be done. *Amazon Elastic Container Service* (Amazon ECS) allows you to define the requirements of, schedule, and configure Docker containers to deploy to a cluster of Amazon EC2 instances. The cluster itself can be easily provisioned with AWS CloudFormation, along with configuration of container requirements such as CPU and memory needs. By combining this with configuration management tools, both the cluster and any active containers can be configured dynamically.

# Using AWS OpsWorks Stacks to Deploy Applications

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

## What Is AWS OpsWorks Stacks?

*AWS OpsWorks Stacks* is the only service that performs configuration management tasks. *Configuration management* is the process designed to ensure that infrastructure in a given system adheres to a specific set of standards, settings, or attributes (its configuration). Popular configuration management tools include Chef and Puppet. AWS OpsWorks Stacks allows you to manage the configuration of both on-premises and cloud infrastructures. To accomplish this, you organize units of infrastructure into stacks and layers. AWS OpsWorks Stacks can also perform application deployments by the configuration of apps. You can implement configuration changes at specific times in the lifecycle of your infrastructure through the use of lifecycle events, such as when an instance is first brought online or offline. Unlike traditional Chef Server installations, AWS OpsWorks Stacks uses Chef Zero, Chef Solo, or local mode Chef Client. You do not need to involve an actual Chef Server in the configuration management process.

There are two additional AWS OpsWorks services, *AWS OpsWorks for Chef Automate* and *AWS OpsWorks for Puppet Enterprise*. Unlike AWS OpsWorks Stacks, both services provision an Amazon EC2 instance in your AWS account with either Chef Automate or Puppet Enterprise software. Table 9.1 lists key differences between each service.

**TABLE 9.1** AWS OpsWorks Services

	AWS OpsWorks Stacks	AWS OpsWorks for Chef Automate	AWS OpsWorks for Puppet Enterprise
Manage infrastructure	X		
Chef	X	X	

**TABLE 9.1** AWS OpsWorks Services (*continued*)

	AWS OpsWorks Stacks	AWS OpsWorks for Chef Automate	AWS OpsWorks for Puppet Enterprise
Puppet			X
Code repository		X	X
Built-in automatic scaling	X		
Amazon EC2 Auto Scaling	X	X	X
Compliance		X	

**Code repository** Unlike AWS OpsWorks Stacks, the other two AWS OpsWorks services create an Amazon EC2 instance in your account. This instance will store any Chef or Puppet code for access by any cloud or on-premises instances (nodes) in your environment. AWS OpsWorks Stacks, however, requires you to store Chef code in an external location such as an Amazon Simple Storage Service (Amazon S3) bucket.

**Amazon EC2 Auto scaling** AWS OpsWorks includes the ability to have instances automatically come online in response to changes in demand. All three AWS OpsWorks services also support “traditional” automatic scaling.

**Chef compliance** You can use Chef Compliance to track, alert, report on, and remediate compliance violations in your infrastructure. For example, if your organization has strict requirements on SSH access to Linux systems, you can use InSpec (<https://www.inspec.io>) policies in Chef Compliance to scan nodes in your environment periodically for violations of the current SSH policy.

AWS OpsWorks Stacks supports these Chef versions:

- Chef 11.10 (Linux)
- Chef 12.0 (Linux)
- Chef 12.2 (Windows)

## AWS OpsWorks Stack Concepts

This section details AWS OpsWorks Stack concepts including cookbooks, recipes, packaging, stacks, layers, instances, apps, users, permissions, lifecycle events, resources, data bags, Chef, and monitoring your configuration.

## Cookbooks and Recipes



AWS OpsWorks Stacks leverages Chef to implement configuration. Chef itself is not in scope for the AWS Certified Developer – Associate exam. For more information, refer to the Chef training material (<https://learn.chef.io>).

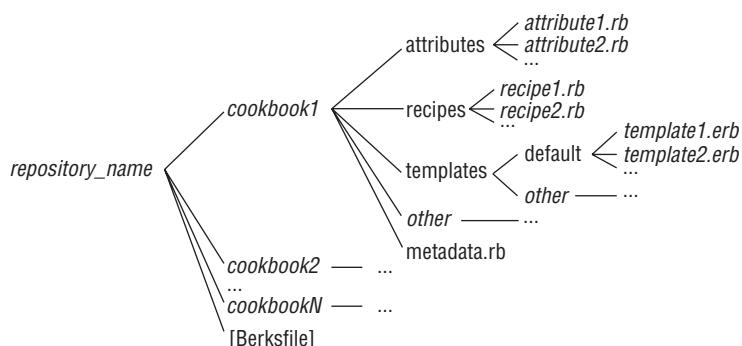
*Chef* is a Ruby-based configuration management language that AWS OpsWorks Stacks uses to enforce configuration on Amazon EC2 on-premises instance nodes. Chef uses a declarative syntax to describe how to configure a node without detailing the actual steps to achieve the desired configuration. Chef organizes these declarative statements into *recipes*, which act as a collection of resources to configure on nodes.

```
template '/tmp/somefile' do
  mode '0755'
  source 'somefile.erb'
  not_if { File.exist?('/etc/passwd') }
end
```

In the previous example, the file `/tmp/somefile` is created from the `somefile.erb` template. This template file is in a *cookbook*, which acts as a container for recipes and any files, templates, attributes, or other components to enforce configuration on a node. *Attribute files* provide data to recipes, and AWS OpsWorks Stacks can modify them with custom JSON at the stack, layer, and deployment levels. You can copy files from cookbooks to nodes to create static files and use templates to provide dynamic data to files before you create them on the node.

Cookbooks normally belong to a cookbook repository, or *chef-repo*, which is a versioned directory that contains cookbooks and their recipes. A single cookbook repository can contain one or more cookbooks. When you set the cookbook repository for a stack, all cookbooks in the repository copy to each instance in the stack. The directory structure of a *chef-repo* must match Figure 9.1.

**FIGURE 9.1** Cookbook repository structure



Recipes use nodes to execute in their “run list.” Recipes that you assign to a node’s run list execute in the order in which they appear. You can assign recipes to roles, which you assign to a node’s run list.

### Assigning Recipes to Roles

If a role named database-server contains two recipes, postgresql::default and monitoring::default, the following two run lists are the same:

```
RUN_LIST="recipe['postgresql::default'],recipe['monitoring::default']"  
RUN_LIST="role['database-server']"
```

In a typical Chef installation, one or more Chef Servers manage multiple nodes across an enterprise. The Chef Server is responsible for distributing cookbooks, managing node information, and providing data to nodes as the Chef runs (an execution of `chef-client` on a node that enforces any assigned recipes).



In AWS OpsWorks Stacks, there is no Chef Server to manage nodes. Chef is run with `chef-client` in local mode on the instance. Local mode creates an in-memory Chef Server to duplicate needed functionality.

### Managing Cookbooks

To install custom cookbooks in your stack, you first have to enable the Use Custom Chef Cookbooks field in the stack properties. Once you enable this field, you can then provide the details of the cookbook repository, such as the Git Repository URL, as shown in Figure 9.2.

**FIGURE 9.2** Enabling custom cookbooks

Use custom Chef cookbooks	<input checked="" type="checkbox"/> Yes
Repository type	Git
Repository URL	<input type="text" value="https://github.com/awslabs/op..."/>
Repository SSH key	Optional
Branch/Revision	Optional
Stack color	

When instances first create and start in a stack, they will download custom cookbooks from the repository you select. However, running instances will not download new cookbooks automatically. You must manually set the Run Command’s Command option to Update Custom Cookbooks, as shown in Figure 9.3.

**FIGURE 9.3** Running a command

The screenshot shows the 'Run Command' configuration page. The 'Command' dropdown is set to 'Update Custom Cookbooks', which is described as 'Deploys an updated set of custom Chef cookbooks from the repository to each instance's cookbooks cache.' The 'Comment' field is labeled 'Optional'. In the 'Instances' section, the 'Rails App Server' layer is selected, and the 'MySQL' layer is also selected. Within the 'Rails App Server' layer, 'rails-app1' is selected. Within the 'MySQL' layer, 'db-master1' is selected. At the bottom right, there are 'Cancel' and 'Update Custom Cookbooks' buttons.



You cannot manually start stopped Amazon EBS backed load-based and time-based instances, and thus you must replace the instance to update custom cookbooks.

## Package Cookbook Dependencies

Chef provides a utility called *Berkshelf* (<https://docs.chef.io/berkshelf.html>) to manage dependencies of cookbooks throughout the development and deployment process. In Chef 11.10 stacks, you can install Berkshelf automatically on any instances in your stack. For a production environment, AWS recommends that you do not use Berkshelf to import dependencies during Chef runs. This process introduces a dependency on the external Chef Supermarket API (<https://supermarket.chef.io>). If the supermarket is unavailable when instances create in your stack, the initial Chef run may fail.

Instead, package the custom cookbooks you develop and their dependencies into a single .zip archive with the `berks package` Berkshelf command. When you execute this command in a cookbook directory, it will automatically scan any `Berksfile` and `metadata.rb` to list any dependencies, download them from their external location, and package them into a compressed .tar archive. You can upload this archive to Amazon S3 and configure it as the custom cookbook repository location for your stack.

```
berks package cookbooks.tar.gz
```

```
Cookbook(s) packaged to /Users/username/tmp/berks/cookbooks.tar.gz
```

When you package dependencies for multiple cookbooks in the parent directory of the cookbooks, create a Berksfile such as this:

```
source "https://supermarket.chef.io"
cookbook "server-app", path: "./server-app"
cookbook "server-utils", path: "./server-utils"
```

After you package the dependencies, run the berks package command from this directory to download and dependencies for your cookbooks.

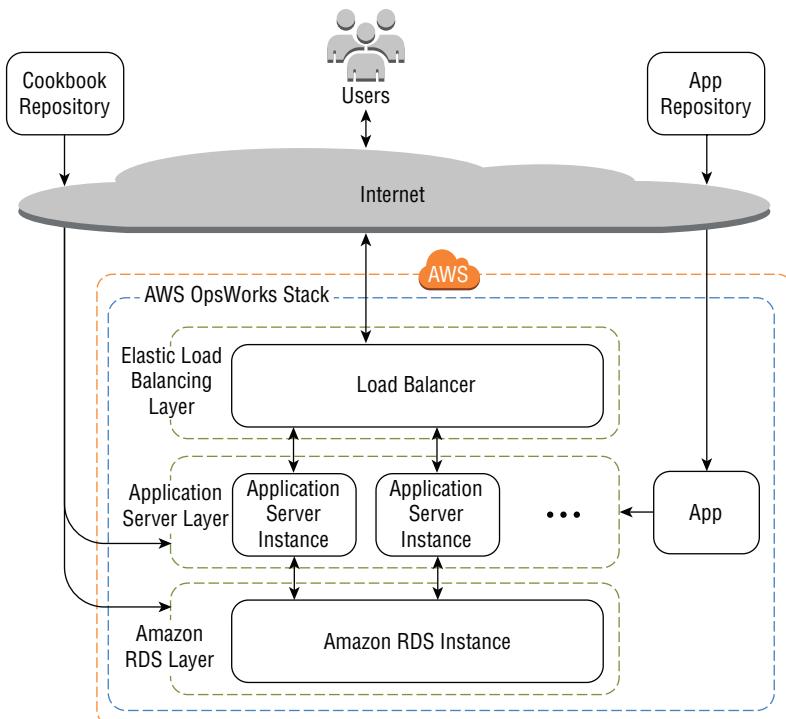
```
berks package cookbooks.tar.gz
```

## Stack

A typical workload in AWS will include systems for various purposes, such as load balancers, application servers, proxy servers, databases, and more. The set of Amazon EC2 on-premises instances, Amazon RDS, Elastic Load Balancing, and other systems make up a stack. You can organize stacks across an enterprise.

Suppose you have a single application with dev, test, and production environments. Each of these environments has a stack that enables you to separate resources to ensure stability of changes. You group resources into stacks by logical or functional purposes. The example stack in Figure 9.4 includes three layers, a cookbook repository, an application repository, and one app to deploy to the application server instances. This stack manages a full application available to users over the Internet.

**FIGURE 9.4** Example stack structure



When you create a new stack, you will have the option to set the stack's properties.

## Stack Name

*Stack Name* identifies stacks in the AWS OpsWorks Stacks console. Since this name is not unique, AWS OpsWorks assigns a Globally Unique Identifier (GUID) to the stack after you create it.

## API Endpoint Region

AWS OpsWorks associates a stack with either a global endpoint or one of multiple regional endpoints. When you create a resource in the stack, such as an instance, it is available only from the endpoint you specify when you create the stack. For example, if a stack is created with the global “classic” endpoint, any instances will be accessible only by AWS OpsWorks Stacks that use the global API endpoint in the US East (N. Virginia) region. Resources are not available across regional endpoints.

## Amazon Virtual Private Cloud

Stacks can create and manage instances in Amazon EC2 Classic or an Amazon Virtual Private Cloud (Amazon VPC). When you select an Amazon VPC, you will be able to specify in what subnets to deploy instances when they are created.

## Default Operating System

AWS OpsWorks Stacks supports many built-in Linux operating systems and Windows Server (only in Chef 12.2 stacks). If there is a custom Amazon Machine Images (AMI) you want to use, you must configure other tasks on the AMI to make it compatible with AWS OpsWorks Stacks. You must base the custom AMI off an AMI that AWS OpsWorks supports.

- The AMI must support cloud-init.
- The AMI must support the instance types you plan to launch.
- The AMI must utilize a 64-bit operating system.

## Layer

A *layer* acts as a subset of instances or resources in a stack. Layers act as groups of instances or resources based on a common function. This is especially important, as the Chef recipe code applies to a layer and all instances in a layer. A layer is the point where any configuration of nodes will be set, such as what Chef recipes to execute at each life-cycle hook. A layer can contain any one or more nodes, and a node must be a member of one or more layers. When a node is a member of multiple layers, it will run any recipes you configure for each lifecycle event for both layers in the layer and recipe order you specify.

From the point of view of a Chef Server installation, a layer is synonymous with a Chef Role. In the node object, the layer and role data are equivalent. This is primarily to ensure compatibility with open-source cookbooks that are not written specifically for AWS OpsWorks Stacks.

## Elastic Load Balancing

After a layer is created, any elastic load balancers in the same region associate with the layer. Any instances that come online in the layer will automatically register with the load balancer. The instances will also deregister from the load balancer when they go offline.

## Elastic IP Addresses

You can configure layers to assign public or elastic IP addresses to instances when they come online. For Amazon Elastic Block Store (Amazon EBS) backed instances, the IP address will remain assigned after the instance stops and starts again. For instance-store backed instances, the IP address may not be the same as the original AWS OpsWorks instance.

## Amazon EBS Volumes

Linux stacks include the option to assign one or more Amazon EBS volumes to a layer. In the process, you configure the mount point, size, Redundant Array of Independent Disks (RAID) configuration, volume type, Input/Output Operations Per Second (IOPS), and encryption settings. When new instances start in the layer, AWS OpsWorks Stacks will attempt to create an Amazon EBS volume with the configuration and attach it to the instance. Through the instance's setup lifecycle event, AWS OpsWorks Stacks runs a Chef cookbook to mount the volume to the instance. When the volumes add or remove volumes to or from a layer, only new instances will receive the configuration updates. Existing instances' volumes do not change.



Only Chef 11.10 stacks support RAID configurations.

## Amazon RDS Layer

*Amazon RDS layers* pass connection information to an existing Amazon RDS instance. When you associate an Amazon RDS instance to a stack, it is assigned to an app. This passes the connection information to the instances via the app's deploy attributes, and you can access the data within your Chef recipes with `node[:deploy][:app_name][:database]` hash.

You can associate a single Amazon RDS instance with multiple apps in the same stack. However, you cannot associate multiple Amazon RDS instances with the same app. If your application needs to connect to multiple databases, use custom JSON to include the connection information for the other database(s).

## Amazon ECS Cluster Layer

*Amazon ECS cluster layers* provide configuration management capabilities to Linux instances in your Amazon ECS cluster. You can associate a single cluster with a single stack at a time. To create this layer, you must register the cluster with the stack. After this, it will appear in the Layer type drop-down list of available clusters from which to create a layer, as shown in Figure 9.5.

**FIGURE 9.5** Creating a layer

Use the console or AWS CLI/SDK commands to create the cluster.

After the layer is created, any existing cluster instances will not import into the stack. Instead, the instances need to register with the stack as on-premises instances, or you need to create a new instance in the layer with the AWS OpsWorks Stacks console or CLI and replace them with new instances. When you create new instances, AWS OpsWorks Stacks will automatically install Docker and the Amazon ECS agent before it registers the instance with the cluster.

An instance *cannot* belong to both an Amazon ECS cluster layer and a Chef 11.10 built-in layer. However, the instance can belong to an Amazon ECS cluster layer and other custom layer(s).

### Chef 11.10 Built-in Layers

AWS OpsWorks Stacks provides several types of built-in layers for Chef 11.10 stacks.

- HAProxy layer
- MySQL layer
- AWS Flow (Ruby) layer
- Java app server layer
- Node.js app server layer
- PHP app server layer
- Rails layer
- Static web server layer
- Amazon ECS cluster layer

Each of the built-in layers provides a number of preconfigured recipes that speed up the process to deploy applications and manage underlying infrastructure. For example, the Node.js app server layer requires you to specify only one or more apps in the stack and associate it with the layer. When you use the built-in recipes, the app (or apps) automatically deploys to any instances in the layer.

Much like wrapper cookbooks in Chef, you can override built-in layers if you specify a custom attribute or template files. To do so, you can create a cookbook with the same name as the cookbook you want to override and include only the files you want to replace.

### Custom Cookbooks or Templates

The built-in apache2 recipe includes an apache2.conf.erb template file that you can override if you create a directory structure as a custom cookbook in your repository.

```
apache2
|- templates
|-- default
|--- apache2.conf.erb
```

When your instance updates its cookbooks, it will merge both the built-in layer's cookbooks with your custom ones and override the template file with your changes.

## Instances

An *instance* represents either an Amazon EC2 or on-premises instance, and the configuration AWS OpsWorks Stacks enforces upon it. You can associate instances with one or more layers, which will define the configuration to apply to the instance. AWS OpsWorks Stacks can create instances. For existing instances or on-premises servers, they can register with a stack, and you can manage them in the same manner as if you created them as AWS OpsWorks Stacks.



Some Linux distributions can register instances.

### Instance Type

AWS OpsWorks Stacks supports three instance types.

**24/7 instances** This instance type runs until you manually stop it.

**Time-based instances** Instances of this type run on a daily and weekly schedule that you configure and are useful for handling predictable changes in a load on your stack.

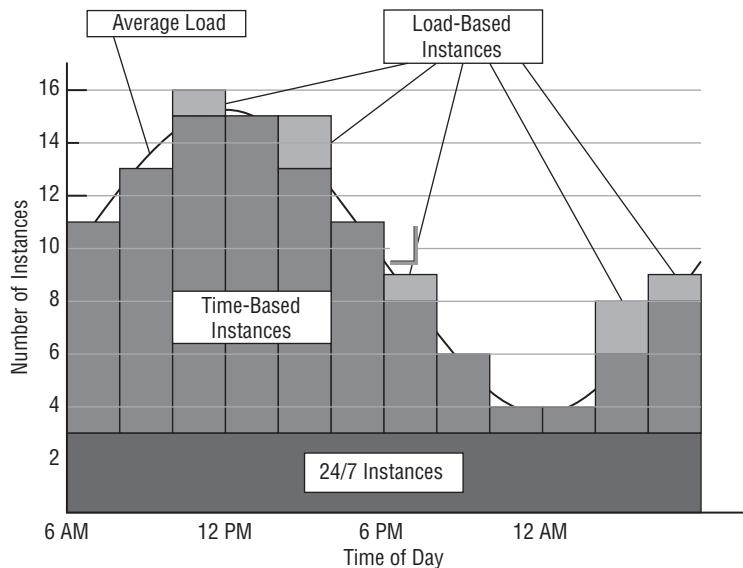
**Load-based instances** Load-based instances start and stop automatically based on metrics such as NetworkOut or CPUUtilization.

You can use time-based and load-based instances to implement automatic scaling in response to predictable or sudden changes in demand. However, unlike Amazon EC2 Auto Scaling groups, you must create time-based and load-based instances ahead of time with the AWS OpsWorks console or AWS CLI. The underlying Amazon EC2 instance will not be created until the time you specify, or the load threshold occurs, but the AWS OpsWorks instance object must exist ahead of time.

If your stack contains more than several instances, a mix of the previous instance types will provide adequate scalability in response to predictable and sudden changes in demand.

For example, if the lowest demand throughout the day in your environment requires three running instances, then it would make sense to include at least three 24/7 instances in the stack. If demand has a predictable pattern throughout the rest of the day, you can use a number of time-based instances to scale out to meet known increases in demand. To accommodate any potential changes outside the norm, you can configure additional load-based instances as well. Figure 9.6 demonstrates the use of each instance type to react dynamically to changes in request volume.

**FIGURE 9.6** Instance usage over time



### Root Device Type

When you create an Amazon EC2 instance, you have the option to choose either the instance-store or Amazon EBS backed instance types. There are several advantages and disadvantages to each type, as shown in Table 9.2.

**TABLE 9.2** Instance-Store–Backed vs. Amazon EBS Backed

Type	Advantages	Disadvantages
Instance-store-backed	Lower cost	Slower boot after initial No data persistence
Amazon EBS backed	Faster boot after initial Retain disk contents	Higher cost

You can apply more configuration settings at the layer, such as Amazon EBS volumes and elastic IP addresses.

### Instance Updates

When an instance first boots, AWS OpsWorks Stacks will automatically install any new security and package updates. However, after the initial boot, this will not occur again. This is to ensure that future updates do not affect the performance of your applications. For Linux stacks, you can initiate updates with the `Update Dependencies` command. Windows stacks do not provide any built-in means to perform updates.

As an alternative to updating instances directly, you can instead regularly launch new instances to replace old ones. As the new instances are created, they will be patched with the latest available security and operating system updates. If you would like to prevent updates entirely and manage this through a separate process, instances can be set to not install updates on startup when you create them. Additionally, this can be set at the layer level to propagate to any new instances that you add to the layer.

### Register Instances

If there are instances running in your own data center or other Amazon EC2 instances in your account (or even other accounts), you can register those instances with your stack. You can perform tasks such as user management, package updates, operating system upgrades, and application deployments on registered instances in the same manner as “native” instances.

To register an instance with a stack, you use the `aws opsworks register` AWS CLI command. The command itself will install the AWS OpsWorks agent on the instance, which is responsible for communicating with the AWS OpsWorks Stacks service endpoint to receive commands and publish information. When you register with other Amazon EC2 instances, they will need both an AWS Identity and Access Management (IAM) instance profile or IAM user credentials with access to register instances with the AWS CLI via the AWS-managed policy, `AWSOpsWorksRegisterWithCLI`.

When you register instances, you must provide a valid SSH user and private key or valid username and password. These must correspond to a Linux user on the target system (unless you call the `register` command from the target system itself). After the instance registers, it will display in the AWS OpsWorks Stacks console for assignment to one or more layers in your stack.

You can also deregister an instance from a stack if you no longer want to manage it as part of that stack. This frees it up for you to register it with a different stack or management process.

## AWS OpsWorks Agent

The *AWS OpsWorks Agent* installs on any instances that the stack registers or creates. The agent is responsible for querying the AWS OpsWorks Stacks endpoint for commands to execute on the instance, provide instance metrics to the service, provide health checks for auto healing, and update itself (if configured to do so). You can configure the stack to use a specific agent version or automatically update to the latest available version.

## Auto-Healing Instances

If you enable auto healing for a layer, instances that fail to communicate with the AWS OpsWorks service endpoint for more than 5 minutes restart automatically. You can view this in the Amazon CloudWatch Events console where `initiated_by` is set to `auto-healing`. Auto healing is enabled by default on all layers in a stack, but you can disable them at any time.

When instances are auto-healed, the exact behavior depends on the type of instance.

- For instance-store backed instances, the underlying instance terminates, and a new one is created in its place.
- Amazon EBS backed instances stop and start with the appropriate Amazon EC2 API command.

## Apps

An *app* refers to the location where you store application code and other files. This can be an Amazon S3 bucket, a Git repository, or an HTTP bundle. If you require credentials to connect to the repository, the app configuration provides them as well. The `Deploy` lifecycle event includes any apps that you configure for an instance at the layer or layers to which it corresponds.



AWS OpsWorks Stacks automatically downloads and deploys applications in built-in layers. For custom layers, you must include this functionality in your recipe code.

After you configure one or more apps for a layer, running a deployment on instances in that layer will copy the application code from the configured repository and perform any needed deployment steps. In Chef 11.10 built-in layers, this occurs automatically. For custom layers in any Chef version, the deployment process is not automatic, and you must use custom cookbooks.

To perform app updates, you first modify the app itself to point to a new version. Either the current location (Amazon S3 file, Git branch, or HTTP archive) must have the update of the new application code or the app must point to a new revision. Currently running

instances will not automatically update. Instances that create after the app updates will deploy the latest version. Any instances that stop at the time of the update will update when the instance starts again.

## Users and Permissions Management

AWS OpsWorks Stacks provides the ability to manage users at the stack level, independent of IAM permissions. This is incredibly useful to provide access to instances in a stack without giving a user actual permission to your account. Since most large organizations have strict access policies for third-party contractors, AWS OpsWorks Stacks allows you to give access to perform some stack management tasks, as well as Secure Socket Shell (SSH) or Remote Desktop Protocol (RDP) access to stack instances, and not allow nonemployees access to perform tasks within your account.



AWS OpsWorks Stacks users associate with regional endpoints and cannot be given access to stacks not in the same region. In this case, you need to import the user into the other region(s).

### Managing Permissions

There are four permission types you can apply to a user to provide stack-level permission.

**Deny** No action is allowed on the stack.

**Show** The user can view stack configuration but cannot interact with it in any way.

**Deploy** The user can view stack configuration and deploy apps.

**Manage** The user can view stack configuration, deploy apps, and manage stack configuration.

AWS OpsWorks Stacks permissions do not allow certain actions, such as to create or clone stacks. IAM permissions restrict these actions, and you must assign them to an IAM user or IAM role. If the user in question is also an IAM user, you can fine-tune the permissions levels.



You can give an IAM user the Manage permission at the stack level but deny the ability to delete layers (`opsworks:DeleteLayer`) at the IAM level. Like IAM, explicit deny will always take precedence over explicit allow.

Along with stack-level permissions, you can give AWS OpsWorks users SSH or RDP access into instances with or without administrative access. You can also configure users to manage their own SSH keys so that they can set their key once they provide access and

do not require shared key files through other means. This is also more secure than Amazon EC2 key pairs, as the keys are unique to individual users.

User permissions are set at the stack level in the AWS OpsWorks console, as shown in Figure 9.7. Here you can assign stack and instance permissions to individual user accounts.

**FIGURE 9.7** AWS OpsWorks Stacks user permissions

User Name	Permission level					Instance access	
	Deny	IAM Policies Only	Show	Deploy	Manage	SSH / RDP	sudo / admin
admin_user	<input checked="" type="radio"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>				
cli-user-test	<input checked="" type="radio"/>	<input type="checkbox"/>	<input type="checkbox"/>				
development	<input checked="" type="radio"/>	<input type="checkbox"/>	<input type="checkbox"/>				

## Managing Users

For existing IAM users for whom you would like to configure stack-level access, you can import them into AWS OpsWorks Stacks on the Users page of the AWS OpsWorks console. Once you import them, you can assign stack-level permissions. These permissions combine with current IAM policies before AWS OpsWorks evaluates them. For example, if you would like to deny access to a specific stack for an IAM user, you can import the user into AWS OpsWorks Stacks and then assign the Deny permission for that stack.

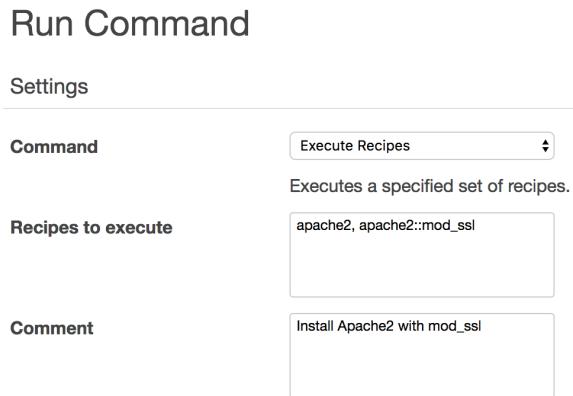


If you import an IAM user into AWS OpsWorks Stacks and then later delete that user, you must manually delete the AWS OpsWorks Stacks user as well to revoke any SSH or RDP access.

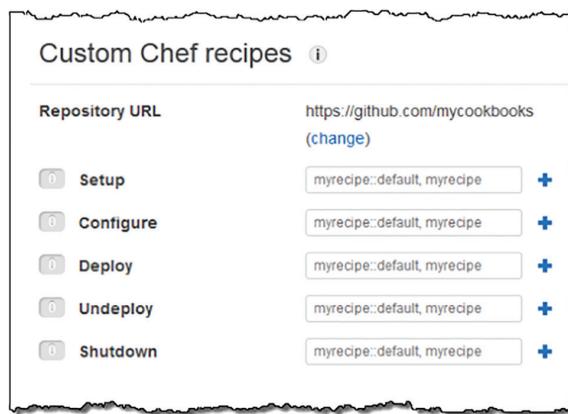
## Lifecycle Events

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

Outside of the lifecycle events, you can execute recipes manually with the Execute Recipes command in the AWS OpsWorks console (see Figure 9.8) or AWS CLI. When invoking the command, you can provide a list of recipes to execute in order.

**FIGURE 9.8** Running command recipes to execute

You can add more custom recipes to each lifecycle event in the layer's configuration. Recipes execute in the order they appear in the Custom Chef recipes lifecycle event, as shown in Figure 9.9.

**FIGURE 9.9** Custom Chef recipes for lifecycle events

## Setup

This event occurs once the instance has come online after initial creation or when the instance stops and starts. Setup automatically invokes the Deploy command after it completes successfully.

## Configure

Any time an instance in a stack comes online or goes offline, all instances in the same stack will undergo a Configure lifecycle event. This ensures that all instances in a stack are

“aware” of each other. For example, if a layer in your stack installs and configures haproxy for load balancing, any instances in the same layer will need to update to include the new node in /etc/hosts (or remove the node that went offline).

## Deploy

After an instance has come online and completes the initial Setup and Configure events, a Deploy event deploys any apps that you configure for the layer. This step can copy application code from a repository, start or refresh services, and perform other tasks to bring your application(s) online.

After an instance has run Deploy for the first time, it will never do so again automatically. This prevents untested changes from reaching production instances. After you test a feature change or bug fix, you must manually run the Deploy event with the AWS OpsWorks Stacks console or AWS CLI.

## Undeploy

The Undeploy lifecycle event runs when you delete or remove an app from a layer. You use this to perform tasks such as when you want to remove an application’s configuration or other cleanup tasks when you remove an app.

## Shutdown

Before the actual shutdown command issues to an instance, the Shutdown lifecycle event gives you the opportunity to perform tasks such as taking snapshots and copying log files to Amazon S3 for later use. If the instance’s layer also includes a load balancer, the instance deregisters after the configured connection draining time.

## Resource Management

AWS OpsWorks Stacks allows for management of other resources in your account as part of your stack, and it includes elastic IP addresses, Amazon EBS volumes, and Amazon RDS instances. You register the resources with the stack to make them available to assign them to instances or layers. If you attach resources to instances in the stack and you delete the instance, the resource remains registered with the stack until it is manually deregistered. Deregistering resources does not automatically delete them. You must delete the resource itself with the respective service console or AWS CLI command.

## Amazon EBS Volumes

Amazon EBS volumes that are not currently attached to any instances can register with a stack, and you can assign them to instances if the volume uses XFS formatting. You cannot attach volumes to running instances. To attach a volume to a running instance, you must stop it. You can move a volume between instances that are both offline.



You cannot attach Amazon EBS volumes to Windows stacks.

## Elastic IP Addresses

As with Amazon EBS volumes, elastic IP addresses that are not associated with resources in your account may be registered with the stack. You can assign an elastic IP address to an instance regardless of whether it is running or not. After an Elastic IP address is disassociated from an instance, a configure lifecycle event updates instances in the stack with the instance's new IP address.

## Amazon RDS Instances

You can register Amazon EBS instances to only one stack at a time. However, you can register a single Amazon RDS instance with multiple apps in the same stack.

## Chef 11 and Chef 12

Both Chef 11 and Chef 12 provide unique functionality differences that are important to note before you use AWS OpsWorks Stacks. Each of the major differences is outlined in this section.

The differences in this section are with respect to AWS OpsWorks Stacks as a service, and they do not include differences between Chef versions 11.10 and 12.0. Since version 11.10 has been deprecated by Chef, community support will not be as strong as for later versions.

## Separate Chef Runs

In Chef 11.10 stacks, AWS-provided cookbooks were run in the same Chef run as any custom cookbooks. The AWS cookbooks performed various tasks such as mounting Amazon EBS volumes that had been attached to the instance in the AWS OpsWorks console. However, this could result in situations where custom cookbooks had naming conflicts with those provided by AWS. You had to split this into two separate Chef runs on the instance to eliminate any potential namespace conflicts.

## Community Support

Since the deprecation of Chef 11.10, community support has gradually decreased. Any open source cookbooks on the Chef Supermarket, for example, will likely make use of Chef 12.0 functionality, removing backward compatibility for Chef 11.10 stacks.

## Built-in Layers

Chef 12.0 stacks no longer include the built-in layers as in Chef 11.10 stacks, such as the Rails layer. To implement these layers in Chef 12.0 stacks, you can still copy the built-in cookbooks from a Chef 11.10 stack and update them to be compatible with Chef 12.0. Chef 12.0 stacks still support built-in layer types from Chef 11.10 stacks.

- Amazon RDS instance layers
- Amazon ECS cluster layers

## Berkshelf

Berkshelf is no longer available for the automatic installation on Chef 12.0 instances. Instead, install Berkshelf with a custom cookbook.

## Data Bags

In lieu of custom JSON, Chef 12.0 stacks support data bags to provide better compatibility with community cookbooks. You can declare data bags in the custom JSON field of the stack, layer, and deployment configurations to provide instances in your stack for any additional data that you would like to provide. The attributes set in the data bags will no longer be available in the node object, as with Chef 11.10 stacks, but instead are available with Chef search.

### Example: Searching Data Bag Content

Here's an example of searching a data bag for a value:

```
app = search("aws_opsworks_app").first
Chef::Log.info("***** The app's short name is '#{app['shortname']}' *****")
Chef::Log.info("***** The app's URL is '#{app['app_source']['url']}' *****")
```

## Data Bags and Custom JSON

In Chef 11.10 stacks, you provide data to instances with custom JSON, which populates in the node object when Chef runs. You can access this information in your recipe code to specify configuration based on the value of custom JSON. You can specify custom JSON at the stack, layer, and deployment levels. Any data that you define at the deployment level overrides the data set at the layer or stack levels. Any data set at the layer level overrides the data set at the stack level.

### Example: Stack-Level Settings in Custom JSON

Here's a JSON example at the stack level:

```
{
  "state": "visible",
  "colors": {
    "foreground": "light-blue",
    "background": "dark-gray"
  }
}
```

Now set the custom JSON for the layer to override the stack-level settings.

```
{
  "state": "hidden",
  "colors": {
    "foreground": "light-blue",
    "background": "dark-gray"
  }
}
```

*(continued)*

*(continued)*

Next, execute the recipe on an instance; the value of node['state'] will be hidden.

```
Chef::Log.info("***** The app's initial state is '#{node['state']}' *****")
Chef::Log.info("***** The app's initial foreground color is '#{node['colors']['foreground']}' *****")
Chef::Log.info("***** The app's initial background color is '#{node['colors']['background']}' *****")
```

Custom JSON is limited to 80 KB in size. If you need to provide larger data, consider the use of Amazon S3 and retrieve files with a custom cookbook.

Chef 12.0 and 12.2 stacks use *data bags* instead of custom JSON. This provides better integration with community cookbooks that rely on data bags as the latest Chef standard to provide structured data to cookbooks. Data bags are also written in JSON at the stack, layer, and deployment levels. Any stack data originally provided in the node object on Chef 11.10 stacks is instead made available through one of several data bags during Chef runs.

### App Data Bag (aws\_opsworks\_app)

Suppose the aws\_opsworks\_app data bag provides information about any apps associated with the layer. In Chef 11.10 stacks, you can access app information in the node object.

```
Chef::Log.info ("***** The app's short name is '#{node['opsworks']['applications'].first['slug_name']}' *****")
Chef::Log.info("***** The app's URL is '#{node['deploy']['simplephpapp']['scm']['repository']}' *****")
```

Moving to data bags in Chef 12.0 and 12.2 stacks, Chef search is used to query data bag contents. The above example would instead look like:

```
app = search("aws_opsworks_app").first

Chef::Log.info("***** The app's short name is '#{app['shortname']}' *****")
Chef::Log.info("***** The app's URL is '#{app['app_source']['url']}' *****")
```

To add custom data bags to your stack, specify them in either the stack, layer, or deployment custom JSON field. To create data bags to call users with a single item, you can do the following:

```
{
  "opsworks": {
    "data_bags": {
      "users": {
        {
          "id": "nick",
          "comment": "Nick Alteen",
          "home": "/opt/alteen",
```

```
        "ssh_keys": ["123...", "456..."]  
    }  
}  
}  
}  
}  
}
```

## Monitor Instance Metrics

AWS OpsWorks Stacks provides a custom dashboard to monitor up to 13 custom metrics for each instance in the stack. The agent that runs on each instance will publish the information to the AWS OpsWorks Stacks service. If you enable the layer, system, application, and custom logs, they automatically publish to Amazon CloudWatch Logs for review without access the instance itself. You can monitor details at the stack, layer, and instance levels. The metrics that the AWS OpsWorks Agent publishes include `cpu_idle`, `memory_free`, `procs`, and others. Since these metrics are provided by the AWS OpsWorks Agent running on the instance itself, information not available to the underlying host of the instance is provided.



Windows instance monitoring provides only the standard Amazon EC2 metrics.

For each stack in your account, a dashboard displays these metrics over time. The metrics divide by layer and display over time periods that vary, as shown in Figure 9.10.

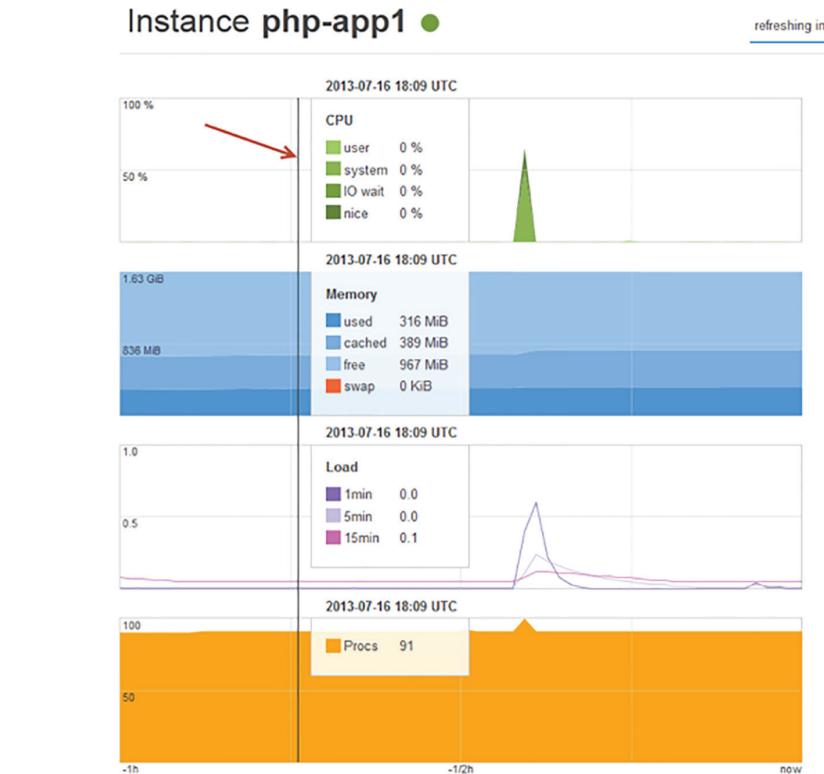
**FIGURE 9.10** Monitoring all layers in a stack



If you select a layer name in the monitoring dashboard, you can review each individual layer. In the layer's dashboard, metrics divide by individual instances. For example, if you select PHP App Server, as shown in Figure 9.10, the screen in Figure 9.11 displays.

**FIGURE 9.11** Monitoring a single layer

From the layer dashboard, you can review individual instance metrics if you select the instance name. For example, if you select php-app1, as shown in Figure 9.11, the screen in Figure 9.12 displays.

**FIGURE 9.12** Monitoring an instance

If you enable Amazon CloudWatch Logs on Linux stacks, you can configure the AWS OpsWorks Agent to send system, application, and custom logs to Amazon CloudWatch Logs. With this, alerts can be set when the logs detect specific string patterns, such as HTTP 500 responses in web servers. To publish logs, the instance profile for any instances in the layer must contain permission to push logs. To do this, you assign the `AWSOpsWorksCloudWatchLogs` managed policy to the corresponding role. Since this integration requires a later agent version, enabling it for your layer will result in all instances being upgraded to a compatible agent version (if they do not already have one).

When you stream logs to Amazon CloudWatch Logs, the log groups use the following naming convention:

```
stack_name\layer_name\chef_log_name
```

Custom logs, which you define by the file path in the layer settings, add to log groups with the naming convention.

```
/stack_name/layer_short_name/file_path_name
```

Along with CloudWatch Logs, CloudWatch Events support stacks. Any time the following types of events occur, you can invoke custom actions in response.

- Instance state change
- Command state change
- Deployment state change
- Alerts

## AWS OpsWorks Stacks Service Limits

AWS OpsWorks Stacks enforces the service limits shown in Table 9.3. These limits can be increased by submitting a request to AWS Support.

**TABLE 9.3** AWS OpsWorks Stacks Service Limits

Limit	Value
Stacks per region per account	40
Layers per stack	40
Instances per stack	40
Apps per stack	40

## Using AWS OpsWorks Stacks with AWS CodePipeline

Inside a stack, you specify a value for App to refer to a repository or archive that contains application code to deploy to one or more layers. You can use AWS CodePipeline to update an AWS OpsWorks app, which then deploys to any instances in layers you associate with this app.

To configure AWS CodePipeline to deploy to a stack, select the appropriate stack, layer, and app to update with the input artifact, as shown in Figure 9.13.

**FIGURE 9.13** Using AWS OpsWorks Stacks with AWS CodePipeline

Create pipeline

Step 1: Name  
Step 2: Source  
Step 3: Build  
**Step 4: Deploy**  
Step 5: Service Role  
Step 6: Review

**Deploy**

Choose how you deploy to instances. Choose the provider, and then provide the configuration details for that provider.

**Deployment provider\*** AWS OpsWorks Stacks

**AWS OpsWorks Stacks**

Choose one of your existing stacks.

**Stack\*** My Sample Stack (Linux)

Choose the layer that your target instances belong to.

**Layer** Node.js App Server

Choose the app that you want to update and deploy, or create a new one in AWS OpsWorks Stacks.

**App\*** Node.js Sample App

The application source that you specified for 'Node.js Sample App' in AWS OpsWorks Stacks will use a new Amazon S3 archive, and the repository URL will point to the version of the artifact that you are deploying.  
[Learn more](#)

\* Required

Cancel Previous **Next step**

## Deployment Best Practices

Since app or cookbook updates do not deploy automatically to running instances, you need a robust deployment strategy to ensure that changes complete successfully (or do not cause outages if they fail). This section details the deployment best practices recommended by AWS.

## **Rolling Deployments**

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

## **Blue/Green Deployments (Separate Stacks)**

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

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

## **Manage Databases Between Deployments**

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

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

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

# **Using Amazon Elastic Container Service to Deploy Containers**

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

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

## What Is Amazon ECS?

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

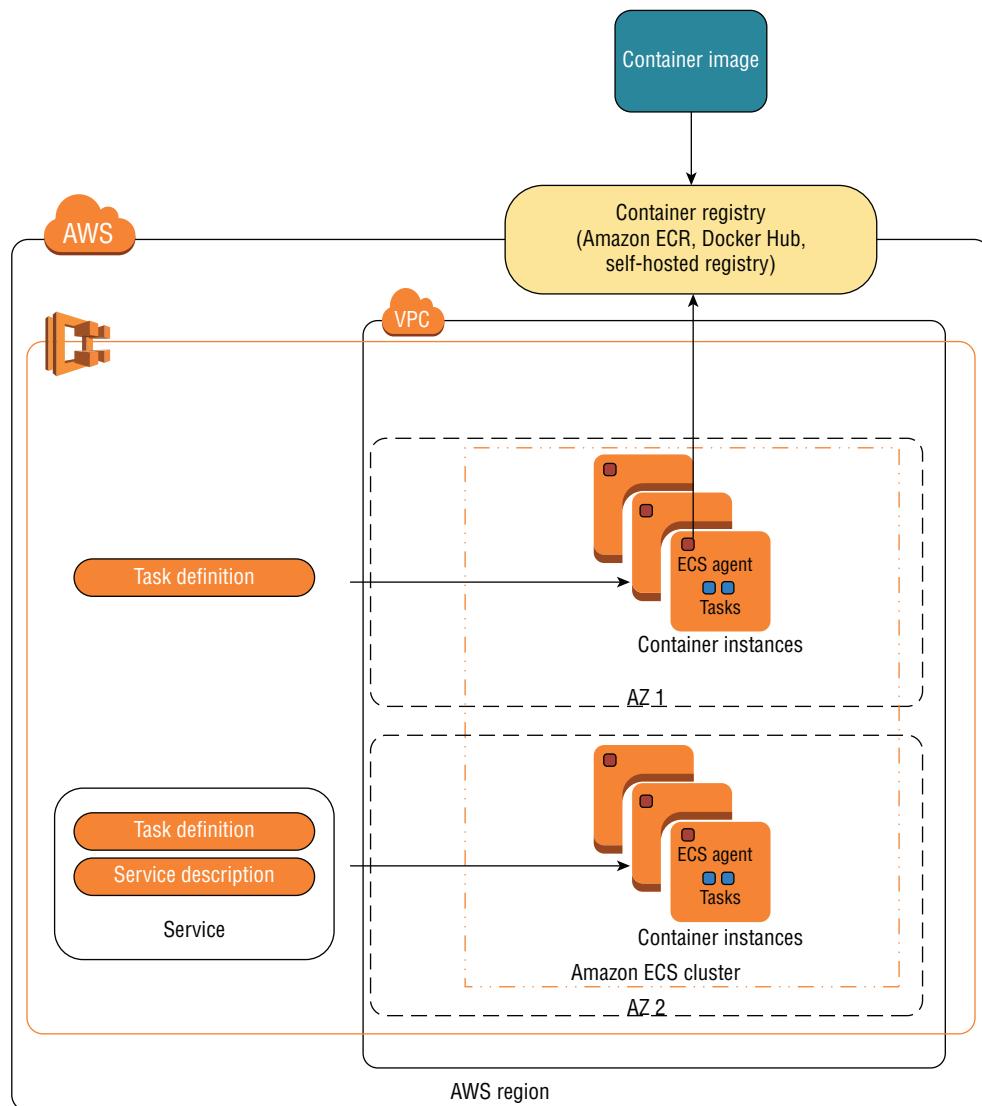
To react to changes in demands for your service or application, Amazon ECS supports Amazon EC2 Auto Scaling groups of cluster instances that allow your service to increase running container counts across multiple instances as demand increases. You can define container isolation and dependencies as part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

## Amazon ECS Concepts

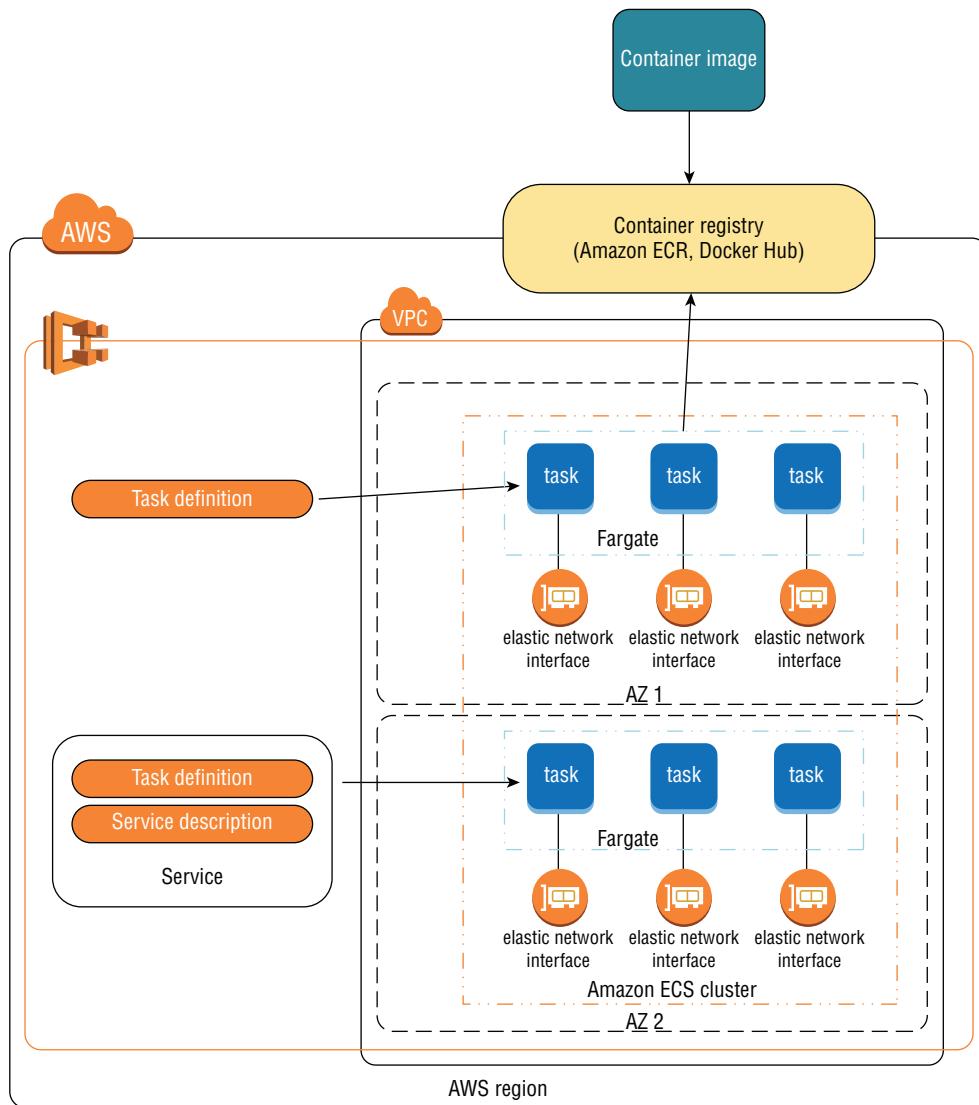
This section details Amazon ECS concepts.

### Amazon ECS Cluster

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

**FIGURE 9.14** Amazon ECS architecture

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

**FIGURE 9.15** AWS Fargate architecture

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

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

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

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

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

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



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

## AWS Fargate

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

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



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



AWS Fargate requires the awslogs driver to enable log configuration.

## Containers and Images

---



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

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

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



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

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

## Task Definition

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

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

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

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

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

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

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

### Task Definition with Two Containers

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

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

*(continued)*

(continued)

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

## Services

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

### Deployment Strategies

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

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

## Balance Loads

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

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

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

## Schedule Tasks

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

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

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

## Target Tracking Policies

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

## Step Scaling Policies

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

## Task Placement Strategies

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

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

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

## Task Placement Constraints

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

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

## Amazon ECS Service Discovery

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



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

## Private Image Repositories

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

## Amazon Elastic Container Repository

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

## Amazon ECS Container Agent

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

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

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

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



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

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

## Amazon ECS Service Limits

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

**TABLE 9.4** Amazon ECS Service Limits

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

## Using Amazon ECS with AWS CodePipeline

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

**FIGURE 9.16** Amazon ECS as a deployment provider

Create pipeline

Step 1: Name  
Step 2: Source  
Step 3: Build  
**Step 4: Deploy**  
Step 5: Service Role  
Step 6: Review

Deploy

Choose how you deploy to instances. Choose the provider, and then provide the configuration details for that provider.

Deployment provider\*

Amazon ECS ⓘ

Choose one of your existing clusters, or [create a new one in Amazon ECS](#).

Cluster name\*

Choose one of your existing services, or [create a new one in Amazon ECS](#).

Service name\*

Type the filename of your image definitions file. This is a JSON file that describes your Amazon ECS service's container name and the image and tag.

Image filename

\* Required

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

## Summary

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Exam Essentials

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Resources to Review

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

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

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

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

Managing Multi-Tiered Applications with AWS OpsWorks:

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

AWS OpsWorks Stacks:

<https://aws.amazon.com/opsworks/stacks/>

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

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

Docker on AWS:

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

What are Containers?

<https://aws.amazon.com/containers/>

Amazon Elastic Container Service (ECS):

<https://aws.amazon.com/ecs/>

# Exercises

## EXERCISE 9.1

### Launch a Sample AWS OpsWorks Stacks Environment

1. Launch the **AWS Management Console**.
  2. Select **Services > AWS OpsWorks**.
  3. Select **Add Stack**, and select **Sample stack**.
  4. Select your preferred operating system (**Linux or Windows**).
  5. Select **Add Instance**, and monitor the stack's progress until it enters the online state. This deploys the app to the stack.
  6. Copy the public IP Address, and paste it into a web browser to display the sample app.
  7. Open the instance in the **AWS OpsWorks Stacks** console, and view the log entries.
  8. Verify that the Chef run was a success and which resources deploy to the instance in the log entries.
  9. Update the recipes of the automatically created layer.
  10. Remove the deploy recipe.
  11. Add a new instance to the stack and monitor its progress.
  12. Once the instance is in the online state, view the run logs to verify that the sample website is not deployed to the instance.
- 

## EXERCISE 9.2

### Launch an Amazon ECS Cluster and Containers

1. Launch the Amazon ECS console.
  2. Create a new cluster with an Amazon EC2 container instances.
  3. Create a new task definition that launches a WordPress and MySQL container.
  4. Use the official images from **Docker Hub**:
    - a. <https://registry.hub.docker.com/wordpress/>
    - b. <https://registry.hub.docker.com/mysql/>
  5. Create a new service that launches this task definition on the cluster.
  6. Copy the public IP address, and paste it into a web browser to access WordPress on the cluster instance.
-

- 
7. Modify the service to launch two tasks.

As the second task attempts to launch, note that this will fail because of the ports configured in the task definition already being registered with the running containers.

8. Launch an additional cluster instance in your cluster.
  9. Monitor the service status to verify that the second task deploys to the new cluster instance.
- 

### EXERCISE 9.3

#### Migrate an Amazon RDS Database

1. Launch the **Amazon RDS console**.
  2. Create a new database instance.
  3. Connect to your database and create a user with a password. For example, to create a user with full privileges on MySQL, use the following command:

```
GRANT ALL PRIVILEGES ON *.* TO 'username'@'localhost' IDENTIFIED BY 'password';
```
  4. Launch the **AWS OpsWorks console**.
  5. Create two stacks (one “A” stack and one “B” stack). For ease of use, try the sample Linux stack with a Node.js app.
  6. Register the RDS database instance that you created with stack A, providing the database username and password you created.
  7. Edit the stack’s app to include **Amazon RDS** as a data source. Select the database you registered and provide the database name.
  8. Verify that you can connect to your database by creating a simple recipe to output the credentials. Specifically, try to output to the database field of the deploy attributes.
  9. Run this recipe to verify that the connection information passes to your nodes.
  10. Pass the same connection information into stack A using custom JSON.
  11. **Deregister** the database from stack A and **register** it with stack B.
  12. Perform the same tasks to verify that connection details pass to the instances in stack B.
  13. Remove the custom JSON from stack A to complete the migration.
-

**EXERCISE 9.4****Configure Auto Healing Event Notifications in AWS OpsWorks Stacks**

1. Launch the **Amazon SNS console**.
2. Create a new notification topic with your email address as a recipient.
3. Launch the **Amazon CloudWatch console**.
4. Create an Amazon CloudWatch Rule.
  - a. Edit the JSON version of the rule pattern to use:

```
{  
    "source": [ "aws.opsworks" ],  
    "detail": {  
        "initiated_by": [  
            "auto-healing"  
        ]  
    }  
}
```

5. Add the Amazon SNS topic that you created as a target.
6. Add permissions to the Amazon SNS topic so that it can be invoked by Amazon CloudWatch Events. An example policy statement is shown here. Replace the value of the Resource block with your topic Amazon Resource Name (ARN).

```
{  
    "Version": "2008-10-17",  
    "Id": "AutoHealingNotificationPolicy",  
    "Statement": [{  
        "Effect": "Allow",  
        "Principal": {  
            "Service": "events.amazonaws.com"  
        },  
        "Action": "sns:Publish",  
        "Resource": "arn:aws:sns:REGION:ACCOUNT:MyTopic"  
    }]  
}
```

7. Create a stack and add an instance. Make sure that Auto Healing is enabled on the stack.
8. Launch the instance.
9. SSH or RDP into the instance.
10. Uninstall the **AWS OpsWorks Stacks Agent**.
11. Wait until the instance is stopped and started by AWS OpsWorks Stacks. You will receive a notification shortly after this occurs.

# Review Questions

1. Which of the following AWS OpsWorks Stacks limits cannot be raised?
  - A. Maximum stacks per account, per region
  - B. Maximum layers per stack
  - C. Maximum instances per layer
  - D. Maximum apps per stack
  - E. None of the above
2. After submitting changes to your cookbook repository, you notice that executing cookbooks on your AWS OpsWorks instances does not result in any changes taking place, even though the logs show successful Chef runs.  
What could be the cause of this?
  - A. The instances are unable to connect to the cookbook repository or archive location because of networking or permissions errors.
  - B. The AWS OpsWorks Stacks agent running on the instance is enforcing cookbook caching, resulting in cached copies being used instead of the new versions.
  - C. The version of the cookbook specified in the recipe list for the lifecycle event is incorrect.
  - D. The custom cookbooks have not yet been downloaded to the instances.
3. When will an AWS OpsWorks Stacks instance register and deregister from an Elastic Load Balancing load balancer associated with the layer?
  - A. Instances are registered or deregistered manually only.
  - B. Instances will be registered when they enter an online state and are deregistered when they leave an online state.
  - C. As an administrator, you are responsible for including the registration and deregistration within your Chef recipes and assigning the recipes to the appropriate lifecycle event.
  - D. Instances are registered when they are created and not deregistered until they are terminated.
4. You have an Amazon ECS cluster that runs on a single service with one task. The cluster currently contains enough instances to support the containers you define in your task, with no additional compute resources to spare (other than those needed by the underlying OS and Docker). Currently the service is configured with a maximum in-service percentage of 100 percent and a minimum of 100 percent. When you attempt to update the service, nothing happens for an extended period of time, as the replacement task appears to be stuck as it launches.

How would you resolve this? (Select TWO.)

- A. The current configuration prevents new tasks from starting because of insufficient resources. Add enough instances to the cluster to support the additional task temporarily.
- B. The current configuration prevents new tasks from starting because of insufficient resources. Modify the configuration to have a maximum in-service percentage of 200 percent and a minimum of 0 percent.

- C. Configure the cluster to leverage an AWS Auto Scaling group and scale out additional cluster instances when CPU Utilization is over 90 percent.
  - D. Submit a new update to replace the one that appears to be failing.
5. Which party is responsible for patching and maintaining underlying clusters when you use the AWS Fargate launch type?
- A. The customer
  - B. Amazon Web Services (AWS)
  - C. Docker
  - D. Independent software vendors
6. Why should instances in a single AWS OpsWorks Stacks layer have the same functionality and purpose?
- A. Because all instances in a layer run the same recipes
  - B. To keep the console clean
  - C. To stop and start at the same time
  - D. To all run configure lifecycle events at the same time
7. Where do instances in an AWS OpsWorks Stacks stack download custom cookbooks?
- A. The Chef Server
  - B. They are included in the Amazon Machine Image (AMI).
  - C. The custom cookbook repository
  - D. Amazon Elastic Container Service (Amazon ECS)
8. How would you migrate an Amazon Relational Database Service (Amazon RDS) layer between two stacks in the same region?
- A. Supply the connection information to the second stack as custom JSON to ensure that the instances can connect. Remove the Amazon RDS layer from the first stack. Add the Amazon RDS layer to the second stack. Remove the connection custom JSON.
  - B. Add the Amazon RDS layer to the second stack and remove it from the first.
  - C. Create a new database instance, migrate data to the new instance, and associate it with the second stack using an Amazon RDS layer.
  - D. This is not possible.
9. Which AWS OpsWorks Stacks instance type would you use for predictable increases in traffic or workload for a stack?
- A. 24/7
  - B. Load-based
  - C. Time-based
  - D. On demand

- 10.** Which AWS OpsWorks Stacks instance type would you use for random, unpredictable increases in traffic or workload for a stack?
- A.** 24/7
  - B.** Load-based
  - C.** Time-based
  - D.** Spot
- 11.** What component is responsible for stopping and starting containers on an Amazon Elastic Container Service (Amazon ECS) cluster instance?
- A.** The Amazon ECS agent running on the instance
  - B.** The Amazon ECS service role
  - C.** AWS Systems Manager
  - D.** The customer
- 12.** What is Service-Oriented Architecture (SOA)?
- A.** The use of multiple AWS services to decouple infrastructure components and achieve high availability
  - B.** A software design practice where applications divide into discrete components (services) that communicate with each other in such a way that individual services do not rely on one another for their successful operation
  - C.** Involves multiple teams to develop application components with no knowledge of other teams and their components
  - D.** Leasing services from different vendors instead of doing internal development
- 13.** How many containers can a single task definition describe?
- A.** 1
  - B.** Up to 3
  - C.** Up to 5
  - D.** Up to 10
- 14.** You have a web proxy application that you would like to deploy in containers with the use of Amazon Elastic Container Service (Amazon ECS). Typically, your application binds to port 80 on the instance on which it runs. How can you use an application load balancer to run more than one proxy container on each instance in your cluster?
- A.** Do not configure the container to bind to port 80. Instead, configure Application Load Balancing (ALB) with dynamic host port mapping so that a random port is bound. The ALB will route traffic coming in on port 80 to the port on which the container is listening.
  - B.** Configure a Port Address Translation (PAT) instance in Amazon Virtual Private Cloud (Amazon VPC).
  - C.** If the container binds to a specific port, only one copy can launch per instance.
  - D.** Configure a classic load balancer to use dynamic host port mapping.

15. Which Amazon Elastic Container Service (Amazon ECS) task placement policy ensures that tasks are distributed as much as possible in a single cluster?
- A. Spread
  - B. Binpack
  - C. Random
  - D. Least Cost

# Chapter 10



AWS® Certified Developer Official Study Guide  
By Nick Alteen, Jennifer Fisher, Casey Gerena, Wes Gruver, Asim Jalil,  
Heiward Osman, Marife Pagan, Santosh Patlolla and Michael Roth  
Copyright © 2019 by Amazon Web Services, Inc.

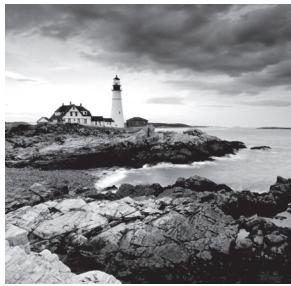
# Authentication and Authorization

---

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

## Domain 2: Security

- ✓ 2.1 Make authenticated calls to AWS services.
- ✓ 2.3 Implement application authentication and authorization.



# Introduction to Authentication and Authorization

*Authentication* is the process or action that verifies the identity of a user or process. *Authorization* is a security mechanism that determines access levels or permissions related to system resources including files, services, computer programs, data, and application features. The authentication and authorization process grants or denies user access to network resources based on the identity.

AWS *Identity and Access Management* (IAM) allows you to create identities (users, groups, or roles) and control access to various AWS services through the use of policies. IAM serves as an *identity provider* (IdP).

The following are the benefits of integrating an *existing IdP*:

- Users are no longer required to manage multiple sets of credentials.
- There are fewer credentials to administer.
- Credentials are centrally managed.
- It is easier to establish and enforce compliance standards.

As an IdP, AWS is responsible for storing identities and providing the mechanism for authentication. You can use AWS as an IdP for the following:

- AWS services
- Applications running on AWS infrastructure
- Applications running on non-AWS infrastructure, such as web or mobile applications

There are multiple benefits for using AWS as the IdP. AWS provides a managed service, eliminates single points of failure, is highly available, and can scale as needed. AWS also provides a number of tools, such as Amazon CloudWatch and AWS CloudTrail, to manage, control, and audit this service.

Using a third party to provide identity services is known as *federation*.

In this chapter, you learn the various ways to integrate existing identity providers into AWS and how to use AWS as an identity provider to control access to applications, both inside and outside the AWS infrastructure.

## Different Planes of Control

There are two different planes of access used to manage and access AWS services: a control plane and a data plane.

The *control plane* permits access to perform operations on a particular AWS instance. AWS can control access to this plane through various AWS application programming interface (AWS API) operations. The *data plane* permits access to the application running on AWS. The data plane permits access to sign in to the compute instance using *Secure Shell* (SSH) or *Remote Desktop Protocol* (RDP) and to make changes to the guest operating system or to the application itself.

The control and data planes use different paths, different protocols, and different credentials; however, for several AWS services, the control and data planes are identical. Amazon DynamoDB allows you to stop and start the compute instances (control plane) and stop and start the database (data plane) using an AWS API.

## Identity and Authorization

A discussion of federation requires a review of the concept's identity and authorization. Each of these concepts asks and answers two different questions. Identity asks and answers "Who are you?"; and authorization asks and answers "What can you do?"

AWS establishes identity in several different ways, as shown in Table 10.1.

**TABLE 10.1** AWS Identity

Name	Identifier	Credential
Root user	Email	Password
User	Email	Password
User, group, or role	Access key ID	Secret access key
API	Access	Secret access key

AWS establishes authorization by user-executed APIs. AWS controls operations and tasks through APIs. Policies are JavaScript Object Notation (JSON) documents that show attribute-value pairs. Every policy document requires a minimum of three attribute-value pairs: effect, action, and resource.

*Effect* has the API value of either ALLOW or DENY. The entity (whether a user, group, or role) is either granted the permission to execute that API or denied the permission to execute that API.

*Action* determines whether the API is allowed or denied. Actions can be determined by an individual API, a grouping of APIs for the same service using a wildcard (for example, S3:\* includes all Amazon Simple Storage Service (Amazon S3 APIs), or APIs for different services.

*Resource* determines where the API is being allowed or denied. For example, with Amazon S3, you can allow the execution of an API in a particular bucket, object, or particular group of objects (using the wildcard \*).



Though the order of the three attribute-value pairs has no impact on their execution, use the acronym EAR to remember the three attribute-value pairs: effect, action, and resource (EAR).

## Federation Defined

A federation consists of two components: identity provider and identity consumer.

Each component plays a different role in the process of federation. An *identity provider* stores identities, provides a mechanism for authentication, and provides a coarse level of authorization. An *identity consumer* stores a reference to the identity, providing authorization at a greater granularity than the identity provider.

An identity provider and an identity consumer work together to create a federation. The identity provider and the identity consumer establish a trust relationship between each other. They agree on the type of information to exchange, what information to exchange, in what format, and what security methods and measures they will use.

An identity provider answers the question “Who are you?” Because a prior trust relationship has been established between the identity provider and the identity consumer, the identity consumer trusts the answer supplied by the identity provider and grants access.

There is no expectation that there will be either a synchronization or replication of data between an identity provider and an identity consumer or that an identity provider and an identity consumer are operated by the same organization or entity.

## Federation with AWS

Federation with AWS allows for two things. First, it allows you to use AWS as an IdP to gain access to both AWS and non-AWS resources. *Amazon Cognito* is an AWS service that acts as an IdP. Second, you can use non-AWS resources like Security Assertion Markup Language (SAML) 2.0, OpenID Connect (OIDC), or Microsoft Active Directory as the IdP to facilitate single sign-on (SSO).

Federation enables you to manage access to your AWS resources centrally. With federation, you can use SSO to access your AWS accounts with credentials from your corporate directory. Federation uses open standards, such as SAML or OIDC, to exchange identity and security information between an IdP and an application.

The five mechanisms that the AWS federation can facilitate are as follows:

- Custom-built IdP
- Cross-account access
- SAML
- OIDC
- Microsoft Active Directory

## Custom Build an Identity Provider

*Custom builds* were the original method of federation within AWS, but they have since been supplanted by SAML, OIDC, and Microsoft Active Directory. With SAML, you can build a custom IdP that verifies users and their identities. Though building a custom IdP offers a high degree of customization, it is a complex process, and most customers now use standard solutions.

## Cross-Account Access

When you need to access resources across multiple AWS accounts, *cross-account access* enables you to do so by using only one set of credentials. You can grant users access to resources in company accounts without having to maintain multiple user entities, and your users do not have to remember multiple passwords. Users can access the resources they need in AWS accounts by switching AWS roles. Access is permitted by the policies attached to each role. There are two accounts in cross-account access: the account in which the user resides, or *source account*, and the account with the resources to which the user wants access, or *target account*.

The target account has an IAM role that includes two components: a permissions policy and a trust policy. The *permissions policy* controls access to AWS services and resources, while the *trust policy* specifies who can assume the role and their external ID.

The source account is given an IAM role (`AssumeRole`) with a permissions policy that allows you to assume this role. The target account issues short-term credentials to the `AssumeRole`, which allows access to AWS services and the resources you specify in this credential.

Use cross-account access when you own either the target account or the source account and require no more than coordination between the owners of the source account and the target account. Cross-account access allows users to access the AWS Management Console, AWS APIs (control plane APIs and data plane APIs), and the AWS CLI.

## Security Assertion Markup Language

*Security Assertion Markup Language* (SAML) provides federation between an IdP and a service provider (SP) when you are in an AWS account and a trust relationship has been established between the IdP and the SP. The IdP and the SP exchange metadata in an `.xml` file that contains both the certificates and attributes that form the basis of the trust relationship between the IdP and the SP.

You interact only with the IdP, and all authentication and authorization occurs between you and the IdP. Based on a successful authentication and authorization, the IdP makes an assertion to the service provider. Based on the previously established trust relationship, the service provider accepts this assertion and provides access.

Use SAML to provide access to the AWS Management Console, AWS APIs (control plane APIs and data plane APIs), and the AWS CLI. SAML can also access Amazon Cognito to control access to cloud services that exist outside AWS, such as software as a solution (SaaS) applications.

## OpenID Connect

*OpenID Connect* (OIDC) is the successor to SAML. OIDC is easier to configure than SAML and uses tokens rather than assertions to provide access. Most use cases for OIDC involve external versus internal users.

With OIDC, *OpenID provider* (OP) uses a *relying party* (RP) trust to track the service provider. OP and RP exchange metadata by focusing on the OP providing information to the RP about the location of its endpoints. The RP must register with the OP and then receive a client ID and a client secret. This exchange establishes a trust relationship between the OP and the RP.

Because you interact solitarily with the OP, all authentication and authorization occur only between you and the OP. The OP issues a token to the service provider, which accepts this token and provides access. OIDC includes three different types of tokens.

- *ID token* establishes a user's identity.
- *Access token* provides access to APIs.
- *Refresh token* allows you to acquire a new access token when the previous one expires.

Companies such as Google, Twitter, Facebook, and Amazon can also establish their own OpenID provider.

After authentication and authorization occur, you can access numerous services, including the AWS Management Console, AWS APIs, and AWS CLIs. You can use OIDC to grant access to AWS services, including Amazon Cognito, Amazon AppStream 2.0, and Amazon Redshift. You can also use OIDC to grant access to SaaS applications outside of AWS.

## Microsoft Active Directory

*Microsoft Active Directory* is the identity provider for a majority of corporations. You use the *Active Directory forest trusts* to establish trust between an Active Directory domain controller and AWS Directory Service for Microsoft Active Directory (AWS Managed Microsoft AD). For Microsoft Active Directory, the domain controller is on-premises or in the AWS Cloud.

In the Microsoft Active Directory setup, the Active Directory domain controller defines the user. However, you add users to the groups that you define in the AWS Managed Microsoft AD. Access to services depends on membership within these groups.

Use Microsoft Active Directory to provide data plane access to Amazon Elastic Compute Cloud (Amazon EC2) instances running Windows, Amazon Relational Database Service (Amazon RDS) instances running SQL Server, Amazon WorkSpaces, Amazon WorkDocs, Amazon WorkMail, and, with limitations, the AWS Management Console.

## AWS Single Sign-On

*AWS Single Sign-On* (AWS SSO) is an AWS service that manages SSO access. AWS SSO allows users to sign in to a user portal with their existing corporate credentials and access

both AWS accounts and business accounts. You can have multiple permission sets, allowing for greater granularity and control over access.

## Setting Up AWS Single Sign-On

To set up AWS SSO, do the following:

1. Enable AWS SSO.
2. Connect your directory.
3. Configure SSO to your AWS accounts.
4. Configure SSO to your cloud applications (if applicable).

## Prerequisites for AWS SSO

There are several prerequisites for using AWS SSO:

- Configure and enable all AWS Organizations features.
- Use Organizations master account credentials for the initial configuration.
- Configure a Microsoft Active Directory in the AWS Directory Service.
- Ensure that the Active Directory resides in the US-East-1 Region.

## AWS CLI Access

You can sign in to the AWS SSO user portal with your existing corporate credentials and receive all AWS CLI credentials for your AWS accounts from a central location. These AWS CLI credentials automatically expire after 60 minutes to prevent unauthorized access to AWS accounts.

## Management with AWS Organizations

AWS SSO enables management of SSO access and user permissions for your AWS accounts managed through AWS Organizations. Additional setup in the individual accounts is not required. AWS SSO automatically configures and maintains the necessary permissions in your accounts. You can assign user permissions based on common job functions and customize these permissions to meet your specific security requirements.

AWS SSO records all user portal sign-in activities in AWS CloudTrail, providing visibility into data, such as which users accessed specific accounts and applications from the user portal. AWS SSO records details, including IP address, user name, date, and time of the sign-in request. Changes made by administrators in the AWS SSO console are also recorded in CloudTrail.

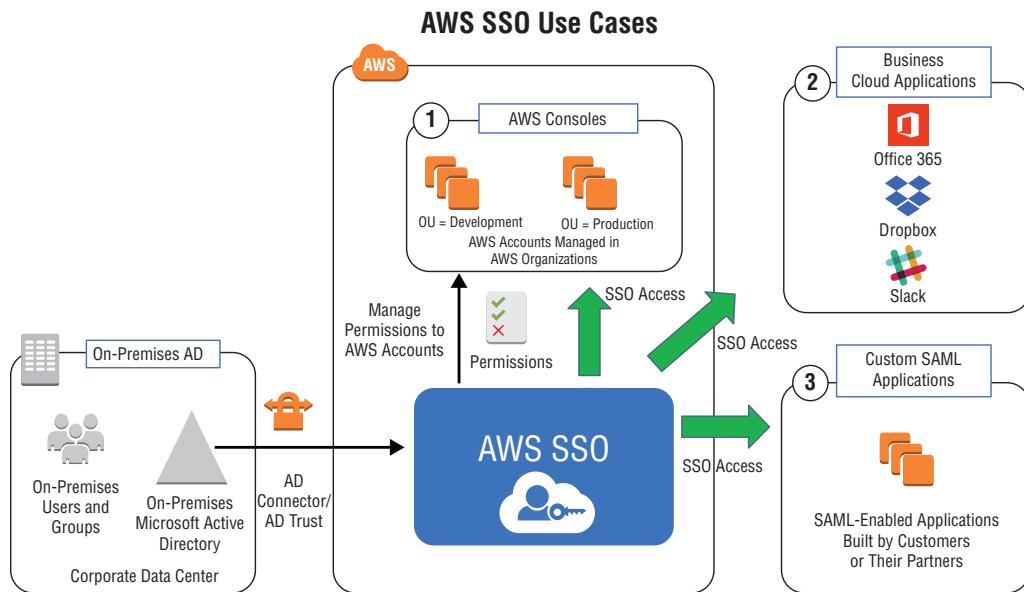
## Integration with Microsoft Active Directory

AWS SSO integrates with Microsoft Active Directory through the Directory Service, enabling you to sign in to the user portal using your Active Directory credentials. With the Active Directory integration, you can manage SSO access to your accounts and applications for users and groups in your corporate directory. For instance, when you add DevOps

Active Directory users to your production AWS group, you are granted access to your production AWS accounts automatically. This makes it easier to onboard new users and gives existing users SSO access so that they can quickly access new accounts and applications.

Figure 10.1 shows the various AWS SSO options.

**FIGURE 10.1** AWS SSO use cases model



## AWS Security Token Service

*AWS Security Token Service* (AWS STS) creates temporary security credentials and provides trusted users with those temporary security credentials. The trusted users then access AWS resources with those credentials. Temporary security credentials work similarly to long-term access key credentials, but with the following differences:

- Temporary security credentials consist of an *access key ID*, a *secret access key*, and a *security token*.
- Temporary security credentials are short-term, and you configure them to remain valid for a duration between a few minutes to several hours. After the credentials expire, AWS no longer recognizes them or allows any kind of access from API requests made with them.
- Temporary security credentials are not stored with you; they are generated dynamically and provided to you upon request. You can request new credentials before or after the temporary security credentials expire, if you still have permission to do so.

Because of these differences, temporary credentials offer the following advantages:

- You do not have to distribute or embed long-term AWS security credentials with an application.
- You can provide users access to your AWS resources without defining an AWS identity for them. Temporary credentials are the basis for AWS roles and identity federation.
- The temporary security credentials have a limited lifetime. You do not have to rotate or explicitly revoke them when the user no longer requires them.
- After temporary security credentials expire, they cannot be reused. You can specify how long the credentials are valid, up to a maximum limit.

AWS STS IdPs come from different sources, including the following:

- IAM users from another account
- Microsoft Active Directory
- Users of IdPs that are SAML 2.0-based
- Web IdPs
- Customer identity brokers



Use Amazon Cognito to authenticate for mobile applications. Amazon Cognito supports the same IdPs as AWS STS. However, it also supports unauthenticated (or guest) access and provides a means for synchronizing user data between multiple devices owned by the same user.

AWS STS supports the following APIs:

- AssumeRole
- AssumeRoleWithSAML
- AssumeRoleWithWebIdentity
- DecodeAuthorizationMessage
- GetCallerIdentity
- GetFederationToken
- GetSessionToken

**AssumeRole** This API provides a set of temporary security credentials to access AWS resources. Use AssumeRole to grant access to existing IAM users who have identities in other AWS accounts. Use this API if you need to support multi-factor authentication (MFA). By default, the maximum duration of the credentials that this API issues is 60 minutes.



The default maximum duration for AssumeRole APIs is 60 minutes. However, you can change the maximum duration to 12 hours (720 minutes) for a specific role.

**AssumeRoleWithSAML** AssumeRoleWithSAML provides a set of temporary security credentials (consisting of an *access key ID*, a *secret access key*, and a *security token*) to access AWS resources. Use this API when you are using an identity store or directory that is SAML-based, rather than having an identity from an IAM user in another AWS account. This API does not support MFA.

**AssumeRoleWithWebIdentity** AssumeRoleWithWebIdentity provides a set of temporary security credentials that you use to access AWS resources. Use this API when users have been authenticated in a mobile or web application with a web IdP, such as Amazon Cognito, Login with Amazon, Facebook, Google, or any OIDC-compatible identity provider. This API does not support MFA.

**DecodeAuthorizationMessage** DecodeAuthorizationMessage decodes additional information about the authorization status of a request from an encoded message returned in response to an AWS request. The message is encoded to prevent the requesting user from seeing details of the authorization status, which can contain privileged information.

The decoded message includes the following:

- Whether the request was denied because of an explicit deny or because of the absence of an explicit allow
- Principal who made the request
- Requested action
- Requested resource
- Values of condition keys in the context of the user's request

**GetCallerIdentity** The GetCallerIdentity API returns details about the IAM identity whose credentials call the API.

**GetFederationToken** The GetFederationToken API provides a set of temporary security credentials to access AWS resources. For example, a typical use is within a proxy application that retrieves temporary security credentials on behalf of distributed applications inside a corporate network.

The permissions for the temporary security credentials returned by GetFederationToken are a combination of the policy or policies that are attached to the IAM user, whose credentials call the GetFederationToken, and the policy passes as a parameter in the call.

Because the call for the GetFederationToken action uses the long-term security credentials of an IAM user, this call is appropriate in contexts where credentials can be safely stored. The API credentials can have a duration of up to 36 hours. This API does not support MFA.



Remember that the most restrictive policy is the one enforced. So, if you have a user who has a policy that ALLOWS access to an API, but a policy is passed as a parameter that DENIES access to that API, the result is a DENY.

**GetSessionToken** `GetSessionToken` provides a set of temporary security credentials to access AWS resources. You normally use `GetSessionToken` to enable MFA to protect programmatic calls to specific AWS APIs like Amazon EC2 `StopInstances`.

MFA-enabled IAM users call `GetSessionToken` and submit an MFA code that is associated with their MFA device. Using the temporary security credentials that return from the call, IAM users can then make programmatic calls to APIs that require MFA authentication.

## Amazon Cognito

*Amazon Cognito* is a service that allows you to manage sign-in and permissions for mobile and web applications through two services: *Amazon Cognito Sync store* and *Amazon Cognito Sync*.

With *Amazon Cognito Sync store*, you can authenticate users using third-party social identity providers or create your own identity store. With *Amazon Cognito Sync*, you can synchronize identities across multiple devices and the web.

By using Amazon Cognito, you can grant users access to AWS resources without having to embed AWS credentials into the web or mobile application. Amazon Cognito integrates with AWS STS to identify the user and give the user a consistent identity throughout the lifetime of an application, even if the device is offline or the user is accessing the application on a different device. Amazon Cognito is a managed service, providing scaling, redundancy, and high availability. You provide authentication with Amazon Cognito in one of three ways:

- Your own identity store
- Social identity providers such as Amazon or Facebook
- SAML-based identity solutions

Amazon Cognito provides a variety of mechanisms to secure the application. You can configure guest access, multi-factor authentication, and confirmation of account with Short Message Service (SMS) or email, among other mechanisms. Amazon Cognito integrates with AWS CloudTrail to track creations, deletions, and configuration changes. You can also use Amazon CloudWatch alarms to monitor for a specific activity and receive Amazon Simple Notification Service (Amazon SNS) or email notifications, if that activity occurs.

Amazon Cognito uses identity for user pools and identity pools. You use Amazon Cognito to access the AWS Management Console, AWS CLI, and AWS SDKs.

## Microsoft Active Directory as Identity Provider

Many enterprises already use Microsoft Active Directory as their identity store. Integrating Active Directory, rather than configuring a new identity store, simplifies administrative overhead. *AWS Managed Microsoft AD* provides multiple ways to use Amazon Cloud Directory and Microsoft Active Directory with other AWS services.

Directories store information about users, groups, and devices, which administrators use to manage access to information and resources. AWS Directory Service provides multiple

directory choices for customers who want to use an existing Microsoft Active Directory or *Lightweight Directory Access Protocol* (LDAP)-aware applications in the cloud. It also offers those same choices to developers who need a directory to manage users, groups, devices, and access.

There are four different ways to implement Microsoft Active Directory in an AWS infrastructure.

- Run Microsoft Active Directory on Amazon EC2 with an AWS account.
- Use *Active Directory Connector* (AD Connector) to connect AWS services with an on-premises Microsoft Active Directory.
- Create a *Simple Active Directory* (Simple AD) that provides basic Active Directory compatibility.
- Deploy AWS Managed Microsoft AD.



AWS publishes a number of Quick Start reference deployment guides, including a deployment guide for Active Directory Domain Services. For more information, see <https://docs.aws.amazon.com/quickstart/latest/active-directory-ds/youelcome.html>.

## Microsoft Active Directory on Amazon EC2 with AWS Account

AWS provides a comprehensive set of services and tools for deploying Microsoft Windows-based workloads in its secure cloud infrastructure. *Active Directory Domain Services* (AD DS) and *Domain Name System* (DNS) are core Windows services that provide the foundation for many enterprise-class Microsoft-based solutions, including Microsoft SharePoint, Microsoft Exchange, and .NET applications.

When deploying AD DS on Amazon EC2, you are responsible for deploying in a highly available configuration. You are also responsible for verifying that AD DS is backed up and configured in a fault-tolerant mode. Microsoft Active Directory deploys either as a primary or secondary domain controller, and you can choose to use Amazon Machine Images (AMI) or import your own virtual machine images.

## Active Directory Connector

*Active Directory Connector* (AD Connector) connects your existing on-premises Microsoft Active Directory with compatible AWS applications. AWS-compatible applications include Amazon WorkSpaces, Amazon QuickSight, Amazon WorkMail, and Amazon EC2 for Windows Server instances, among others. With AD Connector acting as a proxy service, you can add a service account to your Active Directory, and AD Connector eliminates the need for directory synchronization or the cost and complexity of hosting a federation infrastructure.

When you add users to AWS applications, AD Connector reads your existing Active Directory to create lists of users and groups from which to select. When users sign in to

the AWS applications, AD Connector forwards sign-in requests to your on-premises Active Directory domain controllers for authentication.



AD Connector is not compatible with Amazon Relational Database Service (Amazon RDS) SQL Server.

Management of your Active Directory does not change; you add new users and groups and update passwords using the standard Active Directory administration tools in your on-premises Active Directory. This helps you to consistently enforce your security policies, such as password expiration, password history, and account lockouts, regardless of whether users are accessing resources on-premises or on the AWS Cloud.

AD Connector enables you to access the AWS Management Console and manage AWS resources by signing in with your existing Active Directory credentials. AD Connector is not compatible with Amazon RDS for SQL Server.

You can use the AD Connector to enable MFA for your AWS application users by connecting it to your existing RADIUS-based MFA infrastructure. This provides an additional layer of security when users access AWS applications.

## Simple Active Directory

*Simple Active Directory* (Simple AD) is a Microsoft Active Directory that is compatible with AWS Directory Service and is powered by Samba 4. Simple AD is a standalone directory in the cloud, where you create and manage identities and manage access to applications. You can use many familiar Active Directory-aware applications and tools that require basic Active Directory features.

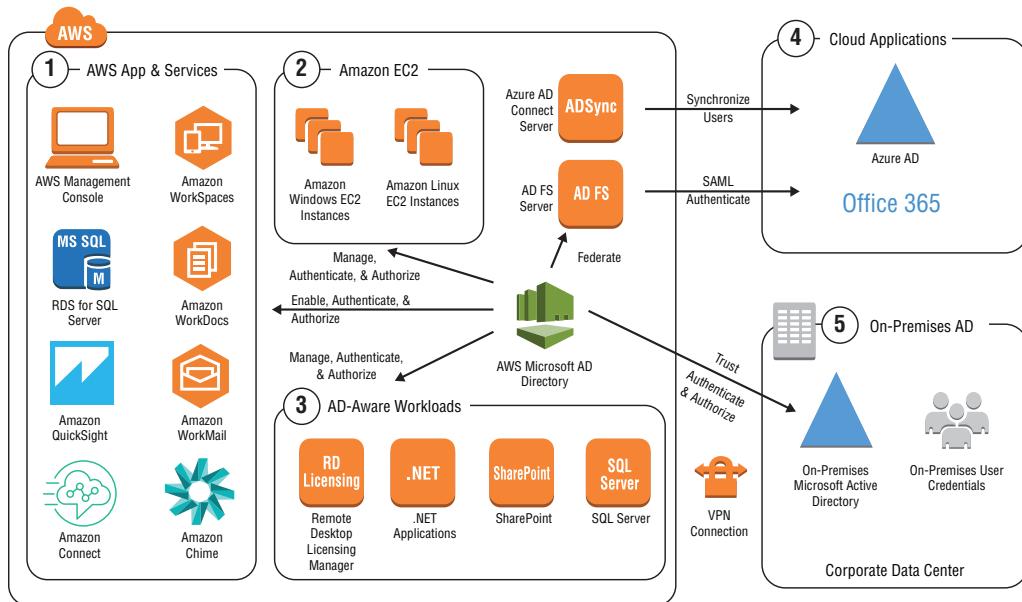
Simple AD supports basic Active Directory features such as user accounts, group memberships, memberships for a Linux domain or Windows-based Amazon EC2 instances, Kerberos-based SSO, and group policies. However, Simple AD does not support trust relationships, DNS dynamic update, schema extensions, MFA, communication over LDAPs, PowerShell Active Directory cmdlets, or Flexible Single Master Operation (FSMO) role transfer. In addition, Simple AD is not compatible with RDS SQL Server.

Simple AD is compatible with the following AWS applications: Amazon WorkSpaces, WorkDocs, Amazon QuickSight, and WorkMail. You can sign in to the AWS Management Console and manage AWS resources with Simple AD user accounts.

## AWS Managed Microsoft AD

*AWS Managed Microsoft AD* is an actual Microsoft Windows Server Active Directory, managed by AWS in the AWS Cloud. It enables you to migrate a broad range of Active Directory-aware applications to the AWS Cloud. AWS Managed Microsoft AD works with Microsoft SharePoint, Microsoft SQL Server Always-On Availability Groups, and many .NET applications.

Figure 10.2 illustrates Directory Service and its relation to AWS applications and services: Amazon EC2, Active Directory-aware workloads, cloud applications, and on-premises Active Directory.

**FIGURE 10.2** AWS Directory Service chart

Directory Service includes key features that enable you to extend your schema, manage password policies, and enable secure LDAP communications through Secure Socket Layer (SSL)/Transport Layer Security (TLS). The service is approved for applications in the AWS Cloud that are subject to the United States *Health Insurance Portability and Accountability Act* (HIPAA) or *Payment Card Industry Data Security Standard* (PCI DSS) compliance when you enable compliance for your directory.

You can add users and groups to AWS Managed Microsoft AD and administration to group policies using familiar Active Directory tools. You scale the directory by deploying additional domain controllers and improve performance by distributing requests across a larger number of domain controllers.

AWS provides monitoring, daily snapshots, and recovery as part of the service. You can connect AWS Managed Microsoft AD with a trust and use credentials to an Active Directory running on-premises. Trust relationship support includes one-way (both in and out) and two-way.

AWS Managed Microsoft AD can support AWS managed applications and services, including Amazon WorkSpaces, WorkDocs, Amazon QuickSight, Amazon Chime, Amazon Connect, and Amazon RDS for SQL Server.

## Summary

This chapter discussed the concepts of identity and authorization and how you can use AWS services to provide them. You learned that identity and authorization can operate at different planes of access—the control plane and the data plane. You also learned that

these planes differ in terms of paths used, protocols configured, services managed, and credentials deployed.

In addition, you learned about the various AWS services and where you use identity and authorization, including the following:

- AWS SSO
- AWS STS
- Amazon Cognito
- AWS Managed Microsoft AD

## Exam Essentials

**Understand what federation is.** Know the difference between federation and SSO. Understand when you would use federation and when you would use SSO.

**Understand the role of an identity provider (IdP).** Know what an IdP does, how it operates, and how it interacts with an identity consumer.

**Know the different federation services that AWS offers.** Understand which services act as IdPs, which act as identity consumers, and which act as SSO.

**Understand AWS Directory Service options.** Know the use cases for Microsoft Active Directory, Cloud Directory, and Amazon Cognito.

**Understand how policies work.** Know the structure of policies and how to apply them.

**Recognize the role of policies in controlling access to AWS resources.** Know how to use AWS services to control access to non-AWS resources and how to use non-AWS services to control access to AWS resources.

**Understand the difference between the data plane and control plane with regard to protocols and commands.** Know how AWS STS and AWS SSO work and how to implement these services.

## Resources to Review

AWS Security Token Service (AWS STS):

<https://docs.aws.amazon.com/STS/latest/APIReference>Welcome.html>

Identity Federation in the AWS Cloud:

<https://aws.amazon.com/identity/federation/>

AWS Identity and Access Management (IAM):

<https://docs.aws.amazon.com/iam/>

AWS Directory Service:

<https://aws.amazon.com/directoryservice>

Amazon Cognito:

<https://aws.amazon.com/cognito>

AWS Single Sign-On:

<https://docs.aws.amazon.com/singlesignon>

## Exercises

### EXERCISE 10.1

#### Setting Up a Simple Active Directory

In this exercise, you will set up an AWS Simple Active Directory (Simple AD). Simple AD is a standalone directory that is powered by a Samba 4 Active Directory Compatible Server. Because it's a standalone managed directory, you do not have to manage user accounts and group memberships. This is achieved through the Microsoft Active Directory.

##### Step 1: Create a Virtual Private Cloud

In this step, you will use the Amazon Virtual Private Cloud (Amazon VPC) wizard in the Amazon VPC console to create a virtual private cloud. The wizard steps create a VPC with a /16 IPv4 CIDR block and attach an internet gateway to the VPC.

1. In the AWS Management Console navigation pane, choose **VPC** and then choose **Launch VPC Wizard**.

2. On the left, select **VPC with a Single Public Subnet**, and then choose **Select**.

To communicate with an Active Directory outside of AWS, you must create the Simple AD directory in a public subnet.

3. On the next page, enter the following settings:

- a. Enter a valid, unused IP CIDR block (for example **10.40.0.0/16**).

- b. Choose a valid name (for example, **simple-ad-demo**).

- c. Choose a valid subnet (for example, **10.40.1.0/24**)

- d. Choose an Availability Zone in which to create the subnet. (Record your selection; you will need this information for the next step.)

4. Choose **Create VPC**.

You have launched a VPC that has a public subnet, an internet gateway attached to it, and the necessary route table and security group configurations to allow traffic to flow between the subnet and the internet gateway. However, because Simple AD is a highly available service, you must create a second public subnet on this VPC.

5. Navigate to the VPC dashboard and choose **Subnets**.

6. Choose **Create Subnet**.
7. On the next page, enter the following settings:
  - a. Choose a name tag.
  - b. Choose the VPC that you created in the previous steps.
  - c. Choose an Availability Zone that is different from the one selected in the previous steps.
  - d. Choose a valid subnet.
8. Choose **Create VPC**.

You have created a VPC that has two public subnets. When you create a Simple AD directory, each node is located in a different Availability Zone.

#### Step 2: Create Your Simple AD Instance in AWS

Create your AWS Managed Microsoft AD directory using the AWS Management Console.

1. In the AWS Directory Service console navigation pane, choose **Directories** and then choose **Set up directory**.
2. On the **Select directory type** page, select **AWS Simple AD** and then choose **Next**.
3. On the **Enter directory information** page, provide the following and then choose **Next**.
  - a. Directory size (Small or Large).
  - b. Directory DNS name.
  - c. (Optional) Directory NetBIOS name (CORP, for example). If one is not provided, a name is created by default.
  - d. An administrator password, which must be 8–24 characters in length. It must also contain at least one character from three of the following four categories: uppercase letters, lowercase letters, numbers, or nonalphanumeric characters.
  - e. (Optional) Description of the directory. This is useful in tracking your services within AWS.
4. On the **VPC and subnets** page, provide the following information, and then choose **Next**.
  - a. For **VPC**, choose the VPC you created earlier (simple-ad-demo).
  - b. Under **Subnets**, choose the two subnets you created for the domain controllers.
5. Review your settings and choose **Create directory**.

It takes 5–10 minutes to create your directory. You may need to refresh the page. When the directory creation is complete, the Status value changes to *Active*.

---

(continued)

**EXERCISE 10.1 (continued)****Step 3: Management and Maintenance of Simple AD Directory in AWS**

When the status changes to *Active*, the AWS Managed Microsoft AD directory is ready to do the following:

- Manage the AWS applications and services available to users
  - Perform various maintenance activities, such as creating Amazon Simple Notification Service (Amazon SNS) email or text messages to inform you of changes in status to your directory, performing a point-in-time backup (snapshot) of your directory, and modifying the schema of your AWS Managed AD directory
- 

**EXERCISE 10.2****Setting Up an AWS Managed Microsoft AD**

In this exercise, you will set up an AWS Managed Microsoft AD. Because this is an Active Directory managed by AWS, you do not have to consider the size or type of compute instances that will be running on this Active Directory. You will, however, have to choose the Amazon VPC that this service will run on.

This service is designed for high availability, so two domain controllers are created. Therefore, two corresponding subnets are used.

To simplify the installation, you will first create the necessary VPC, and then you will create the AWS Managed Microsoft AD.

**Step 1: Create a Virtual Private Cloud**

Create your Amazon VPC by using the AWS Management Console.

1. In the AWS Services console navigation pane, choose **VPC** and then choose **Your VPCs**.
  2. Note the VPCs used (to avoid address conflict), and choose **Create VPC**.
  3. In **VPC name**, enter, **My Directory Service** and specify an IPv4 CIDR block.  
Choose a CIDR block that is not currently used (for example, 10.30.0.0/16).
  4. For **Hardware tenancy**, select **Default**, and choose **Create**.
  5. After the VPC is created, return to the VPC dashboard and select **Subnets**.
  6. Choose **Create subnet**.
-

7. In the **Create Subnet** dialog box, enter a name tag, choose the VPC you just created, select an Availability Zone, and specify an IPv4 CIDR block within that VPC (for example, 10.30.1.0/24).
8. Choose **Create**.
9. Repeat steps 6–8 to create a second subnet, making sure to choose a different Availability Zone and a different subnet other than the ones specified in step 7.

### Step 2: Create Your AWS Managed Microsoft AD Directory in AWS

Create your AWS Managed Microsoft AD directory by using the AWS Management Console.

1. In the AWS Directory Service console navigation pane, choose **Directories** and then choose **Set up directory**.
2. On the **Select directory type** page, choose **AWS Managed Microsoft AD** and choose **Next**.
3. On the **Enter directory information** page, provide the following information and then choose **Next**.
4. For **Edition**, you can select either **Standard Edition** or **Enterprise Edition**. For this exercise, select **Standard Edition**.

For more information about editions, see AWS Directory Service for Microsoft Active Directory at [https://docs.aws.amazon.com/directoryservice/latest/admin-guide/what\\_is.html#microsoftad](https://docs.aws.amazon.com/directoryservice/latest/admin-guide/what_is.html#microsoftad).

5. For **Directory DNS name**, enter **corp.example.com**.
6. For **Directory NetBIOS name**, enter **corp**.
7. For **Directory description**, enter **AWS DS Managed**.
8. For **Admin password**, enter the password that you want to use for this account.

This admin account is automatically created during the directory creation process. The password cannot include the word *admin*. The directory administrator password is case-sensitive, and it must be 8–64 characters in length. It must also contain at least one character from three of the following four categories:

- Lowercase letters (a–z)
- Uppercase letters (A–Z)
- Numbers (0–9)
- Nonalphanumeric characters (~!@#\$%^&\*\_+=`|\`(){})[]:;'"<>,.?/)

9. In **Confirm Password**, enter the password again.

---

(continued)

**EXERCISE 10.2 (continued)**

10. On the **Choose VPC and subnets** page, provide the following information and then choose **Next**.
- For **VPC**, choose the option that begins with AWS-DS-VPC and ends with (10.0.0.0/16).
  - For **Subnets**, choose the 10.0.128.0/20 and 10.0.144.0/20 public subnets.
  - On the **Review & create** page, review the directory information and make any necessary changes. When the information is correct, choose **Create directory**.

Creating the directory takes 20–40 minutes. After the directory is created, the Status value changes to *Active*.

**Step 3: Management and Maintenance of AWS Managed Microsoft AD**

After AWS Managed Microsoft AD is set up, you are able to perform the following maintenance and management operations:

- Manage the AWS applications and services available to users.
- Share directories, see the Availability Zone and subnet of existing controllers, and add additional controllers.
- Create trust relationships, establish IP routing, enable log forwarding, and use multi-factor authentication.
- Create Amazon SNS email or text messages to inform you of changes in status to your directory, perform a point-in time backup (snapshot) of your directory, and modify the schema of your AWS Managed AD directory.

**EXERCISE 10.3****Setting Up an Amazon Cloud Directory**

In this exercise, you will set up an Amazon Cloud Directory. Cloud Directory is a highly available multitenant directory-based store where AWS is responsible for scaling. AWS manages the directory infrastructure, while the administrators focus on building the directories and the applications that use those directories.

**Step 1: Create a Schema**

You'll first create a schema (which defines objects in a directory) and then assign that schema to a directory. A single schema can be assigned to multiple directories, and a directory can have multiple schemas assigned to it (though typically it does not).

1. In the AWS Services console navigation pane, under **Security, Identity & Compliance**, choose **Directory Service > Schemas**.
2. To create a custom schema based on an existing one, in the table listing the schemas, select the schema named **person**.
3. Choose **Actions**.
4. Choose **Download schema**.
5. In the location where you downloaded the schema, rename the file to **test-person**.
6. On the **Schemas** page, choose **Upload new schema**.
7. Select test-person and choose **Upload**.
8. To prevent modifications to the schema, choose **schema test-person**.
9. Choose **Actions**.
10. In **Major Version**, enter the identifier **1**, and choose **Publish**.

You are returned to the Schemas page. You have two versions of the test-person schema: one schema version shows versions and is listed under State as Published; the other schema version does not show versions and is listed under State as Development.

You have successfully created a schema that you will use to create a directory.

### Step 2: Create a Directory

Before you can create a directory in Cloud Directory, Directory Service requires that you first apply a schema to it. A directory cannot be created without a schema and typically has one schema applied to it.

Create a directory that uses the schema you created in step 1.

1. In the AWS Directory Service console navigation pane, under **Security, Identity & Compliance**, choose **Directory Service > Directories**.
2. Choose **Set up Cloud Directory**.
3. Under **Choose a schema to apply to your new directory**, in **Cloud Directory name**, enter **test-cloud-directory**.
4. Choose **Custom schema**.
5. Select the custom schema named test-person with the Status of Published, and then choose **Next**.
6. Review the directory information and make the necessary changes. When the information is correct, choose **Create**.

You have successfully created a Cloud Directory. You can modify and delete the directory, including the schema associated with the directory.

---

**EXERCISE 10.4****Setting Up Amazon Cognito**

In this exercise, you will set up Amazon Cognito, which is the service that provides authentication, authorization, and user management for web and mobile applications.

The two main components of Amazon Cognito are user pools and identity pools. User pools is a user directory that provides sign-up and sign-in services. Identity pools are used to provide access to other AWS services.

You can use identity pools and user pools separately or together. In this exercise, you will set up a user pool.

1. In the AWS Directory Services console navigation pane, under Security, Identity & Compliance, choose **Cognito** and then choose **Manage User Pools**.
2. Provide a name for your user pool. Enter **admin-group**.
3. Specify how a user signs in. In this example, select **user name**, and then choose **Next step**.
4. Retain the default settings and choose **Next step**.

You can set MFA as optional or required. After a user pool is configured, you cannot change the MFA setting. Amazon Cognito uses Amazon SNS to send SMS messages. If MFA is enabled, you must assign a role with the correct policy to send SMS messages.

5. Retain the default settings and choose **Next step**.
6. On the **Attributes** page, retain the default settings for email customization, and choose **Next step**.
7. To manage the AWS infrastructure, apply tags. Enter the following information, and then choose **Next step**:
  - a. In **Tag Key**, enter **user**
  - b. In **Tag Value**, enter **admin-user**.

8. Select **No** and choose **Next step**.

Amazon Cognito can detect and retain your user's device. This step enables you to configure that capability. In this example, however, you will select **No**. Choose **Next step**.

9. Retain the default settings and choose **Next step**.

You can configure how client applications gain access to the user pool. In this exercise, no access is granted.

10. Retain the default settings and choose **Next step**.

You can configure AWS Lambda functions that can be triggered during the Amazon Cognito operation. For this exercise, you will not configure any Lambda functions.

11. On the **Review** page, review your configurations, and choose **Create pool**.

You have successfully created a user pool in Amazon Cognito.

# Review Questions

1. You need to grant a user, who is outside your AWS account, access to an object in an Amazon Simple Storage Service (Amazon S3) bucket. Which is the best way to provide access?
  - A. Create a role and assign that role to the user.
  - B. Create a user ID within Identity and Access Management (IAM) and assign the user ID a policy that allows access.
  - C. Create a new AWS account, assign that user to the account, and then give the account cross-account access.
  - D. Have the user create a user ID using a third-party identity provider (IdP), and based on that user ID, assign a policy that permits access.
2. Which of the following is the purpose of an identity provider (IdP)?
  - A. To control access to applications
  - B. To control access to the AWS infrastructure
  - C. To minimize the opportunity to assign the incorrect policy
  - D. To answer the question “Who are you?”
3. Which of the following is the best way to minimize misuse of AWS credentials?
  - A. Set up multi-factor authentication (MFA).
  - B. Embed the credentials in the bastion host and control access to the bastion host.
  - C. Put a condition on all of your policies that allows execution only from your corporate IP range.
  - D. Make sure that you have a limited number of credentials and limit the number of people that can use them.
4. Which of the following is not a valid identity provider (IdP) for Amazon Cognito?
  - A. Google
  - B. Microsoft Active Directory
  - C. Your own identity store
  - D. A Security Assertion Markup Language (SAML) 1.0-based IdP
5. Which of the following is one benefit of using AWS as an identity provider (IdP) to access non-AWS resources?
  - A. AWS cannot be used as an IdP for non-AWS services.
  - B. Using AWS as an IdP allows you to use Amazon CloudWatch to monitor activity.
  - C. Using AWS as an IdP allows you to use AWS CloudTrail to audit who is using the service.
  - D. Using AWS as an IdP allows you to assign policies to non-AWS resources.

6. Which of the following are benefits from using the Active Directory Connector (AD Connector)? (Select TWO.)
  - A. Easy setup
  - B. Ability to connect to multiple Active Directory domains with a single connection
  - C. Ability to configure changes to Active Directory on your existing Active Directory console
  - D. Ability to support authentication to non-AWS services
7. Which of the following is a prerequisite for using AWS Single Sign-On (AWS SSO)?
  - A. Set up AWS Organizations and enable all features.
  - B. Make sure that your identity provider (IdP) is Security Assertion Markup Language (SAML) 2.0 compatible.
  - C. Deploy AWS Simple Active Directory (Simple AD).
  - D. Deploy Amazon Cognito.
8. AWS Security Token Service (AWS STS) supports a number of different tokens.  
Which token would you use to establish a longer-term session?
  - A. AssumeRole
  - B. GetUserToken
  - C. GetFederationToken
  - D. GetSessionToken
9. Which of the following is not a service that AWS Managed Microsoft AD provides?
  - A. Daily snapshots
  - B. Ability to manage the Amazon Elastic Compute Cloud (Amazon EC2) instances that AWS Managed Microsoft AD is running on
  - C. Monitoring
  - D. Ability to sync with on-premises Active Directory
10. You are using an existing RADIUS-based multi-factor authentication (MFA) infrastructure.  
Which AWS service is your best choice?
  - A. Active Directory Connector (AD Connector)
  - B. AWS Managed Microsoft AD
  - C. Simple Active Directory (Simple AD)
  - D. No AWS service would be suitable.

# Chapter 11



AWS® Certified Developer Official Study Guide  
By Nick Alteen, Jennifer Fisher, Casey Gerena, Wes Gruver, Asim Jalil,  
Heiward Osman, Marife Pagan, Santosh Patlolla and Michael Roth  
Copyright © 2019 by Amazon Web Services, Inc.

# Refactor to Microservices

---

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

## Domain 4: Refactoring

- ✓ 4.1 Optimize application to best use AWS services and features.

Content may include the following:

- Amazon Simple Queue Service (Amazon SQS) message queue service
- Amazon Simple Notification Service (Amazon SNS) producer/consumer (publisher/subscriber) messaging and mobile notifications web
- Amazon Kinesis Data Streams real-time ingestion and real-time analytics
- Replacement of Amazon Kinesis Data Firehose with the CoDA service for data ingestion
- Process and analysis of Amazon Kinesis Data Analytics data with standard structured query language (SQL)
- Process and detection of content patterns in Amazon Kinesis Video Streams
- Publishing messages when Amazon DynamoDB tables change
- Using AWS IoT Device Management to manage IoT devices throughout their lifecycle

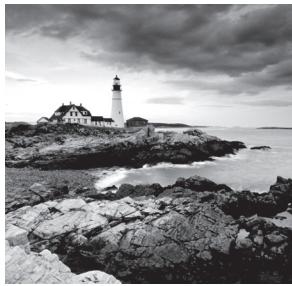


- Amazon MQ message broker service for Apache ActiveMQ
- Using AWS Step Functions to develop, launch, and monitor the progress of workflows

## **Domain 5: Monitoring and Troubleshooting**

Content may include the following:

- Troubleshooting dead-letter queue

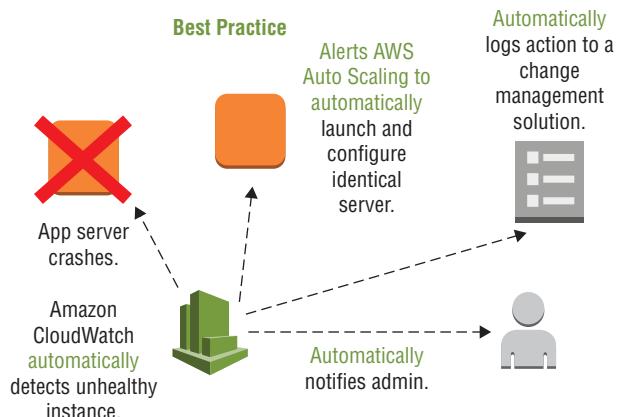


## Introduction to Refactor to Microservices

As applications grow, they become harder to manage and maintain. Application components are tightly coupled with each other, and the failure of one component can cause the failure of the whole application.

*Microservices* architecture is a method to design and build software applications as a suite of modular services, each performing a specific functional task, which deploy and access application components via well-defined standard application programming interfaces (APIs). Where possible, you automate the provisioning, termination, and configuration of resources. A best-practice scenario is shown in Figure 11.1. In this case, if an application fails, Amazon CloudWatch automatically detects the unhealthy instance and alerts AWS Auto Scaling to launch and configure an identical server, notifies the administrator, and logs the action to your change management solution.

**FIGURE 11.1** Microservices in action



*Containers* are software-defined execution environments that you can rapidly provision and independently deploy in server and serverless environments. Microservices that run in containers take portability and interoperability to a new level because the services function the same on-premises as they do in any cloud that supports containers. Independence and modularity also provide opportunities to design for elastic scalability and operational resilience.

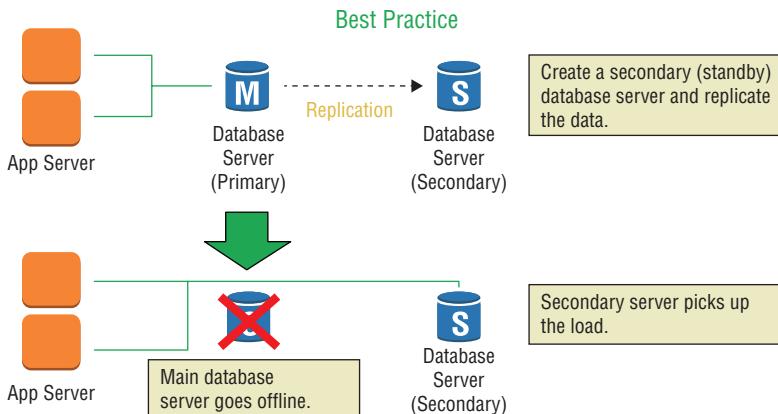
To *refactor to microservices* is to separate the application components into separate microservices so that each microservice has its own data store, scales independently, and deploys on its own infrastructure. Refactoring includes rewriting and decoupling applications, re-architecting a solution, and determining whether you will perform a complete refactor (lift and shift—all in) or only a partial refactor (lift and shift—hybrid).

To refactor to microservices requires a *message infrastructure* so that the microservices can communicate with each other. Message queues communicate between applications. AWS provides the message infrastructure that enables you to build microservice architectures without the need to spend the time and effort for a connective infrastructure.

A serverless solution is provisioned at the time of need. You can store static web assets externally, such as in an Amazon S3 bucket, and user authentication and user state storage are handled by managed AWS offerings and services.

You can further safeguard your application against latency because of failure if you avoid a single point of failure, as shown in Figure 11.2.

**FIGURE 11.2** Avoiding single points of failure



This section describes the different services AWS provides to enable the building of microservice architectures. The certification exam objectives for refactoring to microservices include the following:

- Optimizing an application to best use AWS offerings, services, and features
- Migrating existing application code to run on AWS

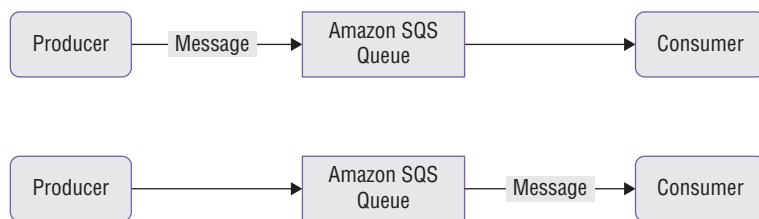
# Amazon Simple Queue Service

*Message-oriented middleware* (MoM) supports messaging types in which the messages that are produced (producers) can broadcast and publish to multiple message consumers, also known as *message subscribers*.

*Amazon Simple Queue Service* (Amazon SQS) is a fully managed message queuing service that makes it easy to decouple and scale microservices, distributed systems, and serverless applications to assist in event-driven solutions. Amazon SQS both moves data between distributed application components and helps you to decouple these components. Amazon SQS is the best option for cloud-designed applications that need unlimited scalability, capacity, throughput, and high availability. *Amazon SQS temporarily stores messages from a message producer while they wait for a message consumer to process the message.*

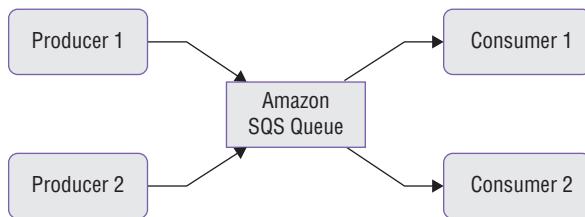
With the use of Amazon SQS, application components send messages to each other and do not have to wait for a response, as shown in Figure 11.3.

**FIGURE 11.3** Amazon Simple Queue Service (Amazon SQS) flow



The *producer* is the component that *sends* the message. The *consumer* is the component that *pulls* the message off the queue. The queue passively stores messages and does not notify you of new messages. When you poll the *Amazon SQS queue*, the queue *responds* with messages that it includes, as shown in Figure 11.4.

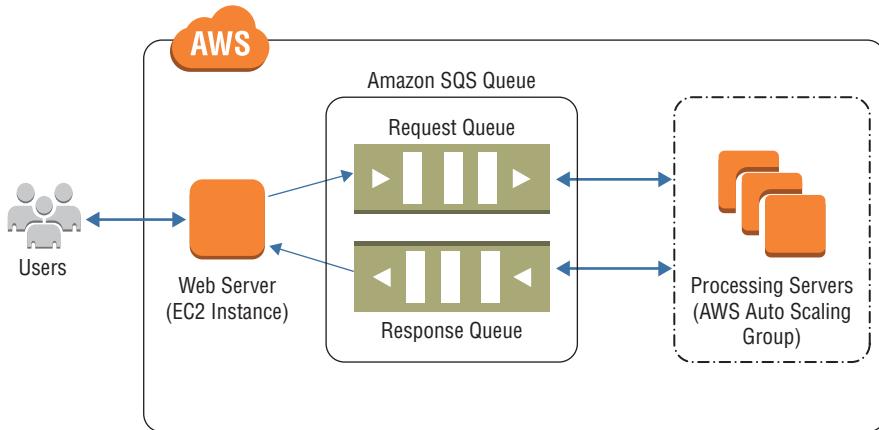
**FIGURE 11.4** Amazon SQS queue



With Amazon SQS, multiple producers can write messages, and multiple consumers can process the messages. One of the consumers processes each message, and when a consumer processes a message, they remove it from the queue. That message is no longer available for other consumers. If the amount of work on the queue exceeds the capacity for a single consumer, you can add more consumers to help the process.

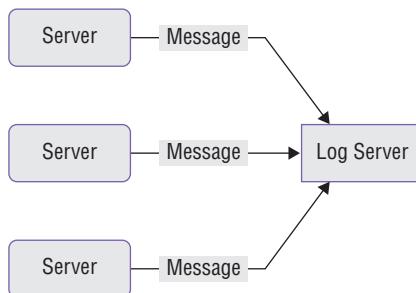
Figure 11.5 illustrates the way that the Amazon SQS queue interacts with both Amazon EC2 and the process servers.

**FIGURE 11.5** Amazon Simple Queue Service



As shown in Figure 11.6, a sign-in service run on a single *log server* is dependent on the reliability of the log server to send and receive messages. If the log server experiences any issues, the sign-in service can go offline.

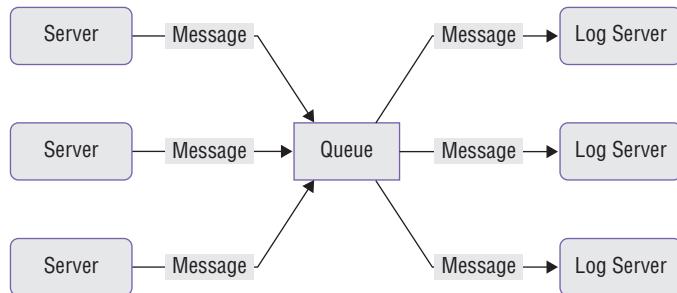
**FIGURE 11.6** Log server



#### Use Amazon SQS Queue to Alleviate Log Server Failures

If you replace the log server with an Amazon SQS queue with multiple log servers, you can remove this point of failure.

As the other servers in your application send their sign-in messages to the queue, the sign-in server can pull messages off the queue and process them, as shown in Figure 11.7.

**FIGURE 11.7** Amazon SQS queue

There are several benefits to using the Amazon SQS queue:

- If you need to take a sign-in server offline for maintenance, the service does not interrupt. The messages remain in the queue until the sign-in server comes back online.
- If the number of messages grows, you can *scale* your sign-in service and add more servers.
- Amazon SQS automatically scales to handle an increase in incoming messages.
- Messages remain in order and deliver only one message.
- Messages can be sent to the dead-letter queue.
- Messages have a visibility timeout, a message retention period, and a receive-message wait time.
- Messages can have a long polling interval or a short polling interval (default).

The Amazon SQS is a *distributed cluster of servers*. There is no limit on the number of producers that can write to the queue, and there is no limit on the number of messages that the queue can store.



Amazon SQS is a Payment Card Industry Data Security Standard (PCI DSS) service.

## Amazon SQS Parameters

An Amazon SQS message has three basic states:

1. Sent to a queue by a producer
2. Received from the queue by a consumer
3. Deleted from the queue

A message is *stored* after it is sent to a queue by a producer but not yet received from the queue by a consumer (that is, between states 1 and 2). There is no limit to the number of stored messages. A message is considered to be *in-flight* after it is received from a queue by a consumer but not yet deleted from the queue (that is, between states 2 and 3). There is a limit to the number of in-flight messages.

Limits that apply to in-flight messages are unrelated to the unlimited number of stored messages.

For most *standard queues* (depending on queue traffic and message backlog), there can be a maximum of approximately 120,000 in-flight messages (received from a queue by a consumer but not yet deleted from the queue). If you reach this limit, Amazon SQS returns the *OverLimit* error message. To avoid reaching the limit, delete messages from the queue after they are processed. You can also increase the number of queues if you file an AWS Support request.

For *first-in, first-out* (FIFO) queues, there can be a maximum of 20,000 in-flight messages (received from a queue by a consumer but not yet deleted from the queue). If you reach this limit, Amazon SQS returns no error messages.

## ReceiveMessage

The `ReceiveMessage` action waits for a message to arrive. Valid values are integers from 0 to 20 seconds, with the default value of 0.

### Long Polling

*Long polling* helps reduce the cost of Amazon SQS by eliminating the number of empty responses (when there are no messages available for a `ReceiveMessage` request) and false empty responses (when messages are available but are not included in a response).

To ensure optimal message processing, do the following:

- Set the `ReceiveMessage` wait time to 20 seconds, which is the default and the maximum value. If 20 seconds is too long for your application, set a shorter `ReceiveMessage` wait time (1 second minimum). You might have to modify your Amazon SQS client either to enable longer requests or to use a shorter wait time for long polling.
- If you implement long polling for multiple queues, use one thread for each queue instead of a single thread for all queues. This enables your application to process the messages in each of the queues as they become available.

## VisibilityTimeout

The `VisibilityTimeout` action is the duration (in seconds) that the received messages are hidden from subsequent retrieve requests after being retrieved by a `ReceiveMessage` request. *The default VisibilityTimeout for a message is 30 seconds. The minimum is 0 seconds. The maximum is 12 hours.*

How you set the `VisibilityTimeout` depends on how long it takes your application to process and delete a message. To ensure that there is sufficient time to process messages, use one of the following strategies:

- If you know (reasonably estimate) how long it takes to process a message, extend the message's `VisibilityTimeout` to the maximum time it takes to process and delete the message.
- If you do not know how long it takes to process a message, create a heartbeat for your consumer process: specify the initial `VisibilityTimeout` (for example, 2 minutes) and then—as long as your consumer still works on the message—keep extending the `VisibilityTimeout` by 2 minutes every minute.



To extend the `VisibilityTimeout` action for longer than 12 hours, consider using AWS Step Functions.

For example, if your application requires 10 seconds to process a message and you set `VisibilityTimeout` to 15 minutes, you must wait for a relatively long time to attempt to process the message again if the previous processing attempt fails. Alternatively, if your application requires 10 seconds to process a message but you set `VisibilityTimeout` to only 2 seconds, a duplicate message is received by another consumer while the original consumer is still working on the original message.

## WaitTimeSeconds

`WaitTimeSeconds` is the duration (in seconds) for which the call waits for a message to arrive in the queue before returning. If a message is available, the call returns sooner than `WaitTimeSeconds`. If no messages are available and the wait time expires, the call returns successfully with an empty list of messages.

## ReceiveMessageWaitTimeSeconds

`ReceiveMessageWaitTimeSeconds` is the length of time, in seconds, for which a `ReceiveMessage` action waits for a message to arrive. Valid values are integers from 0 to 20 (seconds), with the default value equal to 0.

## ChangeMessageVisibility

`ChangeMessageVisibility` changes the visibility timeout of a message in a queue to a new value. The default `VisibilityTimeout` setting for a message is 30 seconds. The minimum is 0 seconds. The maximum is 12 hours.



If you attempt to set `VisibilityTimeout` to a value greater than the maximum time left, Amazon SQS returns an error. Amazon SQS doesn't automatically recalculate and increase the timeout to the maximum remaining time.

Unlike with a queue, when you change the `VisibilityTimeout` value for a specific message, the `TimeoutValue` action applies immediately but is not saved in memory for that message. If you do not delete a message after it is received, the next time the message is received, the `VisibilityTimeout` setting for the message reverts to the original `TimeoutValue` setting and not to the value of the `ChangeMessageVisibility` action.

For example, suppose that you have a message with a `VisibilityTimeout` setting of 5 minutes. After 3 minutes, you call `ChangeMessageVisibility` with a timeout of 10 minutes. You can continue to call `ChangeMessageVisibility` to extend the `VisibilityTimeout` to the maximum allowed time. If you try to extend the `VisibilityTimeout` beyond the maximum, your request is rejected.

## DelaySeconds

DelaySeconds is the length of time, in seconds, that a specific message will be delayed. Valid values are 0–900, with a maximum of 15 minutes. Messages with a positive DelaySeconds value become available for processing after the delay period is finished. If you do not specify a value, the default value for the queue applies.



When you set FifoQueue, you cannot set DelaySeconds per message. You can set this parameter only on a queue level.

## MessageRetentionPeriod

MessageRetentionPeriod is the length of time, in seconds, that Amazon SQS retains a message. It is an integer representing seconds, from 60 (1 minute) to 1,209,600 (14 days). Changes made to the MessageRetentionPeriod attribute can take up to 15 minutes to take effect.

## DeleteMessage

DeleteMessage deletes the specified message from the specified queue. To select the message to delete, use the ReceiptHandle value of the message (not the MessageId that you receive when you send the message). Amazon SQS can delete a message from a queue even if a VisibilityTimeout setting causes the message to be locked by another consumer. Amazon SQS automatically deletes messages kept in a queue longer than the retention period configured for the queue.



Refer to Table 11.5 to view the differences between the Amazon Simple Notification Service (Amazon SNS) and Amazon SQS event-driven solutions.

## Dead-Letter Queue

Amazon SQS supports *dead-letter queues*, which other queues (*source queues*) can target for messages that cannot process (be consumed) successfully. Dead-letter queues are useful when you debug your application or message system because the queues let you isolate problematic messages to determine why their process did not succeed.

Sometimes messages do not process because of a variety of possible issues, such as erroneous conditions within the producer or consumer application or an unexpected state change that causes an issue with your application code. For example, if a user places a web order with a particular product ID but the product ID is deleted, the web store's code fails and displays an error, and the message with the order request is sent to a dead-letter queue.

Occasionally, producers and consumers might fail to interpret aspects of the protocol that they use to communicate, causing message corruption or loss. Also, the consumer's hardware errors might corrupt message payload.

If the consumer of the source queue fails to process a message in the number of times you specify, the *redrive policy* (`RedrivePolicy`) specifies the source queue, the dead-letter queue, and the conditions under which Amazon SQS moves messages from the former to the latter. When the `ReceiveCount` value for a message exceeds the `maxReceiveCount` value for a queue, Amazon SQS moves the message to a dead-letter queue. For example, if the source queue has a redrive policy with `maxReceiveCount` set to 5 and the consumer of the source queue receives a message five times and it does not delete, Amazon SQS moves the message to the dead-letter queue.

To specify a dead-letter queue, you can use the AWS Management Console or the AWS SDK for Java for each queue that sends messages to a dead-letter queue. Multiple queues can target a single dead-letter queue. The dead-letter queue uses the `CreateQueue` or `SetQueueAttributes` action.

*Use the same AWS account to create the dead-letter queue and the other queues that send messages to the dead-letter queue. Also, dead-letter queues must reside in the same region as the other queues that use the dead-letter queue.* For example, if you create a queue in the US East (Ohio) Region, and you want to use a dead-letter queue with that queue, the second queue must also be in the US East (Ohio) Region.

The expiration of a message is based on its original enqueue timestamp. When a message moves to a dead-letter queue, the enqueue timestamp does not change. For example, if a message spends one day in the original queue before it moves to a dead-letter queue and the retention period of the dead-letter queue is set to 5 days, the message is deleted from the dead-letter queue after 3 days. *Thus, AWS recommends that you set the retention period of a dead-letter queue to be longer than the retention period of the original queue.*

## Benefits of Dead-Letter Queues

The main task of a dead-letter queue is to handle message failure. Use a dead-letter queue to set aside and isolate messages that cannot be processed correctly to determine why their processes failed. The dead-letter queue enables you to do the following:

- Configure an alarm for any messages delivered to a dead-letter queue.
- Examine logs for exceptions that might have caused messages to be delivered to a dead-letter queue.
- Analyze the contents of messages delivered to a dead-letter queue to diagnose software or the producer's or consumer's hardware issues.
- Determine whether you have given your consumer sufficient time to process messages.

## Standard Queue Message Failures

*Standard queues* continue to process messages until the expiration of the retention period. This ensures continuous processing of messages, which minimizes the chances of your queue being blocked by messages that cannot process. It also ensures fast recovery for your queue.

*Amazon SQS standard queues* work by using *scalability* and *throughput*. To achieve this, they trade off two qualities:

- Order is not guaranteed.
- Messages can appear twice.

In a system that processes thousands of messages and in which you have a large number of messages that the consumer repeatedly fails to acknowledge and delete, standard queues may increase costs and place an extra load on the hardware. Instead of trying to process messages that fail until they expire, move them to a dead-letter queue after a few process attempts.



Standard queues support a high number of in-flight messages. If the majority of your messages cannot be consumed and are not sent to a dead-letter queue, your rate of processing valid messages can slow down. Thus, to maintain the efficiency of your queue, you must ensure that your application handles message processing correctly.

## Dead-Letter Queue First-In, First-Out Message Queues

Amazon SQS uses FIFO message queues that place the messages in the queue in the order that you receive them. The first messages that you receive display first in the queue. *Message groups* also follow this order so that when you publish messages to different message groups, each message group preserves the messages' internal order.

FIFO queues support 3,000 operations (read, write, and delete) per second with batching and support 300 operations per second without batching.

Amazon SQS standard queues use scalability and throughput, unlike Amazon SQS FIFO queues. To achieve this, they trade off two qualities:

- Order is not guaranteed.
- Messages can appear twice.

If the removal of either or both of these two constraints is important, use Amazon SQS FIFO queues. *Amazon SQS FIFO queues provide order within message groups, and they delete any duplicate messages that occur within 5-minute intervals.*

FIFO queues ensure single processing by consuming messages in sequence from a message group. Thus, although the consumer can continue to retrieve ordered messages from another message group, the first message group remains unavailable until the message that is blocking the queue processes successfully.



FIFO queues support a lower number of in-flight messages. To ensure that your FIFO queue does not get blocked by a message, you must make sure that your application handles message processing correctly.

## When to Use a Dead-Letter Queue

Use dead-letter queues with Amazon SQS standard queues when your application does not depend on the order of messages. Dead-letter queues help you troubleshoot incorrect message transmission operations.



The dead-letter queue of a *FIFO queue* must also be a FIFO queue. Similarly, the dead-letter queue of a *standard queue* must also be a standard queue.



Even when you use dead-letter queues, continue to monitor your queues, and retry to send messages that fail for transient reasons.

Do use dead-letter queues to decrease the number of messages and to reduce the possibility that you expose your system to messages that you can receive but cannot process.

Do not use a dead-letter queue with standard queues when you want to retry the transmission of a message indefinitely. For example, do not use a dead-letter queue if your program must wait for a dependent process to become active or available.

Do not use a dead-letter queue with a FIFO queue if you do not want to break the exact order of messages or operations.

## Troubleshooting Dead-Letter Queues

In some cases, Amazon SQS dead-letter queues might not behave as you expect. This section gives an overview of common issues and shows how to resolve them.

### Viewing Messages Using the AWS Management Console Causes Messages to Be Moved to a Dead-Letter Queue

Amazon SQS counts a message you view in the AWS Management Console against the queue's redrive policy. As a result, if you view a message in the console the number of times you specify in the queue's redrive policy, the message moves to the queue's dead-letter queue.

To adjust this behavior, do the following:

- Increase the *Maximum Receives* setting for the corresponding queue's redrive policy.
- Avoid viewing the corresponding queue's messages in the AWS Management Console.

### The Number of Messages Sent and Number of Messages Received for a Dead-Letter Queue Do Not Match

If you send a message to a dead-letter queue manually, the `NumberOfMessagesSent` metric counts it. However, if a message is sent to a dead-letter queue because of a failed process attempt, the metric does not count it. Thus, the values of `NumberOfMessagesSent` and `NumberOfMessagesReceived` can be different.

## Amazon SQS Attributes, Dead-Letter Queue Settings, and Server-Side Encryption Settings

Table 11.1, Table 11.2, and Table 11.3 provide all the details of the Amazon SQS message attributes, DLQ settings, and server-side encryption (SSE) settings.

**TABLE 11.1** Amazon SQS Message Attributes

Attribute	Default	Meaning
Default Visibility Timeout	30 seconds	How long a message is hidden while it is processed. Maximum limit is 12 hours.
Message Retention Period	4 days	How long a queue retains a message before deleting it.
Maximum Message Size	256-KB text	Maximum size of a message with 10 items maximum.
Delivery Delay	0 seconds	How long to delay before publishing the message to the queue.
Receive Message Wait Time	0 seconds	Maximum time consumer receives call waits for new messages.



### Large Messages

To send a message larger than 256 KB, use Amazon SQS to save the file in Amazon Simple Storage Service (Amazon S3) and then send a link to the file on Amazon SQS.

**TABLE 11.2** Dead-Letter Queue Settings

Setting	Meaning
Use Redrive Policy	Send messages to the dead-letter queue if consumers keep failing to process it.
Dead-Letter Queue	Name of dead-letter queue.
Maximum Receives	Maximum number of times a message is received before it is sent to the dead-letter queue.

**TABLE 11.3** Server-Side Encryption (SSE) Settings

Setting	Meaning
Use SSE	Amazon SQS encrypts all messages sent to this queue.
AWS Key Management Service (AWS KMS) Customer Master Key	The AWS KMS master key that generates the data keys.
Data Key Reuse Period	Length of time to reuse a data key before a new one regenerates.

## Monitoring Amazon SQS Queues Using Amazon CloudWatch

*Amazon CloudWatch* monitors your AWS resources and the applications you run on AWS in real time. You can use CloudWatch to collect and track metrics, which are variables that you can measure for your resources and applications.

CloudWatch alarms send notifications or automatically make changes to the resources you monitor based on rules that you define, for example, when a message is sent to the dead-letter queue.

If you must pass messages to other users, create an Amazon SQS queue, subscribe all the administrators to this queue, and then configure *Amazon CloudWatch Events* to send a message on a daily cron schedule into the Amazon SQS queue.

CloudWatch provides a reliable, scalable, and flexible monitoring solution with no need to set up, manage, and scale your own monitoring systems and infrastructure. You may also use Amazon CloudWatch Logs to monitor, store, and access your log files from Amazon EC2 instances, AWS CloudTrail, or other sources.

The AWS/Events namespace includes the DeadLetterInvocations metric, as shown in Table 11.4. The DeadLetterInvocations metric uses Count as the unit, so Sum and SampleCount are the most useful statistics.

**TABLE 11.4** Amazon CloudWatch Dead-Letter Queue

Metric	Description
DeadLetterInvocations	Measures the number of times a rule's target is not invoked in response to an event. This includes invocations that would result in triggering the same rule again, causing an infinite loop. Valid Dimensions: RuleName Units: Count

# Amazon Simple Notification Service

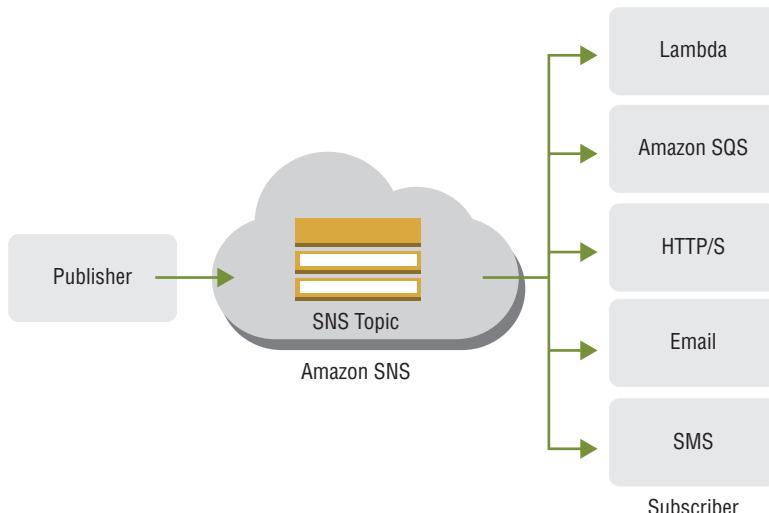
*Amazon Simple Notification Service* (Amazon SNS) is a flexible, fully managed *producer/consumer* (publisher/subscriber) messaging and mobile notifications web service that coordinates the delivery of messages to subscribing endpoints and clients. Amazon SNS coordinates and manages the delivery or sending of messages to subscriber endpoints or clients to assist in event-driven solutions.

Amazon SNS is based on the publish-subscribe model, and it allows the message producer to send a message to a *topic* that has multiple subscribers that choose to receive the same message. The message is delivered to multiple subscribers, which can then consume the message to trigger subsequent processes. A *topic* allows multiple receivers of the message to subscribe dynamically for identical copies of the same notification.

With Amazon SNS, you can easily set up, operate, and reliably send notifications to all your endpoints at any scale. You can also send messages to a large number of subscribers, including distributed systems and services and mobile devices. *By default, Amazon SNS offers 10 million subscriptions per topic and 100,000 topics per account.* To request a higher limit, contact AWS Support.

Amazon SNS enables you to send notifications from the cloud, and it allows applications to publish messages that are immediately delivered to a subscriber, as shown in Figure 11.8.

**FIGURE 11.8** Amazon SNS



There are two types of *clients* in Amazon SNS: *producers* (publishers) and *consumers* (subscribers).

Producers communicate asynchronously with subscribers by producing and sending a message to a topic, which, in the context of Amazon SNS, is a logical access point and communication channel. Subscribers, such as web servers, email addresses, Amazon SQS

queues, and AWS Lambda functions, consume or receive the message or notification over one of the supported protocols, such as Amazon SQS, HTTPS, email, Short Message Service (SMS), and AWS Lambda, when the consumer subscribes to the topic.

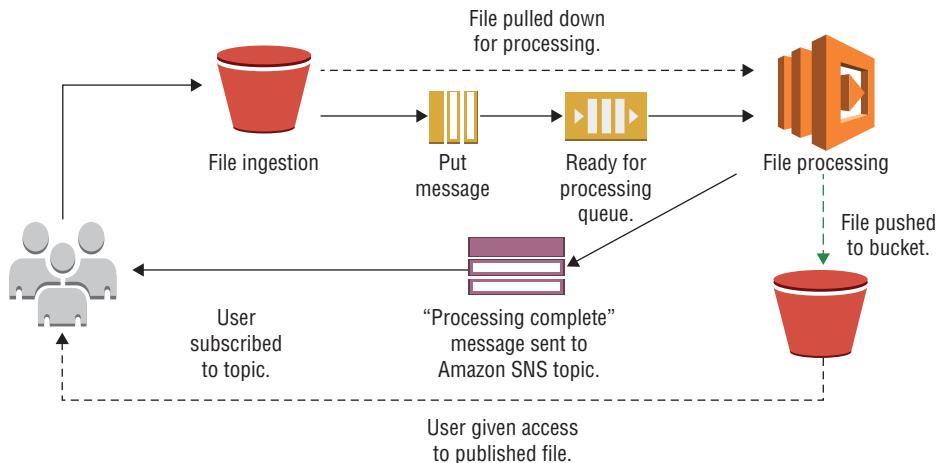
The sequence of operations in Amazon SNS includes the following:

1. The administrator creates a topic.
2. Users subscribe to the topic by using email addresses, SMS numbers, Amazon SQS queues, and other endpoints.
3. The administrator publishes a message on the topic.
4. The subscribers to the topic receive the message that was published.

If a user subscribes to the topic after a message was published, the user will *not* receive the message. A subscriber receives messages that are published only after they have subscribed to the topic. The topics do not buffer messages.

You can use Amazon SNS to produce a single message to multiple subscribers, as shown in Figure 11.9.

**FIGURE 11.9** Amazon SNS workflow



For example, when a cryptocurrency price fluctuates, you must update the *dashboard* to indicate the new price and update the *value* of the portfolio to reflect the new price. All users who subscribed to your cryptocurrency topic then receive a notification on the new prices.

Amazon SNS supports the following *endpoints*:

- AWS Lambda
- Amazon SQS
- HTTP and HTTPS
- Email
- SMS
- Mobile PushRecords

Amazon SNS retries sending messages for HTTPS endpoints as a REST call to these endpoints. You can configure the number of retries and the delay between them.

## Features and Functionality

Amazon SNS topic names have a limit of 256 characters. Topic names must be unique within an AWS account and can include alphanumeric characters plus hyphens (-) and underscores (\_). After you delete a topic, you can reuse the topic name. When a topic is created, Amazon SNS assigns a unique Amazon Resource Name (ARN) to the topic, which includes the service name (SNS), AWS Region, AWS ID of the user, and topic name. *The ARN returns as part of the API call to create the topic. Whenever a producer or consumer needs to perform any action on the topic, you reference the unique topic ARN.*

For example, Amazon SNS clients use the ARN address to identify the right topic.

```
aws sns publish --topic-arn topic-arn --message "message" --message-attributes '{"store":{"DataType":"String","StringValue":"example_corp"}}'
```

This is the ARN for a topic named `mytopic` that you create with the account ID 123456789012 and host in the US East Region:

```
arn:aws:sns:us-east-1:1234567890123456:mytopic
```

Do *not* attempt to build the topic ARN from its separate components—topics should use the name the API calls to create the topic returns.

## Amazon SNS APIs

Amazon SNS provides a set of simple APIs to enable event notifications for topic owners, consumers, and producers.

### Owner Operations

These are the owner operations:

**CreateTopic:** Creates a new topic.

**DeleteTopic:** Deletes a previously created topic.

**ListTopics:** Lists topics owned by a particular user (AWS account ID).

**ListSubscriptionsByTopic:** Lists subscriptions for a particular topic. It allows a topic owner to see the list of all subscribers actively registered to a topic.

**ListSubscriptions:** Allows a user to get a list of all of their active subscriptions (to one or more topics).

**SetTopicAttributes:** Sets/modifies topic attributes, including setting and modifying producer/consumer permissions, transports supported, and so on.

**GetTopicAttributes:** Gets/views existing attributes of a topic.

**AddPermission:** Grants access to selected users for the specified actions.

**RemovePermission:** Removes permissions for selected users for the specified actions.

## Subscriber Operations

These are the subscriber operations:

- **Subscribe:** Registers a new subscription on a particular topic, which will generate a confirmation message from Amazon SNS.
- **ConfirmSubscription:** Responds to a subscription confirmation message, confirming the subscription request to receive notifications from the subscribed topic.
- **UnSubscribe:** Cancels a previously registered subscription.
- **ListSubscriptions:** Lists subscriptions owned by a particular user (AWS account ID).

## Clean Up

After you create a topic, subscribe to it, and publish a message to the topic. You unsubscribe from the topics and delete them to clean up your environment from the Amazon SNS console.

The subscription is deleted unless it is a pending subscription, meaning that it has not yet been confirmed. *You cannot delete a pending subscription, but if it remains pending for 3 days, Amazon SNS automatically deletes it.*

## Transport Protocols

Amazon SNS supports notifications over multiple transport protocols. You can select transports as part of the subscription requests.

- **HTTP, HTTPS:** Subscribers specify a URL as part of the subscription registration; notifications are delivered through an HTTP POST to the specified URL.
- **Email, Email-JSON:** Messages are sent to registered addresses as email. Email-JSON sends notifications as a JSON object, while Email sends text-based email.
- **Amazon SQS:** Users specify an Amazon SQS standard queue as the endpoint. Amazon SNS enqueues a notification message to the specified queue (which subscribers can then process with Amazon SQS APIs, such as `ReceiveMessage` and `DeleteMessage`). Amazon SQS does not support FIFO queues.
- **SMS:** Messages are sent to registered phone numbers as AWS SMS text messages.

## Amazon SNS Mobile Push Notifications

With Amazon SNS, you can send push notification messages directly to apps on mobile devices. Push notification messages sent to a mobile endpoint can appear in the mobile app as message alerts, badge updates, or even sound alerts.

You send push notification messages to both mobile devices and desktops with the following push notification services:

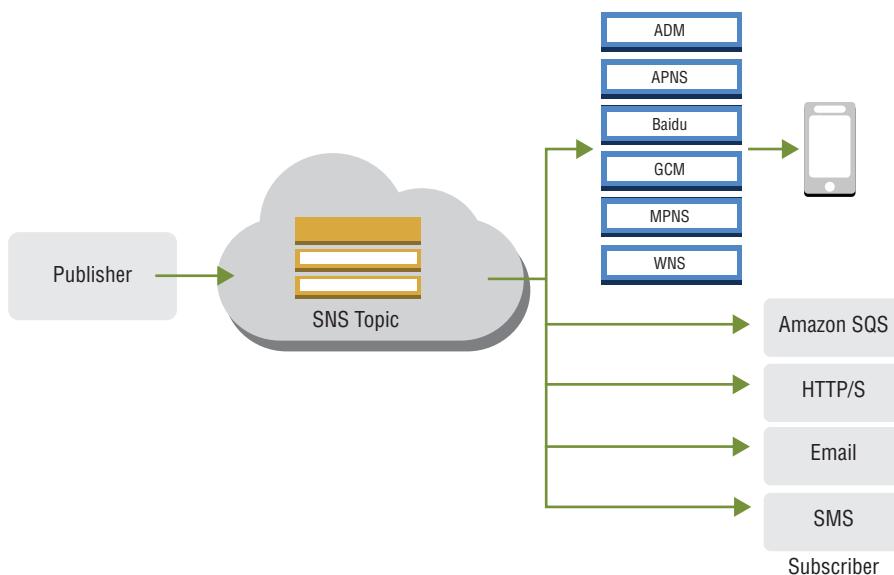
- Amazon Device Messaging (ADM)
- Apple Push Notification Service (APNS) for both iOS and macOS
- Baidu Cloud Push (Baidu)

- Google Cloud Messaging for Android (GCM)
- Microsoft Push Notification Service for Windows Phone (MPNS)
- Windows Push Notification Services (WNS)

Push notification services, such as APNS and GCM, maintain a connection with each app and mobile device registered to use their service. When an app and mobile device are registered, the push notification service returns a device token. Amazon SNS uses the device token to create a mobile endpoint to which it can send direct push notification messages. For Amazon SNS to communicate with the different push notification services, you submit your push notification service credentials to Amazon SNS.

You can also use Amazon SNS to send messages to mobile endpoints subscribed to a topic. The concept is the same as subscribing other endpoint types. The difference is that Amazon SNS communicates with the push notification services for the subscribed mobile endpoints to receive push notification messages sent to the topic. Figure 11.10 shows a mobile endpoint as a subscriber to an Amazon SNS topic. The mobile endpoint communicates with push notification services, whereas the other endpoints do not.

**FIGURE 11.10** Amazon SNS mobile endpoint subscriber



## Add Device Tokens or Registration IDs

When you first register an app and mobile device with a notification service, such as Apple Push Notification Service (APNS) and Google Cloud Messaging for Android (GCM), device tokens or registration IDs return from the notification service. When you add the device tokens or registration IDs to Amazon SNS, they use the `PlatformApplicationArn`

API to create an endpoint for the app and device. When Amazon SNS creates the endpoint, an `EndpointArn` returns, and this is how Amazon SNS knows to which app and mobile device to send the notification message.

You can add device tokens and registration IDs to Amazon SNS by using these methods:

- Manually add a single token to AWS from the AWS Management Console.
- Migrate existing tokens from a CSV file to AWS from the AWS Management Console.
- Upload several tokens by using the `CreatePlatformEndpoint` API.
- Register tokens from devices that will install your apps in the future.

## Create Amazon SNS Endpoints

You can use one of two options to create Amazon SNS endpoints for device tokens or registration IDs.

**Amazon Cognito** Your mobile app requires credentials to create and associate endpoints with your Amazon SNS platform application. AWS recommends that you use temporary security credentials that expire after a period of time. You can use Amazon SNS to receive an event with the new endpoint ARN, or you can use the `ListEndpointByPlatformApplication` API to view the full list of endpoints registered with Amazon SNS.

**Proxy Server** If your application infrastructure is already set up for your mobile apps to call in and register on each installation, you can use your server to act as a proxy and pass the device token to Amazon SNS mobile push notifications. This includes any user data that you would like to store. The proxy server connects to Amazon SNS with your AWS credentials and uses the `CreatePlatformEndpoint` API call to upload the token information. The newly created endpoint ARN is returned, which your server can store to make subsequent publish calls to Amazon SNS.

## Billing, Limits, and Restrictions

Amazon SNS includes a Free Tier, which allows you to use Amazon SNS free of charge for the first 1 million Amazon SNS requests, and with no charges for the first 100,000 notifications over HTTP, no charges for the first 100 notifications over SMS, and no charges for the first 1,000 notifications over email.

With Amazon SNS, there is no minimum fee, and you pay only for what you use. You pay \$0.50 per 1 million Amazon SNS requests, \$0.06 per 100,000 notification deliveries over HTTP, and \$2 per 100,000 notification deliveries over email. *For SMS messaging, users can send 100 free notification deliveries, and for subsequent messages, charges vary by destination country.*

*By default, Amazon SNS offers 10 million subscriptions per topic and 100,000 topics per account.* To request a higher limit, contact AWS Support.



Amazon SNS supports the same attributes and parameters as Amazon SQS. For more information, refer to Table 11.2, Table 11.3, and Table 11.4.

When compared with Amazon SQS, which is a queue with a pull mechanism, Amazon SNS is a fanout with a push mechanism to send messages to subscribers. This means that the Amazon SNS message is sent to a topic and then replicated and pushed to multiple Amazon SQS queues, HTTP endpoints, or email addresses. This operation eliminates the need for the message consumers to poll for any new messages. There are several differences between the Amazon SNS and Amazon SQS event-driven solutions, as listed in Table 11.5.

**TABLE 11.5** Amazon SNS and Amazon SQS Feature Comparison

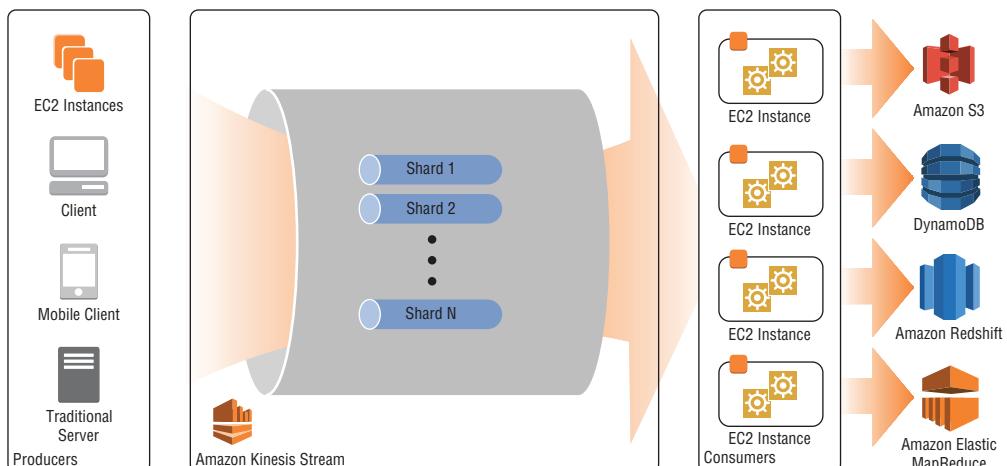
Features	Amazon SNS	Amazon SQS
Message persistence	Not persisted	Persisted
Delivery mechanism	Push (passive)	Pull (active)
Producer/consumer	Publish/subscribe (1 to N)	Send/receive (1 to 1)

## Amazon Kinesis Data Streams

*Amazon Kinesis Data Streams* is a service that ingests large amounts of data in real time and performs real-time analytics on the data. Producers write data into Amazon Kinesis Data Streams, and consumers read data from it.

Figure 11.11 illustrates the high-level architecture of Amazon Kinesis Data Streams. The producers continually push (*PushRecords*) data to Amazon Kinesis Data Streams, and the consumers process the data in *real time*. Consumers (such as a custom application running on Amazon EC2, or an Amazon Kinesis Data Firehose delivery stream) can store their results by using an AWS service, such as Amazon DynamoDB, Amazon Redshift, or Amazon Simple Storage Service (Amazon S3).

**FIGURE 11.11** Amazon Kinesis Data Streams



Multiple types of consumers can consume from the same Amazon Kinesis Data stream. *The messages are not deleted when they are consumed.* The consumers save a reference to the last message they view, and messages iterate based on sequence IDs to fetch the latest messages.

To place (`PutRecords`) data into the stream, specify the *name* of the stream, a *partition key*, and the *data blob* to add to the stream. The partition key determines the shard in the stream to which to add the data record.

All data in the shard is sent to the same worker that processes the shard. The partition key determines how to map a data record to a particular shard, so which partition key you use depends on your application logic. The number of partition keys should typically be much greater than the number of shards, and if you have enough partition keys, the data can be evenly distributed across the shards in a stream.

For example, you use the two-letter abbreviation of the state for each partition key, such as WA for Washington and WY for Wyoming. In this example, all records with a partition key of WA reside in the Washington stream, and all records with a partition key of WY reside in the Wyoming stream.

## Multiple Applications

There are several differences between Amazon Kinesis Data Streams and Amazon SQS.

In Amazon SQS, when a consumer receives a message off the queue and then processes and deletes it, the message is no longer available for any other consumer.

In Amazon Kinesis Data Streams, you can process the same message by multiple applications. Each application tracks which records it last processed. Then it requests the records that came after it. It is the application's responsibility to track its checkpoint within the data stream.

Amazon Kinesis Data Streams do not delete records after they process them, as it is possible that another application will request the message. *Records automatically delete after their retention interval expires, which you configure.* The default retention interval is 1 day, but you can extend it up to 7 days. Before the record's interval expires, multiple applications can consume the message.

## High Throughput

Amazon Kinesis uses shards to configure and support high throughput. When you create an Amazon Kinesis data stream, specify the number of shards in your stream. You can increase or decrease the number of shards through the API.

On the producer side, the shard supports 1 MB per second of ingest, or 1,000 transactions per second. Producers can write up to 1 MB per second of data, or 1000 writes.

On the consumer side, each shard supports 2 MB per second of reads, or five transactions per second. Amazon Kinesis Data Streams support twice as much data for reads as they do for writes (2 MB per second of read versus 1 MB per second of write) per shard. This allows multiple applications to read from a stream to enable more reads. Because the same records might be read by multiple applications, you require more throughput on the read side.

Amazon Kinesis Data Streams support 5,000 transactions per second for writes, but only five transactions per second for reads per shard. Reads frequently acquire many records at once. When a read request asks for all the records that came in after the last read, it acquires a large number of records. Because of this, five transactions per second per shard is sufficient to handle reads.

*To increase your throughput capacity, reshuffle the stream to adjust the number of shards.*

## Real-Time Analytics

Unlike Amazon SQS, *Amazon Kinesis Data Streams enable real-time analytics, which produces metrics from incoming data as it arrives*. The alternative is batch analytics in which the data accumulates for a period, such as 24 hours, and then is analyzed as a batch job. Real-time analytics allow you to detect patterns in the data immediately as it arrives, with a delay of only a few seconds to a few minutes.

After you define your monitoring goals and create your monitoring plan, the next step is to establish a baseline for normal *Kinesis Video Streams* performance in your environment. Measure Kinesis Video Streams performance at various times and under different load conditions. As you monitor Kinesis Video Streams, you should store a history of the monitored data that you collect. You can compare current Kinesis Video Streams performance to this historical data to help you identify normal performance patterns and performance anomalies and devise methods to address issues that may arise.

## Open Source Tools

Open source tools, such as Fluentd and Flume, support Amazon Kinesis Data Streams as a destination, and you can use them to publish messages into an Amazon Kinesis data stream.

Your custom applications, real-time or batch-oriented, can run on Amazon EC2 instances. These applications might process data with open source deep-learning algorithms or use third-party applications that integrate with Kinesis Video Streams.

## Producer Options

After you create the stream in Amazon Kinesis data stream, you need two applications to build your pipeline: a collection of *producers* that write data into the stream and *consumers* to read the data from the stream.

Here are options for you to build producers that can write into Amazon Kinesis Data Streams:

**Amazon Kinesis Agent** This is an application that reads data, appends to a log file, and writes to the stream. The benefit of the Amazon Kinesis Agent is that it does not require you to write application code.

**Amazon Kinesis Data Streams API** You write an application to use the Amazon Kinesis Data Streams API to put data on the stream.

**Amazon Kinesis Producer Library (KPL)** The KPL gives you a higher-level interface over the low-level Amazon Kinesis Data Streams API. It has the logic to retry failures and to buffer and batch-send multiple messages together. The KPL makes it easier to write messages into a stream than if you use the low-level API.

## Consumer Options

Consumers have the following options for the Amazon Kinesis Data Streams:

**Amazon Kinesis Data Streams API** You can write an application with the Amazon Kinesis Data Streams API to read data from a stream. To scale this to process large volumes of data, create a shard for each consumer. With multiple consumers that run independently, there is a risk that one of them might fail. To handle failure, coordinate between the consumers. Use the Amazon Kinesis Client Library (KCL) to track your consumers and shards.

**Amazon Kinesis Client Library** The Amazon Kinesis Client Library handles the complexity of coordinating between different consumers that read from different shards in a stream. It ensures that *no shard is ignored*, and *no shard is processed by two consumers*. The library creates a table in Amazon DynamoDB with the same name as the application name and uses this table to coordinate between the different consumers.

**AWS Lambda** AWS Lambda is another option that you can use to build Amazon Kinesis Data Streams for consumers. AWS Lambda can scale and handle fault tolerance automatically. It does not require the use of the KCL.

# Amazon Kinesis Data Firehose

*Amazon Kinesis Data Firehose* can replace the CoDA service to ingest data. In many business applications, you require a real-time pipeline, but you do not require latency of a few seconds. You can afford to have latency that can run anywhere from 1–15 minutes.

Amazon Kinesis Data Firehose is easier to use than Amazon Kinesis Data Streams, as it does not require you to write a consumer application. Data that arrives at the Amazon Kinesis Data Firehose is automatically delivered to both Amazon S3 and the other destinations. From Amazon S3, you can deliver the data to Amazon Redshift, Amazon Elasticsearch Service, and Splunk.

*Amazon Kinesis Data Firehose also handles dynamically scaling the underlying shards of the stream based on the amount of traffic.*

Amazon Kinesis Data Firehose buffers the data before it writes it to Amazon S3, with a delayed reaction to real-time data based on the length of the buffer, as detailed in Table 11.6.

**TABLE 11.6** Amazon Kinesis Data Firehose Buffers

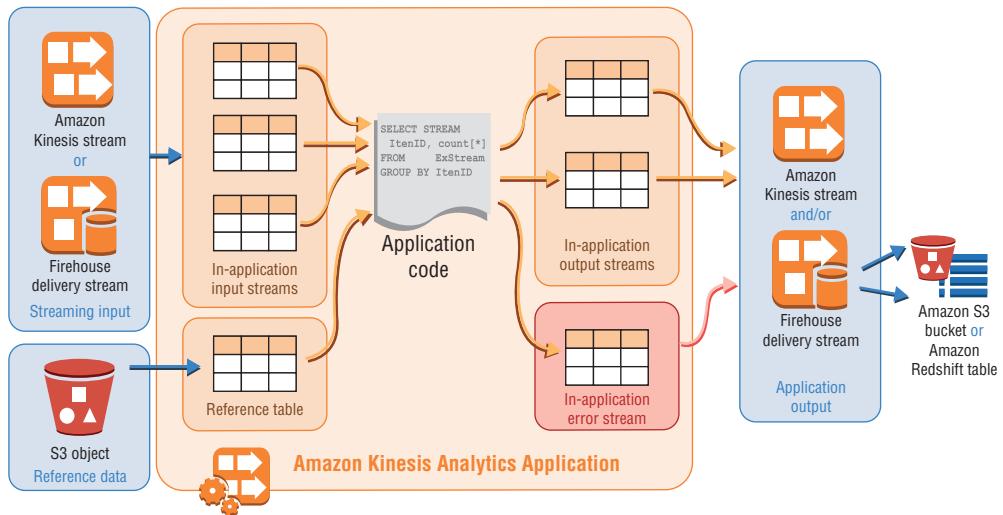
Parameter	Min	Max	Description
Buffer size	1 MB	128 MB	How much data Kinesis Data Firehose buffers
Buffer interval	60 seconds	900 seconds	How long to buffer data

With Amazon Kinesis Data Firehose, you do not need to write consumer applications or manage resources. Configure data producers to send data to Amazon Kinesis Data Firehose, and it will automatically deliver the data to the destination you specify. You can also configure Amazon Kinesis Data Firehose to transform your data before you deliver it. For example, you run a news site, and you analyze the stream of clicks from users who read the articles on your site. You want to use this analysis to move the most popular articles to the top of the page to capture news stories that are going viral. It is simple to verify that a story acquires a large number of hits, with a lag of only a few minutes.

## Amazon Kinesis Data Analytics

*Amazon Kinesis Data Analytics* enables you to process and analyze streaming data with standard structured query language (SQL). It also enables you to run SQL code against streaming sources to perform time-series analytics, feed real-time dashboards, and create real-time metrics. Amazon Kinesis Data Analytics supports ingesting from either Amazon Kinesis Data Streams or Amazon Kinesis Data Firehose, and it continuously reads and processes streaming data. You can configure destinations where Amazon Kinesis Data Analytics sends the results, as shown in Figure 11.12. Amazon Kinesis Data Analytics supports the following destinations:

- Amazon Kinesis Data Firehose
- Amazon S3
- Amazon Redshift
- Amazon ES
- Splunk
- AWS Lambda
- Amazon Kinesis Data Streams

**FIGURE 11.12** Amazon Kinesis Data Analytics flow

Use cases for Amazon Kinesis Data Analytics include the following:

**Generate time series analytics** You can calculate metrics over time windows and stream values to Amazon S3 or Amazon Redshift through a Firehose delivery stream.

**Feed real-time dashboards** You can send aggregated and processed streaming data results downstream to feed real-time dashboards.

**Create real-time metrics** You can create custom metrics and triggers for use in real-time monitoring, notifications, and alarms.

## Amazon Kinesis Video Streams

Use the *Amazon Kinesis Video Streams* service to push device video content into AWS and then onto the cloud to process that content and detect patterns in it.

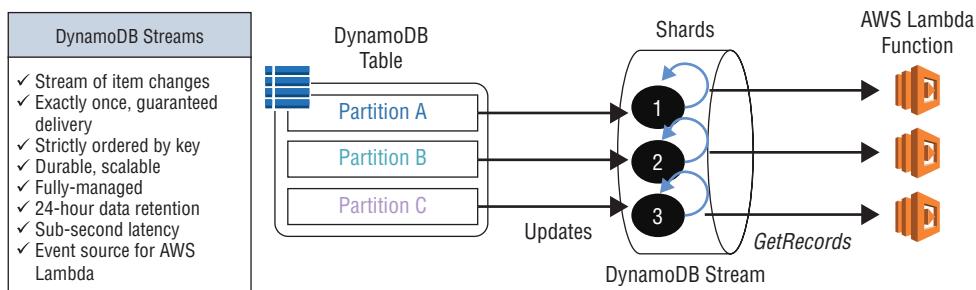
You can use Amazon Kinesis Video Streams to build computer vision and machine learning applications.

A single stream can support one producer connection and three consumer connections at a time.

# Amazon DynamoDB Streams

*Amazon DynamoDB Streams* integrates with Amazon DynamoDB to publish a message every time a change is made in a table. When you insert, delete, or update an item, Amazon DynamoDB produces an *event*, which publishes it to the Amazon DynamoDB Streams, as shown in Figure 11.13. To use this table-level feature, enable Amazon DynamoDB Streams on the table.

**FIGURE 11.13** Amazon DynamoDB Stream



## Amazon DynamoDB Streams Use Case

Amazon DynamoDB Streams is a database trigger for Amazon DynamoDB tables that you can use in any situation in which you continuously poll the database to indicate if a variable changes. An example would be a customer who publishes a vote in an Amazon DynamoDB table called votes in an online application. With Amazon DynamoDB Streams, you can automatically track that change in both the votes table and in the consumer table to update the aggregate votes counted.

## Amazon DynamoDB Streams Consumers

*Amazon DynamoDB* integrates with AWS Lambda so that you can create *triggers*, which are pieces of code that automatically respond to events in DynamoDB Streams. With triggers, you can build applications that react to data modifications in DynamoDB tables.

An AWS Lambda function or application that accesses the Amazon DynamoDB Streams API consumes the event when it publishes in the stream.

If you enable DynamoDB Streams on a table, you can associate the stream ARN with a Lambda function that you write. Immediately after you modify an item in the table, a new record appears in the table's stream. AWS Lambda polls the stream and invokes your AWS Lambda function synchronously when it detects new stream records.

The AWS Lambda function can perform any actions you specify, such as to send a notification or initiate a workflow. For example, you can write a Lambda function simply to copy each stream record to persistent storage, such as Amazon S3, to create a permanent audit trail of write activity in your table. Or, suppose that you have a mobile gaming app that writes to a GameScores table. Whenever the TopScore attribute of the GameScores table updates, a stream record writes to the table's stream. This event could then trigger an AWS Lambda function that posts a congratulatory message on a social media network. The function would ignore any stream records that are not updates to GameScores or that do not modify the TopScore attribute.

## Amazon DynamoDB Streams Concurrency and Shards

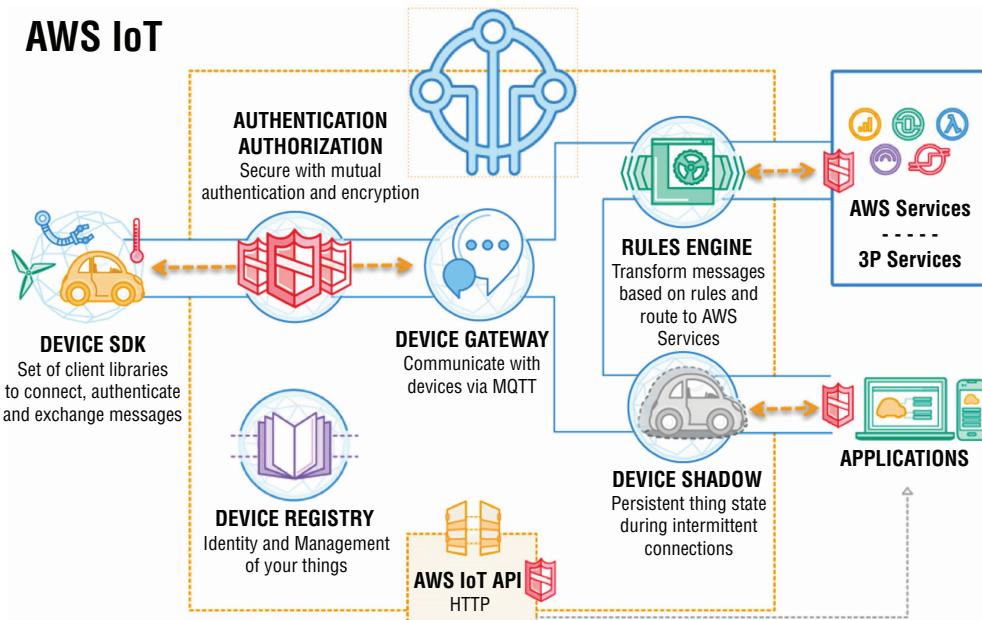
When you run an Amazon DynamoDB as a database backend, a large number of changes can occur at the same time. Amazon DynamoDB Streams publishes the changes in your table into multiple *shards*.

If you create a consumer application using AWS Lambda, each AWS Lambda instance processes the messages in a particular shard. This enables *concurrent* processing and allows Amazon DynamoDB Streams to scale to handle a high volume of concurrent changes. At any given time, each partition in an Amazon DynamoDB table maps to a single shard. The single shard captures all updates to that partition.

# AWS IoT Device Management

*AWS IoT Device Management* is a cloud-based service that makes it easy for customers to manage IoT devices securely throughout their lifecycle. Customers can use *AWS IoT Device Management* to onboard device information and configuration, organize their device inventory, monitor their fleet of devices, and remotely manage devices deployed across many locations. This remote management includes *over-the-air* (OTA) updates to device software.

AWS IoT is a service that manages devices associated with the Internet of Things, collects data from them, and sends out commands with updates to their state. The devices can communicate to the service with Message Queuing Telemetry Transport (MQTT) or HTTP. MQTT is a fire-and-forget asynchronous communication protocol that uses binary encoding. To view the AWS flow for IoT, refer to Figure 11.14.

**FIGURE 11.14** AWS IoT Device Management

## Rules Engine

When messages enter the AWS IoT Device Management service, the service dispatches them to different AWS endpoints. AWS IoT rule actions specify what to do when a rule is triggered. AWS IoT can dispatch the messages to AWS Lambda, an Amazon Kinesis data stream, a DynamoDB database, and other services. This dispatch is done through the *AWS IoT rules engine*. Rules give your devices the ability to interact with AWS products and services. Rules are analyzed, and actions occur based on the MQTT topic stream.

The IoT rules engine supports the following actions:

**CloudWatch alarm action** Use this to change the Amazon CloudWatch alarm state. Specify the state change reason and value in this call.

**Amazon CloudWatch metric action** The CloudWatch metric action allows you to capture an Amazon CloudWatch metric. Specify the metric namespace, name, value, unit, and timestamp.

**DynamoDB action** The dynamoDB action allows you to write all or part of an MQTT message to an Amazon DynamoDB table.

**DynamoDBv2 action** The dynamoDBv2 action allows you to write all or part of an MQTT message to an Amazon DynamoDB table. Each attribute in the payload is written to a separate column in the Amazon DynamoDB database.

**Elasticsearch action** The elasticsearch action allows you to write data from MQTT messages to an Amazon ES domain. You can query and visualize data in Amazon ES with tools such as Kibana.

**Firehose action** A firehose action sends data from an MQTT message that triggers the rule to a Kinesis Data Firehose stream.

**IoT Analytics action** An `iotAnalytics` action sends data from the MQTT message that triggers the rule to an AWS IoT Analytics channel.

**Kinesis action** The `kinesis` action allows you to write data from MQTT messages into a Kinesis stream.

**Lambda action** A `lambda` action calls an AWS Lambda function to pass it to a MQTT message that triggers the rule.

**Republish action** The `republish` action allows you to republish the message that triggers the role to another MQTT topic.

**S3 action** An `s3` action writes the data from the MQTT message that triggers the rule to an Amazon S3 bucket.

**Salesforce action** A `salesforce` action sends data from the MQTT message that triggers the rule to a Salesforce IoT Input Stream.

**SNS action** An `sns` action sends the data from the MQTT message that triggers the rule as an Amazon SNS push notification.

**Amazon SQS action** An `sqs` action sends data from the MQTT message that triggers the rule to an Amazon SQS queue.

**Step Functions action** A `stepFunctions` action starts execution of an AWS Step Functions state machine.



The AWS IoT rules engine does not currently retry delivery for messages that fail to publish to another service.

## Message Broker

The *AWS IoT message broker* is a publish/subscribe broker service that enables you to send messages to and receive messages from IoT. When you communicate with AWS IoT, a client sends a message to a topic address such as `Sensor/temp/room1`. The message broker then sends the message to all clients that have registered to receive messages for that topic. The act of sending the message is referred to as *publishing*. The act of registering to receive messages for a topic filter is referred to as *subscribing*.

The topic namespace is isolated for each account and region pair. For example, the `Sensor/temp/room1` topic for an account is independent from the `Sensor/temp/room1` topic for another account. This is true of regions, too. The `Sensor/temp/room1` topic in the same account in `us-east-1` is independent from the same topic in `us-east-2`.



AWS IoT does not send and receive messages across AWS accounts and regions.

The message broker maintains a list of all client sessions and the subscriptions for each session. When a message publishes on a topic, the broker checks for sessions with subscriptions that map to the topic. The broker then forwards the message to all sessions that have a currently connected client.

## Device Shadow

The *AWS IoT device shadow* is an always-available representation of the device, which allows communications back from cloud applications to the IoT devices. Cloud applications can update the device shadow even when the underlying IoT device is offline. Then when the device is brought back online, it synchronizes its final state with a query to the AWS IoT service for the current state of the instances.

A device's shadow is a JavaScript Object Notation (JSON) document that stores and retrieves current state information for a device. The device shadow service maintains a shadow for each device that you connect to AWS IoT. You can use the shadow to get and set the state of a device over MQTT or HTTP, regardless of whether the device is connected to the internet. Each device's shadow is uniquely identified by the name of the corresponding thing. The device shadow service acts as an intermediary that allows devices and applications to retrieve and update a device's shadow.

## Amazon MQ

*Amazon MQ* is a managed message broker service for Apache ActiveMQ that makes it easy to migrate to a message broker on the cloud. Amazon MQ is a managed Apache Active MQ that runs on Amazon EC2 instances that you select. AWS manages the instances, the operating system, and the Apache Active MQ software stack. You place these instances in your Amazon Virtual Private Cloud (Amazon VPC) and control access to them through security groups.

Amazon MQ makes it easy to migrate to a message broker on the cloud. A message broker allows software applications and components to communicate with the use of various programming languages, operating systems, and formal messaging protocols.

A broker is a message broker environment that runs on Amazon MQ. It is the basic building block of Amazon MQ. The combined description of the broker instance class (`m5.t2`) and size (`large`, `micro`) is a broker instance type (for example, `mq.m5.large`).

A *single-instance broker* is composed of one broker in one Availability Zone. The broker communicates with your application and with an AWS storage location.

An *active/standby broker for high availability* consists of two brokers in two different Availability Zones, which you configure in a redundant pair. These brokers communicate synchronously with your application and with a shared storage location.

You can enable automatic minor version upgrades to new versions of the broker engine, as Apache releases new versions. Automatic upgrades occur during the 2-hour maintenance window that you define by the day of the week, the time of day (in the 24-hour format), and the time zone (UTC, by default).

Amazon MQ works with your existing applications and services without the need to manage, operate, or maintain your own messaging system.

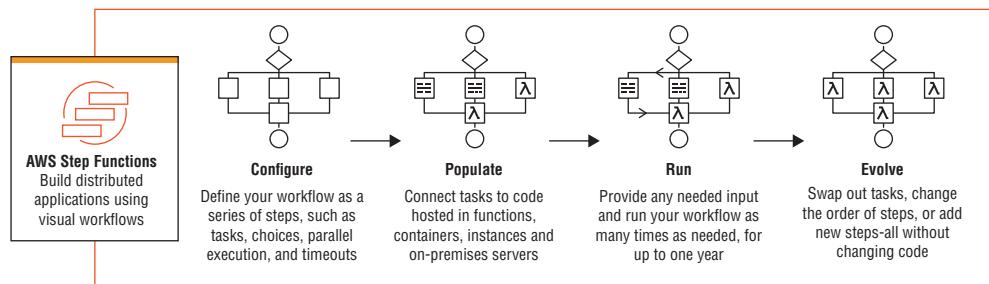
Amazon MQ is a managed message broker service that provides compatibility with many popular message brokers. AWS recommends Amazon MQ to migrate applications from current message brokers that rely on compatibility with APIs, such as JMS, or protocols like Advanced Message Queuing Protocol (AMQP), MQTT, OpenWire, and STOMP.

Amazon SQS and Amazon SNS are queue and topic services that are highly scalable, simple to use, and do not require you to set up message brokers. AWS recommends these services for new applications that can benefit from nearly unlimited scalability and simple APIs.

## AWS Step Functions

The *AWS Step Functions* service enables you to launch and develop workflows that can run for up to several months, and it allows you to monitor the progress of these workflows. You can coordinate the components of distributed applications and microservices by using visual workflows to build applications quickly, scale and recover reliably, and evolve application easily. Figure 11.15 displays the AWS Step Functions service.

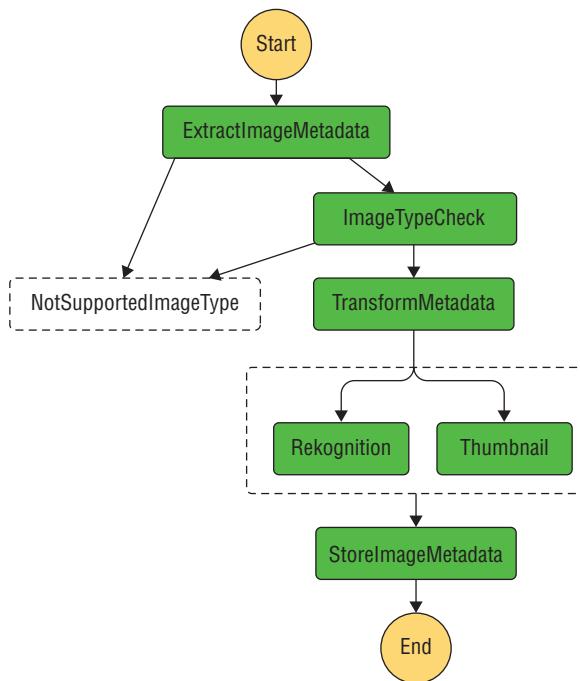
**FIGURE 11.15** AWS Step Functions



## State Machine

The *state machine* is the workflow template that is made up of a collection of states. Each time you launch a workflow, you provide it with an input. Each state that is part of the state machine receives the input, modifies it, and passes it to the next state.

These workflow templates are called *state machines*. You can use AWS Step Functions as event sources to trigger AWS Lambda. Figure 11.16 is an example of using the AWS Step Functions service.

**FIGURE 11.16** State machine code and visual workflow

Use AWS Step Functions to build visual workflows that enable fast translation of business requirements into technical requirements. You can build applications in a matter of minutes. When your needs change, you can swap or reorganize components without customizing any code.

AWS Step Functions manages state, checkpoints, and restarts for you to make sure that your application executes in order and as you would expect. Built-in try/catch, retry, and rollback capabilities deal with errors and exceptions automatically.

AWS Step Functions manages the logic of your application for you, and it implements basic primitives such as branching, parallel execution, and timeouts. This removes extra code that may be repeated in your microservices and functions.

A finite state machine can express an algorithm as a number of states, their relationships, and their input and output. AWS Step Functions allows you to coordinate individual tasks by expressing your workflow as a finite state machine, written in the *Amazon States Language*. Individual states can decide based on their input, perform actions, and pass output to other states. In Step Functions, you can express your workflows in the Amazon States Language, and the *Step Functions console* provides a graphical representation of that state machine to help visualize your application logic.

Names identify states, which can be any string, but must be unique within the state machine specification. Otherwise, it can be any valid string in JSON text format.



An instance of a state exists until the end of its execution.

States can perform the following functions in your state machine:

**Task state:** Performs work in your state machine

**Choice state:** Makes a choice between branches of execution

**Fail or Succeed state:** Stops an execution with a failure or success

**Pass state:** Passes inputs to outputs or inject corrected data

**Wait state:** Provides a delay for a certain amount of time or until a specified time/date

**Parallel state:** Begins parallel branches of execution

Here is an example state named `HelloWorld`, which performs an AWS Lambda function:

```
"HelloWorld": {  
    "Type": "Task",  
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloFunction",  
    "Next": "AfterHelloWorldState",  
    "Comment": "Run the HelloWorld Lambda function"  
}
```

States share the following common features:

- Each state must have a `Type` field to indicate what type of state it is.
- Each state can have an optional `Comment` field to hold a human-readable comment about, or description of, the state.
- Each state (except a `Succeed` or `Fail` state) requires a `Next` field or, alternatively, can become a terminal state if you specify an `End` field.

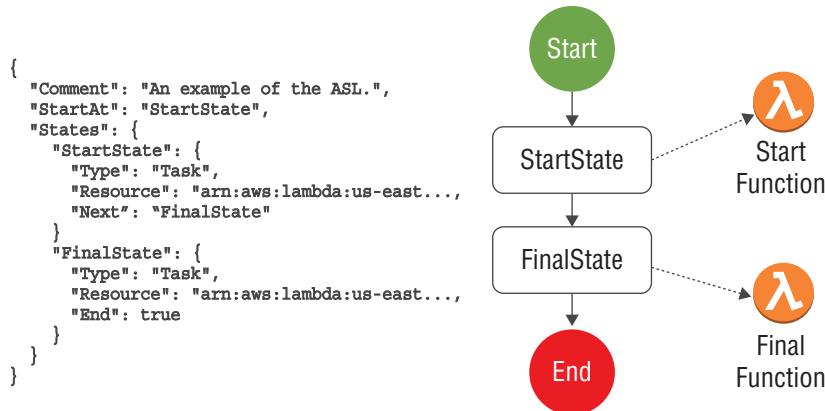
These fields are common within each state:

- **Type (Required):** The state's type.
- **Next:** The name of the next state that runs when the current state finishes. Some state types, such as `Choice`, allow multiple transition states.
- **End:** Designates this state as a terminal state (it ends the execution) if set to true. There can be any number of terminal states per state machine. State supports only one `Next` or `End` statement. Some state types, such as `Choice`, do not support or use the `End` field.

- **Comment (Optional):** Holds a human-readable description of the state.
- **InputPath (Optional):** A path that selects a portion of the state's input to pass to the state's task process. If omitted, it has the value \$, which designates the entire input.
- **OutputPath (Optional):** A path that selects a portion of the state's input to pass to the state's output. If omitted, it has the value \$, which designates the entire input.

To see the Amazon Function State Language, refer to Figure 11.17.

**FIGURE 11.17** Amazon Function State Language



## Task State

A *task state* involves a form of compute. A task executes on an AWS Lambda function or on an Amazon EC2 instance. An *activity* is a task that executes on an Amazon EC2 instance.

A task state ("Type": "Task") represents a single unit of work that a state machine performs.

In addition to the common state fields, task state fields include the following:

**Resource (Required):** Amazon Resource Name (ARN) that uniquely identifies the specific task to execute.

**ResultPath (Optional):** Specifies where in the input to place the results from the task Resource. The input is filtered as prescribed by the OutputPath field (if present) before being used as the state's output.

**Retry (Optional):** An array of objects, called *Retriers*, that define a retry policy if the state encounters runtime errors.

**Catch (Optional):** An array of objects, called *Catchers*, that define a fallback state. This state is executed if the state encounters runtime errors and the retry policy has been exhausted or is not defined.

**TimeoutSeconds** (Optional): If the task runs longer than the specified number of seconds, this state fails with a States.Timeout error name. This must be a positive, nonzero integer. If not provided, the default value is 99999999.

**HeartbeatSeconds** (Optional): If more time than the specified seconds elapses between heartbeats from the task, then this state fails with a States.Timeout error name. This must be a positive, nonzero integer less than the number of seconds specified in the TimeoutSeconds field. If not provided, the default value is 99999999.

A Task state either must set the End field to true if the state ends the execution or must provide a state in the Next field that runs upon completion of the Task state. Here's an example:

```
"ActivityState": {  
    "Type": "Task",  
    "Resource": "arn:aws:states:us-east-1:123456789012:activity:HelloWorld",  
    "TimeoutSeconds": 300,  
    "HeartbeatSeconds": 60,  
    "Next": "NextState"  
}
```

The ActivityState schedules the HelloWorld activity for execution in the us-east-1 region on the caller's account. When HelloWorld completes, the Next state (NextState) runs.

If this task fails to complete within 300 seconds or it does not send heartbeat notifications in intervals of 60 seconds, then the task is marked as failed. Set a Timeout value and a HeartbeatSeconds interval for long-running activities.

## Specify Resource Amazon Resource Names in Tasks

To specify the Resource field's Amazon Resource Name (ARN), use the syntax:

arn:partition:service:region:account:task\_type:name

where:

- partition is the AWS Step Functions partition to use, most commonly aws.
- service indicates the AWS service that you use to execute the task, which is one of the following values:
  - states for an activity
  - lambda for an AWS Lambda function
- region is the AWS region in which the Step Functions activity/state machine type or AWS Lambda function has been created.
- account is your Account ID.
- task\_type is the type of task to run. It is one of the following:
  - activity: An activity
  - function: An AWS Lambda function
- name is the registered resource name (activity name or AWS Lambda function name).



### Step Functions Referencing ARNs

You cannot reference ARNs across partitions with Step Functions. For example, aws-cn cannot invoke tasks in the aws partition, and vice versa.

## Task Types

Task types support activity and AWS Lambda functions.

**Activity** Activities represent workers (processes or threads) that you implement and host, which perform a specific task.

Activity resource ARNs use the following syntax:

```
arn:partition:states:region:account:activity:name
```



Create activities with Step Functions (using a `CreateActivity` API action or the Step Functions console) before their first use.

**AWS Lambda Functions** Lambda tasks execute a function using AWS Lambda. To specify an AWS Lambda function, use the ARN of the AWS Lambda function in the Resource field. AWS Lambda function Resource ARNs use the following syntax:

```
arn:partition:lambda:region:account:function:function_name
```

Here's an example:

```
"LambdaState": {  
    "Type": "Task",  
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:HelloWorld",  
    "Next": "NextState"  
}
```

When the AWS Lambda function you specify in the Resource field completes, its output is sent to the state you identify in the Next field (NextState).

## Choice State

The *Choice state* enables control flow between several different paths based on the input you select. In a choice state, you place a *condition* on the input. The state machine evaluates the condition, and it follows the path of the first condition that is true about the input.



A Choice state may have more than one Next, but only one within each Choice Rule. A Choice state cannot use End.

A Choice state ("Type": "Choice") adds branch logic to a state machine.

Other Choice state fields include the following:

**Choices (Required)** An array of Choice Rules that determines which state the state machine transitions to next

**Default (Optional, Recommended)** The name of the state to transition to if none of the transitions in Choices is taken



Choice states do not support the End field. They also use Next only inside their Choices field.



You must specify the \$.type field. If the state input does not contain the \$.type field, the execution fails, and an error displays in the execution history.

This is an example of a Choice state and other states to which it transitions:

```
"ChoiceStateX": {  
    "Type": "Choice",  
    "Choices": [  
        {  
            "Not": {  
                "Variable": "$.type",  
                "StringEquals": "Private"  
            },  
            "Next": "Public"  
        },  
        {  
            "Variable": "$.value",  
            "NumericEquals": 0,  
            "Next": "ValueIsZero"  
        },  
        {  
            "And": [  
                {  
                    "Variable": "$.value",  
                    "NumericGreaterThanOrEqual": 20  
                },  
                {  
                    "Variable": "$.value",  
                    "NumericLessThan": 30  
                }  
            ]  
        }  
    ]  
}
```

```

        }
    ],
    "Next": "ValueInTwenties"
}
],
"Default": "DefaultState"
},

"Public": {
    "Type" : "Task",
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Foo",
    "Next": "NextState"
},

"ValueIsZero": {
    "Type" : "Task",
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Zero",
    "Next": "NextState"
},

"ValueInTwenties": {
    "Type" : "Task",
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:Bar",
    "Next": "NextState"
},

"DefaultState": {
    "Type": "Fail",
    "Cause": "No Matches!"
}
}

```

In this example, the state machine starts with the input value:

```
{
  "type": "Private",
  "value": 22
}
```

Step Functions transitions to the ValueInTwenties state, based on the value field.

If there are no matches for the Choice state's Choices, the state in the Default field runs instead. If there is no value in the Default state, the execution fails with an error.

## Choice Rules

A Choice state must have a `Choices` field whose value is a nonempty array and whose every element is an object called a Choice Rule. A Choice Rule contains the following:

**Comparison** Two fields that specify an input variable to compare, the type of comparison, and the value to which to compare the variable.

**Next field** The value of this field must match a state name in the state machine.

This example checks whether the numerical value is equal to 1:

```
{  
  "Variable": "$.foo",  
  "NumericEquals": 1,  
  "Next": "FirstMatchState"  
}
```

This example checks whether the string is equal to `MyString`:

```
{  
  "Variable": "$.foo",  
  "StringEquals": "MyString",  
  "Next": "FirstMatchState"  
}
```

This example checks whether the string is greater than `MyStringABC`:

```
{  
  "Variable": "$.foo",  
  "StringGreater Than": "MyStringABC",  
  "Next": "FirstMatchState"  
}
```

This example checks whether the timestamp is equal to `2018-01-01T12:00:00Z`:

```
{  
  "Variable": "$.foo",  
  "TimestampEquals": "2018-01-01T12:00:00Z",  
  "Next": "FirstMatchState"  
}
```

Step Functions examines each of the Choice Rules in the order that they appear in the `Choices` field and transitions to the state you specify in the `Next` field of the first Choice Rule in which the variable matches the value equal to the comparison operator.

The comparison supports the following operators:

- `And`
- `BooleanEquals`

- Not
- NumericEquals
- NumericGreaterThan
- NumericGreaterThanOrEqual
- NumericLessThan
- NumericLessThanOrEqual
- Or
- StringEquals
- StringGreaterThan
- StringGreaterThanOrEqual
- StringLessThan
- StringLessThanOrEqual
- TimestampEquals
- TimestampGreaterThan
- TimestampGreaterThanOrEqual
- TimestampLessThan
- TimestampLessThanOrEqual

For each of these operators, the value corresponds to the appropriate type: string, number, Boolean, or timestamp. Step Functions do not attempt to match a numeric field to a string value. However, because timestamp fields are logically strings, you can match a timestamp field by a StringEquals comparator.



**NOTE** For interoperability, do not assume that numeric comparisons work with values outside the magnitude or precision that the IEEE 754-2008 binary64 data type represents. In particular, integers outside of the range [-2<sup>53</sup>+1, 2<sup>53</sup>-1] might fail to compare in the way that you would expect.

Timestamps (for example, 2016-08-18T17:33:00Z) must conform to RFC3339 profile ISO 8601, with the following further restrictions:

- An uppercase T must separate the date and time portions.
- An uppercase Z must denote that a numeric time zone offset is not present.

To understand the behavior of string comparisons, see the Java compareTo documentation here:

<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#compareTo-java.lang.String->

The values of the And and Or operators must be nonempty arrays of Choice Rules that do not themselves contain Next fields. Likewise, the value of a Not operator must be a single Choice Rule with no Next fields.

You can create complex, nested Choice Rules using And, Not, and Or. However, the Next field can appear only in a top-level Choice Rule.

## Parallel State

The *Parallel state* enables control flow to execute several different execution paths at the same time in parallel. This is useful if you have activities or tasks that do not depend on each other, can execute in parallel, and can help your workflow complete faster.

You can use the Parallel state ("Type": "Parallel") to create parallel branches of execution in your state machine.

In addition to the common state fields, Parallel states introduce these additional fields:

**Branches (Required)** An array of objects that specify state machines to execute in parallel. Each such state machine object must have the fields States and StartAt and mean the same as those in the top level of a state machine.

**ResultPath (Optional)** Specifies where in the input to place the output of the branches. The OutputPath field (if present) filters the input before it becomes the state's output.

**Retry (Optional)** An array of objects, called *Retriers*, which define a retry policy in case the state encounters runtime errors.

**Catch (Optional)** An array of objects, called *Catchers*, which define a fallback state that executes in case the state encounters runtime errors and you do not define the retry policy or it has been exhausted.

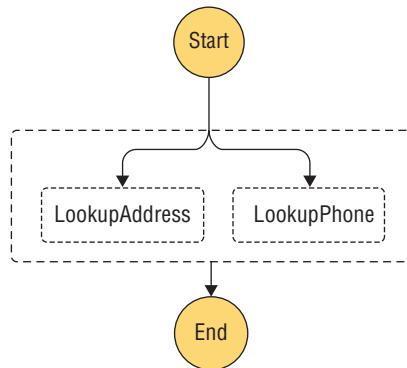
A Parallel state causes AWS Step Functions to execute each branch. The state starts with the name of the state in that branch's StartAt field, as concurrently as possible, and waits until all branches terminate (reach a terminal state) before it processes the Parallel state's Next field. Here's an example:

```
{  
  "Comment": "Parallel Example.",  
  "StartAt": "LookupCustomerInfo",  
  "States": {  
    "LookupCustomerInfo": {  
      "Type": "Parallel",  
      "End": true,  
      "Branches": [  
        {  
          "StartAt": "LookupAddress",  
          "States": {  
            "LookupAddress": {  
              "Type": "Task",  
              "Resource":  
                "arn:aws:lambda:us-east-1:123456789012:function:AddressFinder",  
              "End": true  
            }  
          }  
        }  
      ]  
    }  
  }  
}
```

```
        }
    }
},
{
  "StartAt": "LookupPhone",
  "States": {
    "LookupPhone": {
      "Type": "Task",
      "Resource":
        "arn:aws:lambda:us-east-1:123456789012:function:PhoneFinder",
      "End": true
    }
  }
}
]
```

In this example, the `LookupAddress` and `LookupPhone` branches execute in parallel. Figure 11.18 displays the workflow in the Step Functions console.

**FIGURE 11.18** Parallel state visual workflow



Each branch must be self-contained. A state in one branch of a `Parallel` state must not have a `Next` field that targets a field outside of that branch, nor can any other state outside the branch transition into that branch.

## Parallel State Output

A Parallel state provides each branch with a copy of its own input data (`InputPath`). It generates output, which is an array with one element for each branch that contains the output from that branch. There is no requirement that all elements be of the same type. You can insert the output array into the input data (and the whole sent as the Parallel state's output) with a `ResultPath` field. Here's an example:

```
{
  "Comment": "Parallel Example.",
  "StartAt": "FunWithMath",
  "States": {
    "FunWithMath": {
      "Type": "Parallel",
      "End": true,
      "Branches": [
        {
          "StartAt": "Add",
          "States": {
            "Add": {
              "Type": "Task",
              "Resource": "arn:aws:swf:us-east-1:123456789012:task:Add",
              "End": true
            }
          }
        },
        {
          "StartAt": "Subtract",
          "States": {
            "Subtract": {
              "Type": "Task",
              "Resource": "arn:aws:swf:us-east-1:123456789012:task:Subtract",
              "End": true
            }
          }
        }
      ]
    }
  }
}
```

If the FunWithMath state was given the array [3, 2] as input, then both the Add and Subtract states receive that array as input. The output of Add would be 5, that of Subtract would be 1, and the output of the Parallel state would be an array.

[ 5, 1 ]

### Error Handling

If any branch fails, because of an unhandled error or by a transition to a Fail state, the entire Parallel state fails, and all of its branches stop. If the error is not handled by the Parallel state itself, Step Functions stops the execution with an error.



When a Parallel state fails, invoked AWS Lambda functions continue to run, and activity workers that process a task token do not stop.

To stop long-running activities, use heartbeats to detect whether Step Functions has stopped its branch, and stop workers that are processing tasks. If the state has failed, calling `SendTaskHeartbeat`, `SendTaskSuccess`, or `SendTaskFailure` generates an error.

You cannot stop AWS Lambda functions that are running. If you have implemented a fallback, use a Wait state so that cleanup work happens after the AWS Lambda function finishes.

### End State

A state machine completes its execution when it reaches an *end state*. Each state defines either a next state or an end state, and the end state terminates the execution of the step function.

### Input and Output

Each execution of the state machine requires an input as a JSON object and passes that input to the first state in the workflow. The state machine receives the initial input by the process initiating the execution. Each state modifies the input JSON object that it receives and injects its output into this object. The final state produces the output of the state machine.

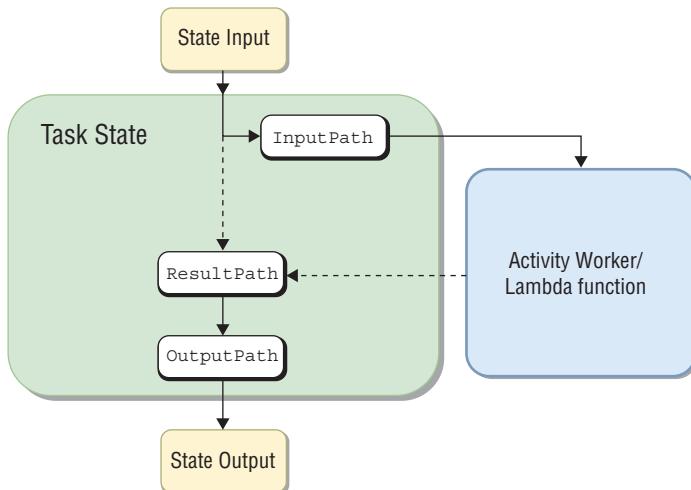
Individual states receive JSON as the input and usually pass JSON as the output to the next state. *Understand how this information flows from state to state and learn how to filter and manipulate this data to design and implement workflows in AWS Step Functions effectively.*

In the Amazon States Language, three components filter and control the flow of JSON from state to state: `InputPath`, `OutputPath`, and `ResultPath`.

Figure 11.19 shows how JSON information moves through a task state. `InputPath` selects which components from the input to pass to the task of the Task state, for example, an AWS Lambda function. `ResultPath` then selects what combination of the state input

and the task result to pass to the output. `OutputPath` can filter the JSON output to limit further the information that passes to the output.

**FIGURE 11.19** Input and output processing



`InputPath`, `OutputPath`, and `ResultPath` each use paths to manipulate JSON as it moves through each state in your workflow.



`ResultPath` uses reference paths, which limit scope so that it can identify only a single node in JSON.

## Paths and Reference Paths

In this section, you will learn how to use paths and reference paths to process inputs and outputs.

**Paths** In Amazon States Language, a *path* is a string that begins with \$ that you can use to identify components within JSON text. Paths follow the JsonPath syntax.

**Reference paths** A *reference path* is a path whose syntax can identify only a single node in a JSON structure.

You can access object fields with only a dot (.) and square brackets ([ ]) notation.



Paths and reference paths do not support the operators @ ... , : ? \* and functions such as `length()`.

For example, state input data contains the following values:

```
{
  "foo": 123,
  "bar": ["a", "b", "c"],
  "car": {
    "cdr": true
  }
}
```

In this case, the reference paths return the following:

```
$.foo => 123
$.bar => ["a", "b", "c"]
$.car.cdr => true
```

Certain states use paths and reference paths to control the flow of a state machine or configure a state's options.

### Paths in InputPath, ResultPath, and OutputPath Fields

To specify how to use part of the state's input and what to send as output to the next state, you can use `InputPath`, `OutputPath`, and `ResultPath`.

For `InputPath` and `OutputPath`, you must use a path that follows the `JsonPath` syntax.

For `ResultPath`, you must use a reference path.

**InputPath** The `InputPath` field selects a portion of the state's input to pass to the state's task to process. If you omit the field, it receives the `$` value, which represents the entire input. If you use `null`, the input is not sent to the state's task, and the task receives JSON text representing an empty object `{}`.



A path can yield a selection of values. Here's an example:

```
{ "a": [1, 2, 3, 4] }
```

If you apply the path `$.a[0:2]`, the result is as follows:

```
[ 1, 2 ]
```

**ResultPath** If a state executes a task, the task results are sent along as the state's output, which becomes the input for the next task.

If a state does not execute a task, the state's own input is sent, unmodified, as its output. However, when you specify a path in the value of a state's `ResultPath` and `OutputPath` fields, different scenarios become possible.

The `ResultPath` field takes the results of the state's task that executes and places them in the input. Next, the `OutputPath` field selects a portion of the input to send as the state's

output. The `ResultPath` field might add the results of the state's task that executes to the input, overwrites an existing part, or overwrites the entire input.

- If the `ResultPath` matches an item in the state's input, only that input item is overwritten with the results of executing the state's task. The entire modified input becomes available to the state's output.
- If the `ResultPath` does not match an item in the state's input, an item adds to the input. The item contains the results of executing the state's task. The expanded input becomes available to the state's output.
- If the `ResultPath` has the default value of `$`, it matches the entire input. In this case, the results of the state execution overwrite the input entirely, and the input becomes available to pass along.
- If the `ResultPath` is `null`, the results of executing the state are discarded, and the input remains the same.



ResultPath field values must be reference paths.

**OutputPath** If the `OutputPath` matches an item in the state's input, only that input item is selected. This input item becomes the state's output.

- If the `OutputPath` does not match an item in the state's input, an exception specifies an invalid path.
- If the `OutputPath` has the default value of `$`, this matches the entire input completely. In this case, the entire input passes to the next state.
- If the `OutputPath` is `null`, JSON text represents an empty object, `{}`, and is sent to the next state.

The following example demonstrates how `InputPath`, `ResultPath`, and `OutputPath` fields work in practice. Consider this input for the current state:

```
{  
  "title": "Numbers to add",  
  "numbers": { "val1": 3, "val2": 4 }  
}
```

In addition, the state has the following `InputPath`, `ResultPath`, and `OutputPath` fields:

```
"InputPath": "$.numbers",  
"ResultPath": "$.sum",  
"OutputPath": "$"
```

The state's task receives only the numbers object from the input. In turn, if this task returns 7, the output of this state equals the following:

```
{  
  "title": "Numbers to add",  
  "numbers": { "val1": 3, "val2": 4 }  
  "sum": 7  
}
```

You can modify the OutputPath as follows:

```
"InputPath": "$.numbers",  
"ResultPath": "$.sum",  
"OutputPath": "$.sum"
```

As before, you use the following state input data:

```
{  
  "numbers": { "val1": 3, "val2": 4 }  
}
```

However, now the state output data is 7.

## AWS Step Functions Use Case

You can use state machines to process long-running workflows. For example, if a customer orders a book and it requires several different events, use the state machine to run all the events. When the customer orders the book, the state machine creates a credit card transaction, generates a tracking number for the book, notifies the warehouse to ship the order, and then emails the tracking number to the customer. The AWS Step Functions service runs all these steps.

The benefit of AWS Step Functions is that it enables the compute to be stateless. The AWS Lambda functions and the Amazon EC2 instances provide compute to the state machine to execute in a stateless way. AWS Lambda functions and Amazon EC2 do not have to remember the information about the state of the current execution. The AWS Step Functions service remembers the information about the state of the current execution.

## Summary

This chapter covered the different services to refactor larger systems into smaller components that can communicate with each other through infrastructure services. To be successful, the refactoring infrastructure must exist, which enables the different components to communicate with each other. You also now know about the different infrastructure communication services that AWS provides for different use cases.

# Exam Essentials

**Know how refactoring to microservices is beneficial and what services it includes.** This includes the use of the Amazon Simple Queue Service (Amazon SQS), Amazon Simple Notification Service (Amazon SNS), Amazon Kinesis Data Streams, Amazon Kinesis services, Amazon DynamoDB Streams, AWS Internet of Things (IoT), Amazon Message Query (Amazon MQ), and AWS Step Functions.

**Know about the Amazon Simple Queue Service.** Know that the Amazon Simple Queue Service (Amazon SQS) is a fully managed message queuing service that makes it easy to decouple and scale microservices, distributed systems, and serverless applications. *There will be questions about the dead-letter queue and how to pass messages with Amazon CloudWatch.*

**Know about the Amazon Simple Notification Service.** Familiarize yourself with the Amazon Simple Notification Service (Amazon SNS) and how it is a flexible, fully managed producer/consumer (publisher/subscriber) messaging and mobile notifications web service for coordinating the delivery of messages to subscribing to endpoints and clients. Amazon SNS coordinates and manages the delivery or sending of messages to subscriber endpoints or clients.

**Know about Amazon Kinesis Data Streams.** Study how Amazon Kinesis Data Streams is a service for ingesting large amounts of data in real time and for performing real-time analytics on the data. Producers write data into Amazon Kinesis Data Streams, and consumers read data from it. Be familiar with the use of multiple applications, high throughput, real-time analytics, and open source tools that Kinesis supports. *There will be questions about producer and consumer options on the exam.*

**Know about Amazon Kinesis Data Firehose.** Familiarize yourself with Amazon Kinesis Data Firehose latency. Amazon Kinesis Data Firehose also handles automatic scaling of the underlying shards of the stream based on the amount of traffic.

**Know about Amazon Kinesis Data Analytics.** *There will also be questions about how Amazon Kinesis Data Analytics enables you to process and analyze streaming data with standard SQL.* Make sure that you know which destinations it supports.

**Know about Amazon Kinesis Video Streams.** Know that the Amazon Kinesis Video Streams service allows you to push device video content into AWS and then onto the cloud to process that content and detect patterns in it. You can also use Amazon Kinesis Video Streams to build computer vision and machine learning applications.

**Know about Amazon DynamoDB Streams.** Remember that Amazon DynamoDB Streams allows Amazon DynamoDB to publish a message every time a change is made in a table. When you insert, update, or delete an item, Amazon DynamoDB produces an event that publishes it to the Amazon DynamoDB Streams. Familiarize yourself with tables, consumers, concurrency, and streams.

**Know about AWS Internet of Things (AWS IoT).** Make sure that you know that AWS IoT Device Management is a cloud-based device management service that makes it easy for customers to manage IoT devices securely throughout their lifecycle. Memorize information on the rules engine, message, broker, and device shadow.

**Know about Amazon MQ.** Know that the primary use for Amazon MQ is to enable customers who use Apache Active MQ to migrate to the cloud. A message broker allows software applications and components to communicate with various programming languages, operating systems, and formal messaging protocols. Know how the Amazon SQS and Amazon SNS differ from Amazon MQ.

**Know about AWS Step Functions.** The exam includes questions that require a thorough understanding of AWS Step Functions. Ensure that you know each step in the state machine, task state, Choice state, Parallel state, and end state. Remember the inputs and outputs in the step functions.

**Know how state information flows and how to filter it.** Understand how this information flows from state to state and learn how to filter and manipulate this data to design and implement workflows effectively in AWS Step Functions.

## Resources to Review

Amazon Simple Notification Service (Amazon SNS):

<https://aws.amazon.com/sns/>

Amazon SNS Documentation:

<https://docs.aws.amazon.com/sns/latest/dg/welcome.html>

Amazon SNS FAQs:

<https://aws.amazon.com/sns/faqs/>

Amazon SNS Message Filtering:

<https://docs.aws.amazon.com/sns/latest/dg/message-filtering.html>

Amazon SNS Mobile Push Notifications:

<https://docs.aws.amazon.com/sns/latest/dg/SNSMobilePush.html>

Amazon SNS Mobile Push High-Level Steps:

<https://docs.aws.amazon.com/sns/latest/dg/mobile-push-pseudo.html>

Add Amazon SNS Device Tokens or Registration IDs:

<https://docs.aws.amazon.com/sns/latest/dg/mobile-push-send-devicetoken.html>

Amazon Simple Queue Service (Amazon SQS) Documentation:

[https://aws.amazon.com/documentation/sqs/?id=docs\\_gateway](https://aws.amazon.com/documentation/sqs/?id=docs_gateway)

Amazon SQS Dead-Letter Queues:

<https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-dead-letter-queues.html>

Amazon SQS FAQs:

<https://aws.amazon.com/sqs/faqs/>

Amazon SQS Features:

<https://aws.amazon.com/sqs/features/>

Amazon SQS Resources:

<https://aws.amazon.com/sqs/resources/>

Amazon SQS Visibility Timeout:

<https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-visibility-timeout.html>

Amazon SQS FIFO Queues:

<https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/FIFO-queues.html>

Amazon CloudWatch Documentation:

<https://aws.amazon.com/documentation/cloudwatch/>

What is Amazon CloudWatch?

<https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html>

Pass Messages via Amazon CloudWatch:

[https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/US\\_SetupSNS.html](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/US_SetupSNS.html)

Amazon Kinesis:

<https://aws.amazon.com/kinesis/>

Amazon Kinesis Data Streams Data Streams:

<https://aws.amazon.com/kinesis/data-streams/>

Amazon Kinesis Data Streams Resources:

<https://aws.amazon.com/kinesis/data-streams/resources/>

Resharding Amazon Kinesis Data Streams:

<https://docs.aws.amazon.comstreams/latest/dev/kinesis-using-sdk-java-resharding.html>

Splitting Amazon Kinesis Data Streams Shards:

<https://docs.aws.amazon.comstreams/latest/dev/kinesis-using-sdk-java-resharding-split.html>

Amazon Kinesis Data Firehose Data Delivery:

<https://docs.aws.amazon.com/firehose/latest/dev/basic-deliver.html>

Amazon Kinesis Data Firehose FAQs:

<https://aws.amazon.com/kinesis/data-firehose/faqs/>

Amazon Kinesis Data Firehose Streaming:

<https://aws.amazon.com/kinesis/data-firehose/>

Amazon Kinesis Video Streams and How It Works:

<https://docs.aws.amazon.com/kinesisvideostreams/latest/dg/how-it-works.html>

Amazon Kinesis Video Streams and What It is:

<https://docs.aws.amazon.com/kinesisvideostreams/latest/dg/what-is-kinesis-video.html>

Amazon Kinesis Analytics RecordFormat:

[https://docs.aws.amazon.com/kinesisanalytics/latest/dev/API\\_RecordFormat.html](https://docs.aws.amazon.com/kinesisanalytics/latest/dev/API_RecordFormat.html)

Amazon Kinesis Analytics API CSV Mapping Parameters:

[https://docs.aws.amazon.com/kinesisanalytics/latest/dev/API\\_CSVMappingParameters.html](https://docs.aws.amazon.com/kinesisanalytics/latest/dev/API_CSVMappingParameters.html)

Amazon Kinesis Analytics API Mapping Parameters:

[https://docs.aws.amazon.com/kinesisanalytics/latest/dev/API\\_MappingParameters.html](https://docs.aws.amazon.com/kinesisanalytics/latest/dev/API_MappingParameters.html)

Amazon Kinesis Analytics Source Reference:

<https://docs.aws.amazon.com/kinesisanalytics/latest/dev/how-it-works-input.html#source-reference>

Amazon DynamoDB Streams KCL Adapter:

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.KCLAdapter.html>

Amazon DynamoDB Streams:

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html>

AWS IoT Analytics User Guide

[https://aws.amazon.com/documentation/iotanalytics/?id=docs\\_gateway](https://aws.amazon.com/documentation/iotanalytics/?id=docs_gateway)

AWS Internet of Things (IoT):

<https://docs.aws.amazon.com/iotanalytics/latest/userguide/welcome.html>

AWS IoT Analytics Rule Actions:

<https://docs.aws.amazon.com/iot/latest/developerguide/iot-rule-actions.html>

Amazon MQ:

<https://aws.amazon.com/amazon-mq/>

AWS Step Functions:

<https://aws.amazon.com/step-functions/>

AWS Step Functions Documentation:

<https://aws.amazon.com/documentation/step-functions/>

AWS Step Functions Input/Output Filters:

<https://docs.aws.amazon.com/step-functions/latest/dg/concepts-input-output-filtering.html>

AWS Step Functions State Languages:

<https://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-states.html>

Amazon States Languages:

<https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html>

# Exercises

## EXERCISE 11.1

### Create an Amazon SQS Queue, Add Messages, and Receive Messages

In this exercise, you will use the AWS SDK for Python (Boto) to create an Amazon SQS queue, and then you will put messages in the queue. Finally, you will receive messages from this queue and delete them.

1. Make sure that you have AWS administrator credentials set up in your account.
2. Install the AWS SDK for Python (Boto).  
Refer to <https://aws.amazon.com/sdk-for-python/>.
3. Enter the following code into your development environment for Python or the IPython shell.

This is the code that you downloaded at the beginning of the exercises.

```
# Test SQS.  
import boto3  
  
# Pretty print.  
import pprint
```

---

*(continued)*

**EXERCISE 11.1 (*continued*)**

```
pp = pprint.PrettyPrinter(indent=2)

# Create queue.
sns = boto3.resource('sns')
queue = sns.create_queue(QueueName='test1')
print(queue.url)

# Get existing queue.
queue = sns.get_queue_by_name(QueueName='test1')
print(queue.url)

# Get all queues.
for queue in sns.queues.all(): print queue

# Send message.
response = queue.send_message(MessageBody='world')
pp.pprint(response)

# Send batch.
response = queue.send_messages(Entries=[
    { 'Id': '1', 'MessageBody': 'world' },
    { 'Id': '2', 'MessageBody': 'hello' } ])
pp.pprint(response)

# Receive and delete all messages.
for message in queue.receive_messages():
    pp.pprint(message)
    message.delete()

# Delete queue.
queue.delete()
```

**4. Run the code.**

This creates a queue, sends messages to it, receives messages from it, deletes the messages, and then deletes the queue.

- 5.** To experiment with the queue further, remove the comment //queue.delete() from the last line, which deletes the queue.
  - 6.** After you are satisfied with your changes, delete the code.
-

**EXERCISE 11.2****Send an SMS Text Message to Your Mobile Phone with Amazon SNS**

In this exercise, you will use Amazon SNS to publish an SMS message to your mobile phone. This solution can be useful when you run a job that will take several hours to complete, and you do not want to wait for it to finish. Instead, you can have your app send you an SMS text message when it is done.

1. Enter the following code into your development environment for Python or the IPython shell.

This is the code that you downloaded at the beginning of the exercises.

```
import boto3

# Create SNS client.
sns_client = boto3.client('sns')

# Send message to your mobile number.
# (Replace dummy mobile number with your number.)
sns_client.publish(
    PhoneNumber='1-222-333-3333',
    Message='Hello from your app')
```

2. Replace the PhoneNumber value with your own number.

The 1 at the beginning is the U.S. country code, 222 is the area code, and 333-3333 is the mobile phone number.

3. Run the code.

Check your phone to view the message.

**EXERCISE 11.3****Create an Amazon Kinesis Data Stream and Write/Read Data**

In this exercise, you will create an Amazon Kinesis data stream, put records on it (write to the stream), and then get those records back (read from the stream). At the end, you will delete the stream.

1. Enter this code into your development environment for Python or the IPython shell.

This is the code that you downloaded at the beginning of the exercises.

```
import boto3
import random
```

*(continued)*

**EXERCISE 11.3 (*continued*)**

```
import json

# Create the client.
kinesis_client = boto3.client('kinesis')

# Create the stream.
kinesis_client.create_stream(
    StreamName='donut-sales',
    ShardCount=2)

# Wait for stream to be created.
waiter = kinesis_client.get_waiter('stream_exists')
waiter.wait(StreamName='donut-sales')

# Store each donut sale using location as partition key.
location = 'california'
data = b'{"flavor":"chocolate","quantity":12}'
kinesis_client.put_record(
    StreamName='donut-sales',
    PartitionKey=location, Data=data)
print("put_record: " + location + " -> " + data)

# Next lets put some random records.

# List of location, flavors, quantities.
locations = ['california', 'oregon', 'washington', 'alaska']
flavors = ['chocolate', 'glazed', 'apple', 'birthday']
quantities = [1, 6, 12, 20, 40]

# Generate some random records.
for i in xrange(20):

    # Generate random record.
    flavor = random.choice(flavors)
    location = random.choice(locations)
    quantity = random.choice(quantities)
    data = json.dumps({"flavor": flavor, "quantity": quantity})

    # Put record onto the stream.
    kinesis_client.put_record(
        StreamName='donut-sales',
        PartitionKey=location, Data=data)
```

```
print("put_record: " + location + " -> " + data)

# Get the records.

# Get shard_ids.
response = kinesis_client.list_shards(StreamName='donut-sales')
shard_ids = [shard['ShardId'] for shard in response['Shards']]
print("list_shards: " + str(shard_ids))

# For each shard_id print out the records.
for shard_id in shard_ids:

    # Print current shard_id.
    print("shard_id=" + shard_id)

    # Get a shard iterator from this shard.
    # TRIM_HORIZON means start from earliest record.
    response = kinesis_client.get_shard_iterator(
        StreamName='donut-sales',
        ShardId=shard_id,
        ShardIteratorType='TRIM_HORIZON')
    shard_iterator = response['ShardIterator']

    # Get records on shard and print them out.
    response = kinesis_client.get_records(ShardIterator=shard_iterator)
    records = response['Records']
    for record in records:
        location = record['PartitionKey']
        data = record['Data']
        print("get_records: " + location + " -> " + data)

# Delete the stream.
kinesis_client.delete_stream(
    StreamName='donut-sales')

# Wait for stream to be deleted.
waiter = kinesis_client.get_waiter('stream_not_exists')
waiter.wait(StreamName='donut-sales')
```

## 2. Run the code.

Observe the output and how all the records for a specific location occur in the same shard. This is because they have the same partition keys. All records with the same partition key are sent to the same shard.

---

**EXERCISE 11.4****Create an AWS Step Functions State Machine 1**

In this exercise, you will create an AWS Step Functions state machine. The state machine will extract price and quantity from the input and inject the billing amount into the output.

This state machine will calculate how much to bill a customer based on the price and quantity of an item they purchased.

1. Sign in to the AWS Management Console and open the Step Functions console at <https://console.aws.amazon.com/step-functions/>.
2. Select **Get Started**.
3. On the **Define state machine** page, select **Author from scratch**.
4. In **Name type**, enter **order-machine**.
5. Enter the code for the state machine definition.

This is the code that you downloaded at the beginning of the exercises.

```
{  
    "StartAt": "CreateOrder",  
    "States": {  
        "CreateOrder": {  
            "Type": "Pass",  
            "Result": {  
                "Order" : {  
                    "Customer" : "Alice",  
                    "Product" : "Coffee",  
                    "Billing" : { "Price": 10.0, "Quantity": 4.0 }  
                }  
            },  
            "Next": "CalculateAmount"  
        },  
        "CalculateAmount": {  
            "Type": "Pass",  
            "Result": 40.0,  
            "ResultPath": "$.Order.Billing.Amount",  
            "OutputPath": "$.Order.Billing",  
            "End": true  
        }  
    }  
}
```

---

**6.** On the **State machine definition** page, select **Reload**.

This updates the visual representation of the state machine. The state machine consists of two states: CreateOrder and CalculateAmount. They are both Pass types and pass hardcoded values.

This is useful for build the outline of your final state machine. You can also use this to debug ResultPath and OutputPath. ResultPath determines where in the input to inject the result. OutputPath determines what data passes to the next state.

**7.** Select **Create state machine**.**8.** Select **Start execution**.**9.** In **Input type**, enter `{}`.**10.** Select **Start execution**.**11.** Expand **Output**, and it should look like the following:

```
{
    "Price": 10,
    "Quantity": 4,
    "Amount": 40
}
```

In CalculateAmount, "ResultPath": "\$.Order.Billing.Amount" injected Amount under Billing under Order. Then in the same element, "OutputPath": "\$.Order.Billing" threw away the rest of the input and passed only the contents of the Billing element forward. This is why the output contains only Price, Quantity, and Amount.

**12.** (Optional) Experiment with different values of ResultPath to understand how it affects where the result of a state inserts into the input.**13.** (Optional) Experiment with different values of OutputPath to understand how it affects what part of the data passes to the next state.

---

**EXERCISE 11.5****Create an AWS Step Functions State Machine 2**

In this exercise, you will create an AWS Step Functions state machine. The state machine will contain a conditional branch. It will use the Choice state to choose which state to transition to next.

---

*(continued)*

**EXERCISE 11.5 (continued)**

The state machine inspects the input and based on it decides whether the user ordered green tea, ordered black tea, or entered invalid input.

1. Sign in to the AWS Management Console and open the Step Functions console at: <https://console.aws.amazon.com/step-functions/>.
2. Select **Get Started**.
3. On the **Define state machine** page, select **Author from scratch**.
4. In **Name type**, enter **tea-machine**.
5. Enter the state machine definition.

This is the code that you downloaded at the beginning of the exercises.

```
{  
    "Comment" :  
        "Input should look like {'tea':'green'} with double quotes instead of  
        single.",  
    "StartAt": "MakeTea",  
    "States" : {  
        "MakeTea": {  
            "Type": "Choice",  
            "Choices": [  
                {"Variable":"$.tea","StringEquals":"green","Next":"Green"},  
                {"Variable":"$.tea","StringEquals":"black","Next":"Black"}  
            ],  
            "Default": "Error"  
        },  
        "Green": { "Type": "Pass", "End": true, "Result": "Green tea" },  
        "Black": { "Type": "Pass", "End": true, "Result": "Black tea" },  
        "Error": { "Type": "Pass", "End": true, "Result": "Bad input" }  
    }  
}
```

6. On the **State machine definition** page, select **Reload**.

This updates the visual representation of the state machine. The MakeTea state is a Choice state. Based on the input it receives, it will branch out to Green, Black, or Error.

7. Select **Create state machine**.
8. Select **Start execution**.

9. In **Input type**, enter this value:

```
{ "tea" : "green" }
```

10. Select **Start execution**.

11. Select **Expand Output**, and it should look like this:

"Green tea"

12. (Optional) Experiment with different inputs to the state machine.

For example, try the following inputs:

For **Input type**, enter **black tea**. This input works.

```
{ "tea" : "black" }
```

For **Input type**, enter **orange tea**. This produces an error.

```
{ "tea" : "orange" }
```

13. Change the state machine so that **orange tea** also works.
-

## Review Questions

1. When a user submits a build into the build system, you want to send an email to the user, acknowledging that you have received the build request, and start the build. To perform these actions at the same time, what type of a state should you use?
  - A. Choice
  - B. Parallel
  - C. Task
  - D. Wait
2. Suppose that a queue has no consumers. The queue has a maximum message retention period of 14 days. After 14 days, what happens?
  - A. After 14 days, the messages are deleted and move to the dead-letter queue.
  - B. After 14 days, the messages are deleted and do not move to the dead-letter queue.
  - C. After 14 days, the messages are not deleted.
  - D. After 14 days, the messages become invisible.
3. What is size of an Amazon Simple Queue Service (Amazon SQS) message?
  - A. 256 KB
  - B. 128 KB
  - C. 1 MB
  - D. 5 MB
4. You want to send a 1 GB file through Amazon Simple Queue Service (Amazon SQS). How can you do this?
  - A. This is not possible.
  - B. Save the file in Amazon Simple Storage Service (Amazon S3) and then send a link to the file on Amazon SQS.
  - C. Use AWS Lambda to push the file.
  - D. Bypass the log server so that it does not get overloaded.
5. You want to design an application that sends a status email every morning to the system administrators. Which option will work?
  - A. Create an Amazon SQS queue. Subscribe all the administrators to this queue. Set up an Amazon CloudWatch event to send a message on a daily cron schedule into the Amazon SQS queue.
  - B. Create an Amazon SNS topic. Subscribe all the administrators to this topic. Set up an Amazon CloudWatch event to send a message on a daily cron schedule to this topic.

- C. Create an Amazon SNS topic. Subscribe all the administrators to this topic. Set up an Amazon CloudWatch event to send a message on a daily cron schedule to an AWS Lambda function that generates a summary and publishes it to this topic.
  - D. Create an AWS Lambda function that sends out an email to the administrators every day directly with SMTP.
6. What is the size of an Amazon Simple Notification Service (Amazon SNS) message?
- A. 256 KB
  - B. 128 KB
  - C. 1 MB
  - D. 5 MB
7. You have an Amazon Kinesis data stream with one shard and one producer. How many consumer applications can you consume from the stream?
- A. One consumer
  - B. Two consumers
  - C. Limitless number of consumers
  - D. Limitless number of consumers as long as all consumers consume fewer than 2 MB and five transactions per second
8. A company has a website that sells books. It wants to find out which book is selling the most in real time. Every time a book is purchased, it produces an event. What service can you use to provide real-time analytics on the sales with a latency of 30 seconds?
- A. Amazon Simple Queue Service (Amazon SQS)
  - B. Amazon Simple Notification Service (Amazon SNS)
  - C. Amazon Kinesis Data Streams
  - D. Amazon Kinesis Data Firehose
9. A company sells books in the 50 states of the United States. It publishes each sale into an Amazon Kinesis data stream with two shards. For the partition key, it uses the two-letter abbreviation of the state, such as WA for Washington, WY for Wyoming, and so on. Which of the following statements is true?
- A. The records for Washington are all on the same shard.
  - B. The records for both Washington and Wyoming are on the same shard.
  - C. The records for Washington are on a different shard than the records for Wyoming.
  - D. The records for Washington are evenly distributed between the two shards.

**10.** What are the options for Amazon Kinesis Data Streams producers?

- A. Amazon Kinesis Agent
- B. Amazon Kinesis Data Steams API
- C. Amazon Kinesis Producer Library (KPL)
- D. Open-Source Tools
- E. All of these are valid options.

# Chapter 12



AWS® Certified Developer Official Study Guide  
By Nick Alteen, Jennifer Fisher, Casey Gerena, Wes Gruver, Asim Jalil,  
Heiwad Osman, Marife Pagan, Santosh Patlolla and Michael Roth  
Copyright © 2019 by Amazon Web Services, Inc.

# Serverless Compute

---

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

#### **Domain 1: Deployment**

- ✓ 1.3 Prepare the application deployment package to be deployed to AWS.

- ✓ 1.4 Deploy serverless applications.

#### **Domain 2: Security**

- ✓ 2.1 Make authenticated calls to AWS Services.

#### **Domain 3: Development with AWS Services**

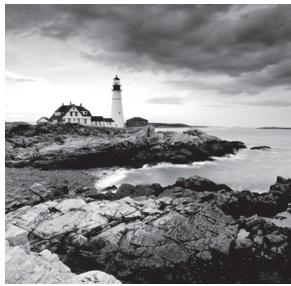
- ✓ 3.1 Write code for serverless applications.

- ✓ 3.4 Write code that interacts with AWS services by using APIs, SDKs, and AWS CLI.

#### **Domain 5: Monitoring and Troubleshooting**

- ✓ 5.1 Write code that can be monitored.

- ✓ 5.2 Perform root cause analysis on faults found in testing or production.



## Introduction to Serverless Compute

*Serverless compute* is a cloud computing execution model in which the AWS Cloud acts as the server and dynamically manages the allocation of machine resources. AWS bases the price on the amount of resources the application consumes rather than on prepurchased units of capacity.

For decades, people used local computers to interpret, process, and execute code, and they have encountered relatively few serious issues when they ran powerful web and data processing applications on their servers. However, this model has its problems.

The first issue with running servers is that you have to purchase them; a costly endeavor depending on the number of servers that you require for your project. Servers also depreciate and become obsolete, which facilitates the need to replace them.

Second, you must patch servers on a frequent and consistent manner to prevent security exploits. They require time-consuming maintenance to prolong their longevity. Servers may also experience hardware failures, which you must diagnose and repair. All of this consumes both time and money, which could be spent on other efforts such as improving applications.

Third, the needs of the users change over time. When an application first releases, it is not frequently accessed, and infrastructure needs are minimal. Over time, the application grows, and the infrastructure must also grow to accommodate it. This requires more servers, more maintenance, and more hardware costs. It also requires more time, as to add new servers to your data center can take several weeks or months.

## AWS Lambda

*AWS Lambda* is the AWS *serverless compute* platform that enables you to run code without provisioning or managing servers. With AWS Lambda, you can run code for nearly any type of application or backend service—with zero administration. Only upload your code,

and AWS Lambda performs all the tasks you require to run and scale your code with high availability. You can configure code to trigger automatically from other AWS services, or call it directly from any web or mobile app. AWS Lambda is sometimes referred to as a *function-as-a-service* (FaaS). AWS Lambda executes code whenever the function is triggered, and no *Amazon Elastic Compute Cloud* (Amazon EC2) instances need to be spun up in your infrastructure.

AWS Lambda offers several key benefits over Amazon EC2. First, there are no servers to manage. You are no longer responsible for provisioning or managing servers, patching servers, or worrying about high availability.

Second, you do not have to concern yourself with scaling. AWS Lambda automatically scales your application by running code in response to each trigger. Your code runs in parallel and processes each trigger individually, scaling precisely with the size of the workload.

Third, when you run Amazon EC2 instances, you are responsible for costs associated with the instance runtime. It does not matter whether your site receives little to no traffic—if the server is running, there are costs. With AWS Lambda, if no one executes the function or if the function is not triggered, no charges are incurred.

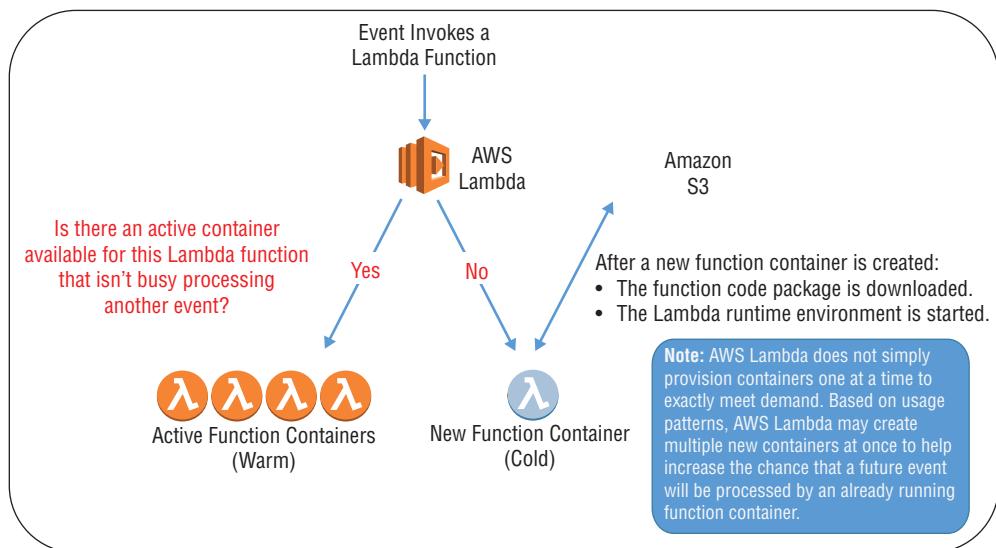
With the use of AWS Lambda and other AWS services, you can begin to decouple the application, which allows you to improve your ability to both scale horizontally and create asynchronous systems.

## Where Did the Servers Go?

Serverless computing still requires servers, but the server management and capacity planning decisions are hidden from the developer or operator. You can use serverless code with code you deploy in traditional styles, such as microservices. Alternatively, you can write applications to be purely serverless with no provisioned servers.

AWS Lambda uses containerization to run your code. When your function is triggered, it creates a *container*. Then your code executes and returns your application or services the result. If a container is created on the first invocation, AWS refers to this as a *cold start*. Once the container starts to run, it remains active for several minutes before it terminates. If an invocation runs on a container that is already available, that invocation runs on a *warm container*.

By default, AWS Lambda runs containers inside the AWS environment, and not within your personal AWS account. However, you can also run AWS Lambda inside your *Amazon Virtual Private Cloud* (Amazon VPC). Figure 12.1 shows the execution flow process.

**FIGURE 12.1** AWS Lambda execution flow

## Monolithic vs. Microservices Architecture

*Microservices* are an architectural and organizational approach to software development whereby software is composed of small independent services that communicate over well-defined application programming interfaces (APIs). Small, self-contained teams own these services.

Historically, applications have been developed as *monolithic* architectures. With monolithic architectures, all processes are tightly coupled and run as a single service. If one process of the application experiences a spike in demand, you have to scale the entire architecture. To add or improve a monolithic application's features, it becomes more complicated as the code base grows. This complexity limits experimentation and makes it difficult to implement new ideas. Monolithic architectures increase the risk for application availability, as many dependent and tightly coupled processes increase the impact of a single process failure.

Microservices are more agile, and scaling is more flexible than with monolithic applications. You can deploy new portions of your code faster and more easily. With AWS Lambda and other services, you can begin to create microservices for your application.

## AWS Lambda Functions

This section discusses how to use AWS Lambda to execute functions, such as how to create, secure, trigger, debug, monitor, improve, and test AWS Lambda functions.

## Languages AWS Lambda Supports

AWS Lambda functions currently support the following languages:

- C# (.NET Core 1.0)
- C# (.NET Core 2.0)
- Go 1.x
- Java 8
- Node.js 4.3
- Node.js 6.10
- Node.js 8.10
- Python 2.7
- Python 3.6

## Creating an AWS Lambda Function

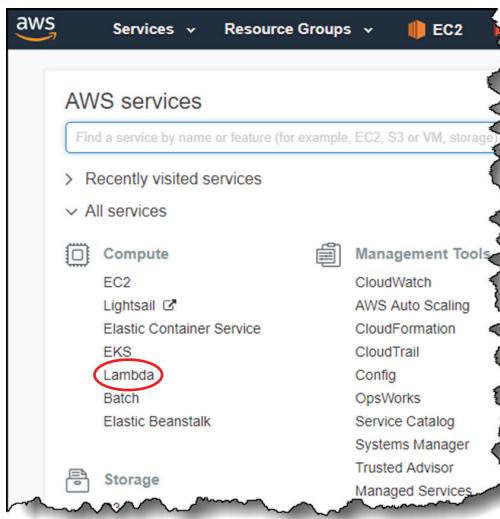
You can use any of the following methods to access AWS services and create an AWS Lambda function that will call an AWS service:

- AWS Management Console—graphical user interface (GUI)
- AWS command line interface (AWS CLI)—Linux Shell and Windows PowerShell
- AWS Software Development Kit (AWS SDK)—Java, .NET, Node.js, PHP, Python, Ruby, Go, Browser, and C++
- AWS application programming interface (API)—send HTTP/HTTPS requests manually using API endpoints



In this chapter, you will create an AWS Lambda function and properties with the AWS Management Console. In the “Exercises” section, you will use AWS CLI and the Python SDK for the AWS Lambda function.

Launch the AWS Management Console, and select the AWS Lambda service under the Compute section (see Figure 12.2).

**FIGURE 12.2** AWS Management Console

When you create an AWS Lambda function, there are three options:

**Author from scratch** Manually create all settings and options.

**Blueprints** Select a preconfigured template that you can modify.

**Serverless application repository** Deploy a publicly shared application with the AWS Serverless Application Model (AWS SAM).



There is no charge for this repository, as it is where you deploy a prebuilt application and then modify it.

When authoring from scratch, you must provide three details to create an AWS Lambda function:

- Name—name of the AWS Lambda function
- Runtime—language in which the AWS Lambda function is written
- Role—permissions of your functions

After you name the function and select a runtime language, you define an *AWS Identity and Access Management (IAM) role*.

## Execution Methods/Invocation Models

There are two invocation models for AWS Lambda.

- Nonstreaming Event Source (Push Model)—Amazon Echo, Amazon Simple Storage Service (Amazon S3), Amazon Simple Notification Service (Amazon SNS), and Amazon Cognito
- Streaming Event Source (Pull Model)—Amazon Kinesis or Amazon DynamoDB stream

Additionally, you can execute an AWS Lambda function synchronously or asynchronously. The `InvocationType` parameter determines when to invoke an AWS Lambda function. This parameter has three possible values:

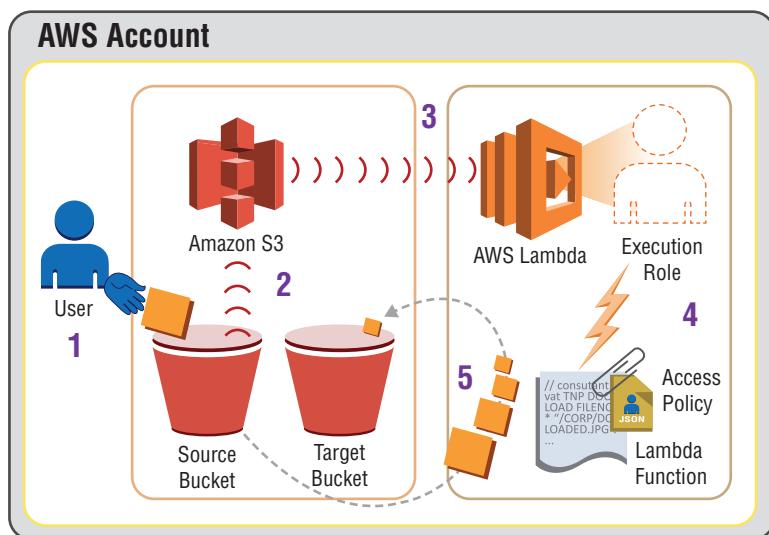
- **RequestResponse**—Execute synchronously.
- **Event**—Execute asynchronously.
- **DryRun**—Test that the caller permits the invocation but does not execute the function.

With an event source (*push model*), a service such as Amazon S3 invokes the AWS Lambda function each time an event occurs with the bucket you specify.

Figure 12.3 illustrates the push model flow.

1. You create an object in a bucket.
2. Amazon S3 detects the object-created event.
3. Amazon S3 invokes your AWS Lambda function according to the event source mapping in the bucket notification configuration.
4. AWS Lambda verifies the permissions policy attached to the AWS Lambda function to ensure that Amazon S3 has the necessary permissions.
5. AWS Lambda executes the AWS Lambda function, and the AWS Lambda function receives the event as a parameter.

**FIGURE 12.3** Amazon S3 push model



With a *pull model* invocation, AWS Lambda polls a stream and invokes the function upon detection of a new record on the stream. Amazon Kinesis uses the pull model.

Figure 12.4 illustrates the sequence for a pull model.

1. A custom application writes records to an Amazon Kinesis stream.
2. AWS Lambda continuously polls the stream and invokes the AWS Lambda function when the service detects new records on the stream. AWS Lambda knows which stream to poll and which AWS Lambda function to invoke based on the event source mapping you create in AWS Lambda.
3. Assuming that the attached permissions policy, which allows AWS Lambda to poll the stream, is verified, then AWS Lambda executes the function.

**FIGURE 12.4** Amazon Kinesis pull model

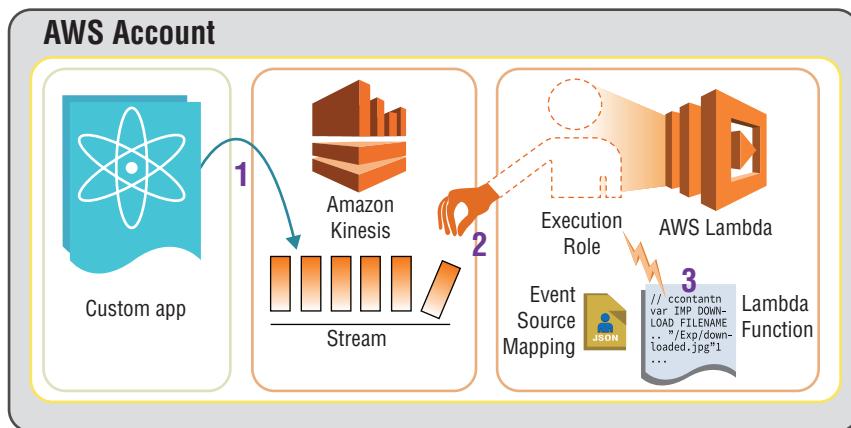


Figure 12.4 uses an Amazon Kinesis stream, but the same principle applies when you work with an Amazon DynamoDB stream.

The final way to invoke an AWS Lambda function applies to custom applications with the RequestResponse invocation type. Using this invocation method, AWS Lambda executes the function synchronously, returns the response immediately to the calling application, and alerts you to whether the invocation occurs.

Your application creates an HTTP POST request to pass the necessary parameters and invoke the function. To use this type of invocation model, you must set the RequestResponse in the X-Amz-Invocation-Type HTTP header.

## Securing AWS Lambda Functions

AWS Lambda functions include two types of permissions.

*Execution permissions* enable the AWS Lambda function to access other AWS resources in your account. For example, if the AWS Lambda function needs access to Amazon S3 objects, you grant permissions through an AWS IAM role that AWS Lambda refers to as an *execution role*.

*Invocation permissions* are the permissions that an event source needs to communicate with your AWS Lambda function. Depending on the invocation model (push or pull), you can either update the access policy you associate with your AWS Lambda function (push) or update the execution role (pull).

AWS Lambda provides the following AWS permissions policies:

**LambdaBasicExecutionRole** Grants permissions only for the Amazon CloudWatch logactions to write logs. Use this policy if your AWS Lambda function does not access any other AWS resources except writing logs.

**LambdaKinesisExecutionRole** Grants permissions for Amazon Kinesis data stream and Amazon CloudWatch log actions. If you are writing an AWS Lambda function to process Amazon Kinesis stream events, attach this permissions policy.

**LambdaDynamoDBExecutionRole** Grants permissions for Amazon DynamoDB stream and Amazon CloudWatch log actions. If you are writing an AWS Lambda function to process Amazon DynamoDB stream events, attach this permissions policy.

**LambdaVPCAccessExecutionRole** Grants permissions for Amazon EC2 actions to manage elastic network interfaces. If you are writing an AWS Lambda function to access resources inside the Amazon VPC service, attach this permissions policy. The policy also grants permissions for Amazon CloudWatch log actions to write logs.

## Inside the AWS Lambda Function

The primary purpose of AWS Lambda is to execute your code. You can use any libraries, artifacts, or compiled native binaries that execute on top of the runtime environment as part of your function code package. Because the runtime environment is a Linux-based *Amazon Machine Image* (AMI), always compile and test your components within the matching environment. To accomplish this, use AWS Serverless Application Model (AWS SAM) CLI to test AWS Lambda functions locally, which is also referred to as AWS SAM CLI (<https://github.com/awslabs/aws-sam-cli>).

## Function Package

Two parts of the AWS Lambda function are considered critical: the *function package* and the *function handler*. The function code package contains everything you need to be available locally when your function is executed. At minimum, it contains your code for the function itself, but it may also contain other assets or files that your code references upon execution. This includes binaries, imports, or configuration files that your code/function needs. The maximum size of a function code package is 50 MB compressed and 250 MB extracted/decompressed.

You can create the AWS Lambda function by using the AWS Management Console, SDK, API, or with the CreateFunction API.

Use the AWS CLI to create a function with the commands, as shown here:

```
aws lambda create-function \
--region us-east-2 \
--function-name MyCLITestFunction \
--role arn:aws:iam:account-id:role/role_name \
--runtime python3.6 \
--handler MyCLITestFunction.my_handler \
--zip-file fileb://path/to/function/file.zip
```

## Function Handler

When the AWS Lambda function is invoked, the code execution begins at the handler. The handler is a method inside the AWS Lambda function that you create and include in your package. The handler syntax depends on the language you use for the AWS Lambda function.

For *Python*, the handler is written as follows:

```
def aws_lambda_handler(event, context):
    return "My First AWS Lambda Function"
```

For *Java*, it is written as follows:

```
MyOutput output handlerName(MyEvent event, Context context) {
    return "My First AWS Lambda Function"
}
```

For *Node.js*, it is written as follows:

```
exports.handlerName = function(event, context, callback) {
    return "My First AWS Lambda Function"
}
```

And for *C#*, it is written as follows:

```
myOutput HandlerName(MyEvent event, ILambdaContext context) {
    return "My First AWS Lambda Function"
}
```

When the handler is specified and invoked, the code inside the handler executes. Your code can call other methods and functions within other files and classes that you store in the ZIP archive. The handler function can interact with other AWS services and make third-party API requests to web services that it might need to interact with.

## Event Object

You can pass event objects that you pass into the handler function. For example, the Python function is written as follows:

```
def aws_lambda_handler(event, context):
    return "My First AWS Lambda Function"
```

This first object you pass is the event object. The event includes all the data and metadata that your AWS Lambda function needs to implement the logic.



If you use the Amazon API Gateway service with the AWS Lambda function, it contains details of the HTTPS request that was made by the API client. Values, such as the path, query string, and the request body, are within the event object. The event object has different data depending on the event that it creates. For example, Amazon S3 has different values inside the event object than the Amazon API Gateway service.

## Context Object

The second object that you pass to the handler is the context object. The context object contains data about the AWS Lambda function invocation itself. The context and structure of the object vary based on the AWS Lambda function language. There are three primary data points that the context object contains.

**AWS Requestid** Tracks specific invocations of an AWS Lambda function, and it is important for error reports or when you need to contact AWS Support.

**Remaining time** Amount of time in milliseconds that remain before your function timeout occurs. AWS Lambda functions can run a maximum of 300 seconds (5 minutes) as of this writing, but you can configure a shorter timeout.

**Logging** Each language runtime provides the ability to stream log statements to Amazon CloudWatch Logs. The context object contains information about which Amazon CloudWatch Log stream your log statements are sent to.

# Configuring the AWS Lambda Function

This section details how to configure the AWS Lambda functions.

## Descriptions and Tags

An AWS best practice is to tag and give descriptions of your resources. As you start to scale services and create more resources on the AWS Cloud, identifying resources becomes a challenge if you do not implement a tagging strategy.

## Memory

After you write your AWS Lambda function code, configure the function options. The first parameter is `function memory`. For each AWS Lambda function, increase or decrease the function resources (amount of random access memory). You can allocate 128 MB of RAM up to 3008 MB of RAM in 64-MB increments. This dictates the amount of memory available to your function when it executes and influences the central processing unit (CPU) and network resources available to your function.

## Timeout

When you write your code, you must also configure how long your function executes for before a timeout is returned. The default timeout value is 3 seconds; however, you can specify a maximum of 300 seconds (5 minutes), the longest timeout value. You should not automatically set this function for the maximum value for your AWS Lambda function, as AWS charges based on execution time in 100-ms increments. If you have a function that fails quickly, you spend less money, because you do not wait a full 5 minutes to fail. If you wait on an external dependency that fails or you have programmed code incorrectly in your function, AWS Lambda processes the error for 5 minutes, or you can set it to fail within a fraction of that time to save time, cost, and resources.

After the execution of an AWS Lambda function completes or a timeout occurs, the response returns and all execution ceases. This includes any processes, subprocesses, or asynchronous process that your AWS Lambda function may have spawned during its execution.

## Network Configuration

There are two ways to integrate your AWS Lambda functions with external dependencies (other AWS services, publicly hosted web services, and such) with an outbound network connection: default network configuration and Amazon VPC.

With the default network configuration, your AWS Lambda function communicates from inside an Amazon VPC that AWS Lambda manages. The AWS Lambda function can connect to the internet, but not to any privately deployed resources that run within your own VPCs, such as Amazon EC2 servers.

Your AWS Lambda function uses an Amazon VPC network configuration to communicate through an elastic network interface (NIC). This interface is provisioned within the Amazon VPC and subnets, which you choose within your own account. You can assign NIC to security groups, and traffic routes based on the route tables of the subnets where you place the NIC with the Amazon EC2 service.

If your AWS Lambda function does not need to connect to any privately deployed resources, such as an Amazon EC2, select the default networking option, as the VPC option requires you to manage more details when implementing an AWS Lambda function. These details include the following:

- Select an appropriate number of subnets, while you keep in mind the principles of high availability and Availability Zones.
- Allocate enough IP addresses for each subnet.
- Implement an Amazon VPC network design that permits your AWS Lambda function to have the correct connectivity and security to meet your requirements.
- Increase the AWS Lambda cold start times if your invocation pattern requires a new NIC to create just in time.
- Configure a network address translation (NAT) (instance or gateway) to enable outbound internet access.

If you deploy an AWS Lambda function with access to your Amazon VPC, use the following formula to estimate the NIC capacity:

Projected peak concurrent executions \* (Memory in GB / 3GB)



If you had a peak of 400 concurrent executions, use 512 MB of memory. This results in about 68 network interfaces. You therefore need an Amazon VPC with at least 68 IP addresses available. This provides a /25 network that includes 128 IP addresses, minus the five that AWS uses. Next, you subtract the AWS addresses from the /25 network, which gives you 123 IP addresses.

AWS Lambda easily integrates with AWS *CloudTrail*, which records and delivers log files to your Amazon S3 bucket to monitor API usage inside your account.

## Concurrency

Though AWS allows you to scale infinitely, AWS recommends that you fine-tune your concurrency options. By default, the account-level concurrency within a given region is set with 1,000 functions as a maximum to provide you 1,000 concurrent functions to execute. You can request a limit increase for concurrent executions from the AWS Support Center.

To view the account-level setting, use the `GetAccountSettings` API and view the `AccountLimit` object and the `ConcurrentExecutions` element.

For example, run this command in the AWS CLI:

```
aws lambda get-account-settings
```

This returns the following:

```
{  
  "AccountLimit": {  
    "CodeSizeUnzipped": number,  
    "CodeSizeZipped": number,  
    "ConcurrentExecutions": number,  
    "TotalCodeSize": number,  
    "UnreservedConcurrentExecutions": number  
  },  
  "AccountUsage": {  
    "FunctionCount": number,  
    "TotalCodeSize": number  
  }  
}
```

## Concurrency Limits

Set a function-level concurrent execution limit. By default, the concurrent execution limit is enforced against the sum of the concurrent executions of all functions. The shared concurrent execution pool is referred to as the *unreserved concurrency allocation*. If you have not set up any function-level concurrency limit, the unreserved concurrency limit is the same as the *account level concurrency limit*. Any increases to the account-level limit will have a corresponding increase in the unreserved concurrency limit.

You can optionally set the *concurrent execution limit* for a function. Here are some examples:

- The default behavior is described as a surge of concurrent executions in one function, preventing the function you have isolated with an execution limit from being throttled. By setting a concurrent execution limit on a function, you reserve the specified concurrent execution value for that function.
- Functions scale automatically based on the incoming request rate, but not all resources in your architecture may be able to do so. For example, relational databases have limits on how many concurrent connections they can handle. You can set the concurrent execution limit for a function to align with the values of its downstream resources support.
- If your function connects to an Amazon VPC based resource, each concurrent execution consumes one IP within the assigned subnet. You can set the concurrent execution limit for a function to match the subnet size limits.
- If you need a function to stop processing any invocations, set the *concurrency* to 0 and then throttle all incoming executions.



By setting a concurrency limit on a function, AWS Lambda ensures that the allocation applies individually to that function, regardless of the number of traffic-processing remaining functions. If that limit is exceeded, the function is throttled. How that function behaves when throttled depends on the event source.

## Dead Letter Queues

All applications and services experience failure. Reasons that an AWS Lambda function can fail include (but are not limited to) the following:

- Function times out while trying to reach an endpoint
- Function fails to parse input data successfully
- Function experiences resource constraints, such as out-of-memory errors or other timeouts

If any of these failures occur, your function generates an exception, which you handle with a dead letter queue (DLQ). A DLQ is either an Amazon Simple Notification Service (Amazon SNS) topic or an Amazon Simple Queue Service (Amazon SQS) queue, which you configure as the destination for all failed invocation events. If a failure event occurs, the DLQ retains the message that failed, analyzes it further, and reprocesses it if necessary.

For asynchronous event sources (`InvocationType` is a declared event), after two retries with automatic back-off between the retries, the event enters the DLQ, and you configure it as either an Amazon SNS topic or Amazon SQS queue.

After you enable DLQ on an AWS Lambda function, an Amazon CloudWatch metric (`DeadLetterErrors`) is available. The metric increments whenever the dead letter message payload cannot be sent to the DLQ at any time.

## Environment Variables

AWS recommends that you separate code and configuration settings. Use *environment variables* for configuration settings. Environment variables are key-value pairs that you create and modify as part of your function configuration. These key-value pairs pass variables to your AWS Lambda function at execution time.

By default, environment variables are encrypted at rest, using a default KMS key of `aws/lambda`. Examples of environment variables that you can store include database (DB) connection strings and the type of environment (PROD, DEV, TEST, and such).

## Versioning

You can publish one or more *versions* and *aliases* for your AWS Lambda functions. Versioning is an important feature to develop serverless compute architectures, as it allows you to create multiple versions without affecting what is currently deployed in

the production environment. Each AWS Lambda function version has a unique *Amazon Resource Name* (ARN). After you publish a version, it is immutable, and you cannot change it.

After you create an AWS Lambda function, you can publish a version of that function. Here's an example with the AWS CLI:

```
aws lambda publish-version \
--region region \
--function-name myCoolFunction \
--profile devuser
```

This returns the version number along with other details after the command executes.

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "Sha265Hash",
    "FunctionName": "myCoolFunction",
    "CodeSize": 218,
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:region:account:function:myCoolFunction:1",
    "Version": "1",
    "Role": "arn:aws:iam::account:role/service-role/lambda-basic",
    "Timeout": 3,
    "LastModified": "2018-05-11T14:59:47.753+0000",
    "Handler": "lambda_function.lambda_handler",
    "Runtime": "python3.6",
    "Description": ""
}
```

## Creating an Alias

After you create a version of an AWS Lambda function, you *could* use that version number in the ARN to reference that exact version of the function. However, if you release an update to the AWS Lambda function, you must then locate all the places where you call that ARN inside the application and change the ARN to the new version number. Instead, assign an alias to a particular version and use that alias in the application.

Assign an alias of PROD to the newly created version 1, and use the alias version of the ARN in the application. This way, you change the AWS Lambda function without affecting the production environment. When you are ready to move the function to production for the next version, reassign the alias to a different version, as you can

reassign the alias while the version numbers remain static. To create an alias with the AWS CLI, use this:

```
aws lambda create-alias \
--region region \
--function-name myCoolFunction \
--description "My Alias for Production" \
--function-version "1" \
--name PROD \
--profile devuser
```

Now point applications to the PROD alias for the AWS Lambda function. This allows you to modify the function and improve code without affecting production. To migrate the PROD alias to the next version, run the command where 4 is the example version.

```
aws lambda update-alias \
--region region \
--function-name myCoolFunction \
--function-version 4 \
--name PROD \
--profile devuser
```

The production system points to version 4 of the AWS Lambda function. As you can see with versioning and aliases, you can continue to innovate your service without affecting the current production systems.

## Invoking AWS Lambda Functions

There are many ways to invoke an AWS Lambda function. You can use the push or pull method, use a custom application, or use a schedule and event to run an AWS Lambda trigger. AWS Lambda supports the following AWS services as event sources:

- Amazon S3
- Amazon DynamoDB
- Amazon Kinesis Data Streams
- Amazon SNS
- Amazon Simple Email Service
- Amazon Cognito
- AWS CloudFormation
- Amazon CloudWatch Logs
- Amazon CloudWatch Events

- AWS CodeCommit
- Scheduled events (powered by Amazon CloudWatch Events)
- AWS Config
- Amazon Alexa
- Amazon Lex
- Amazon API Gateway
- AWS IoT Button
- Amazon CloudFront
- Amazon Kinesis Data Firehose
- Manually invoking a Lambda function on demand

## Monitoring AWS Lambda Functions

As with all AWS services and all applications, it is critical to monitor your environment and application. With AWS Lambda, there are two primary tools to monitor functions to ensure that they are running correctly and efficiently: *Amazon CloudWatch* and *AWS X-Ray*.

### Using Amazon CloudWatch

Amazon CloudWatch monitors AWS Lambda functions. By default, AWS Lambda enables these metrics: invocation count, invocation duration, invocation errors, throttled invocations, iterator age, and DLQ errors.

You can leverage the reported metrics to set CloudWatch custom alarms. You can create a CloudWatch alarm that watches a single CloudWatch metric. The alarm performs one or more actions based on the value of the metric. The action can be an Amazon EC2 action, an Amazon EC2 Auto Scaling action, or a notification sent to an Amazon SNS topic.

The AWS Lambda namespace includes the metrics shown in Table 12.1.

**TABLE 12.1** AWS Lambda Amazon CloudWatch Metrics

Metric	Description
Invocations	Measures the number of times a function is invoked in response to an event or invocation API call. Replaces the deprecated RequestCount metric. Includes successful and failed invocations but does not include throttled attempts. This equals the billed requests for the function. AWS Lambda sends these metrics to CloudWatch only if they have a nonzero value. Units: Count

---

Metric	Description
Errors	<p>Measures the number of invocations that failed as the result of errors in the function (response code 4XX). Replaces the deprecated ErrorCount metric. Failed invocations may trigger a retry attempt that succeeds. This includes the following:</p> <ul style="list-style-type: none"><li>▪ Handled exceptions (for example, context.fail(error))</li><li>▪ Unhandled exceptions causing the code to exit</li><li>▪ Out-of-memory exceptions</li><li>▪ Timeouts</li><li>▪ Permissions errors</li></ul> <p>This does not include invocations that fail because invocation rates exceeded default concurrent limits (error code 429) or failures resulting from internal service errors (error code 500).</p> <p>Units: Count</p>
DeadLetterErrors	<p>Incremented when AWS Lambda is unable to write the failed event payload to DLQs that you configure. This could be because of the following:</p> <ul style="list-style-type: none"><li>▪ Permissions errors</li><li>▪ Throttles from downstream services</li><li>▪ Misconfigured resources</li><li>▪ Timeouts</li></ul> <p>Units: Count</p>

---

## Using AWS X-Ray

AWS X-Ray is a service that collects data about requests that your application serves, and it provides tools to view, filter, and gain insights into that data to identify issues and opportunities for optimization. For any traced request to the application, information displays about the request and response, but also about calls that the application makes to downstream AWS resources, microservices, databases, and HTTP web APIs.

There are three main parts to the X-Ray service:

- Application code runs and uses the *AWS X-Ray SDK* (Node.js, Java, and .NET, Ruby, Python, and Go).
- *AWS X-Ray daemon* is an application that listens for traffic on User Datagram Protocol (UDP) port 2000, gathers raw segment data, and relays it to the AWS X-Ray API.
- AWS X-Ray displays in the AWS Management Console.

With the AWS SDK, you integrate X-Ray into the application code. The AWS SDK records data about incoming and outgoing requests and sends it to the X-Ray daemon,

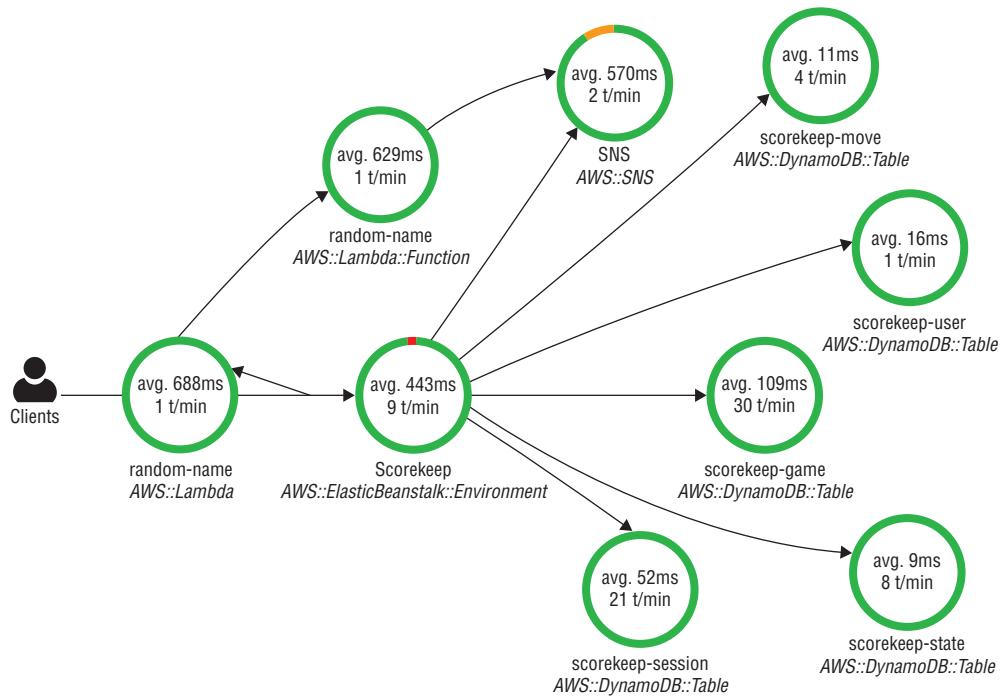
which relays the data in batches to X-Ray. For example, when your application calls Amazon DynamoDB to retrieve user information from an Amazon DynamoDB table, the X-Ray SDK records data from both the client request and the downstream call to Amazon DynamoDB.

When the SDK sends data to the X-Ray daemon, the SDK sends JSON segment documents to a daemon process listening for UDP traffic. The X-Ray daemon buffers segments in a queue and uploads them to X-Ray in batches. The X-Ray daemon is available for Linux, Windows, and macOS, and it is included on both AWS Elastic Beanstalk and AWS Lambda platforms.

When the daemon sends the data to X-Ray, X-Ray uses trace data from the AWS resources that power the cloud applications to generate a detailed service graph. The *service graph* shows the client, your frontend service, and backend services that your frontend service calls to process requests and persist data. Use the service graph to identify bottlenecks, latency spikes, and other issues to improve the performance of your applications.

With X-Ray and the service map, as shown in Figure 12.5, you can visualize how your application is running and troubleshoot any errors.

**FIGURE 12.5** AWS X-Ray service map



# Summary

In this chapter, you learned about serverless compute, explored what it means to use a serverless service, and took an in-depth look at AWS Lambda. With AWS Lambda, you learned how to create a Lambda function with the AWS Management Console and AWS CLI and how to scale Lambda functions by specifying appropriate memory allocation settings and properly defining timeout values. Additionally, you took a closer look at the Lambda function handler, the `event` object, and the `context` object to use data from an event source with AWS Lambda. Finally, you looked at how to invoke Lambda functions by using both the push and pull models and monitor functions. We wrapped up the chapter with a brief look at Amazon CloudWatch and AWS X-Ray.

## Exam Essentials

**Know how to use execution context for reuse.** Take advantage of execution context reuse to improve the performance of your AWS Lambda function. Verify that any externalized configuration or dependencies that your code retrieves are stored and referenced locally after initial execution. Limit the re-initialization of variables or objects on every invocation. Instead, use static initialization/constructor, global/static variables, and singletons. Keep connections (HTTP or database) active, and reuse any that were established during a previous invocation.

**Know how to use environmental variables.** Use environment variables to pass operational parameters to your AWS Lambda function. For example, if you are writing to an Amazon S3 bucket, instead of hardcoding the bucket name to which you are writing, configure the bucket name as an environment variable.

**Know how to control the dependencies in your function's deployment package.** The AWS Lambda execution environment contains libraries, such as the AWS SDK, for the Node.js and Python runtimes. To enable the latest set of features and security updates, AWS Lambda periodically updates these libraries. These updates may introduce subtle changes to the behavior of your AWS Lambda function. Package all your dependencies with your deployment package to have full control of the dependencies that your function uses.

**Know how to minimize your deployment package size to its runtime necessities.**

Minimizing your deployment package size reduces the amount of time that it takes for your deployment package to download and unpack ahead of invocation. For functions authored in Java or .NET Core, it is best to not upload the entire AWS SDK library as part of your deployment package. Instead, select only the modules that include components of the SDK you need, such as Amazon DynamoDB, Amazon S3 SDK modules, and AWS Lambda core libraries.

**Know how memory works.** Performing AWS Lambda function tests is a crucial step to ensure that you choose the optimum memory size configuration. Any increase in memory size triggers an equivalent increase in CPU that is available to your function. The memory usage for your function is determined per invocation, and it displays in the Amazon CloudWatch Logs.

**Know how to load test your AWS Lambda function to determine an optimum timeout value.** It is essential to analyze how long your function runs to determine any problems with a dependency service. Dependency services may increase the concurrency of the function beyond what you expect. This is especially important when your AWS Lambda function makes network calls to resources that may not handle AWS Lambda's scaling.

**Know how permissions for IAM policies work.** Use the most-restrictive permissions when you set AWS IAM policies. Understand the resources and operations that your AWS Lambda function needs, and limit the execution role to these permissions.

**Know how to use AWS Lambda metrics and Amazon CloudWatch alarms.** Use AWS Lambda metrics and Amazon CloudWatch alarms (instead of creating or updating a metric from within your AWS Lambda function code). This is a much more efficient way to track the health of your AWS Lambda functions, and it allows you to catch issues early in the development process. For instance, you can configure an alarm based on the expected duration of your AWS Lambda function execution time to address any bottlenecks or latencies attributable to your function code.

**Know how to capture application errors.** Leverage your log library and AWS Lambda metrics and dimensions to catch application errors, such as ERR, ERROR, and WARNING.

**Know how to create and use dead letter queues (DLQs).** Create and use DLQs to address and replay asynchronous function errors.

## Resources to Review

AWS Lambda Documentation:

<https://aws.amazon.com/documentation/lambda/>

AWS Lambda Developer Guide

<https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>

Invoke AWS Lambda Functions:

<https://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda-functions.html>

Invoke AWS API:

[https://docs.aws.amazon.com/lambda/latest/dg/API\\_Invoke.html](https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html)

CreateFunction API:

[https://docs.aws.amazon.com/lambda/latest/dg/API\\_CreateFunction.html](https://docs.aws.amazon.com/lambda/latest/dg/API_CreateFunction.html)

Function Handler Syntax:

<https://docs.aws.amazon.com/lambda/latest/dg/programming-model-v2.html>

AWS Lambda Pricing:

<https://aws.amazon.com/lambda/pricing/>

AWS Lambda with Amazon CloudWatch Metrics:

<https://docs.aws.amazon.com/lambda/latest/dg/monitoring-functions-access-metrics.html>

AWS Lambda Execution Environment and Libraries:

<https://docs.aws.amazon.com/lambda/latest/dg/current-supported-versions.html>

Python SDK Reference:

<https://boto3.readthedocs.io/en/latest/reference/services/index.html>

Invoke AWS Lambda Functions:

<https://docs.aws.amazon.com/lambda/latest/dg/invoking-lambda-function.html#supported-event-source-dynamo-db>

AWS Lambda Deployment Package:

<https://docs.aws.amazon.com/lambda/latest/dg/deployment-package-v2.html>

AWS Lambda Limits:

<https://docs.aws.amazon.com/lambda/latest/dg/limits.html>

AWS Serverless Application Model (SAM) Local (AWS SAM Local):

<https://github.com/awslabs/aws-sam-cli>

## Exercises



To complete these exercises, download the AWS Certified Developer – Associate Exam code examples for Chapter 12, Chapter12\_Code.zip, from Resources at <http://www.wiley.com/go/sybextestprep>.

For these exercises, you are a developer for a shoe company. The shoe company has a third-party check processor who sends checks, pay stubs, and direct deposits to the shoe company's employees. The third-party service requires a JSON document with the employee's name, the number of hours they worked for the current week, and the employee's hourly rate. Unfortunately, the shoe company's payroll system exports this data only in CSV format. Devise a serverless method to convert the exported CSV file to JSON.

**EXERCISE 12.1****Create an Amazon S3 Bucket for CSV Ingestion**

To solve this, create two Amazon S3 buckets (CSV ingestion and JSON output) and an AWS Lambda function to process the file.

After you export the CSV file, upload the file to Amazon S3. First, create an Amazon S3 bucket with the following Python code:

```
import boto3
# Variables for the bucket name and the region we will be using.
# Important Note: s3 Buckets are globally unique, as such you need to change the
name of the bucket to something else.
# Important Note: If you would like to use us-east-1 as the region, when making
the s3.create_bucket call, then do not specify any region.
bucketName = "shoe-company-2018-ingestion-csv-demo"
bucketRegion = "us-west-1"

# Creates an s3 Resource; this is a higher level API type service for s3.
s3 = boto3.resource('s3')

# Creates a bucket
bucket = s3.create_bucket(ACL='private',Bucket=bucketName,CreateBucketConfiguration
= {'LocationConstraint': bucketRegion})
```

This Python code creates a resource for interacting with the Amazon S3 service. After the resource is created, you can call the function `.create_bucket` to create a bucket.

After executing this Python code, verify that the bucket has been successfully created inside the Amazon S3 console. If it is not successfully created, the most likely cause is that the bucket name is not unique; therefore, renaming the bucket should solve the issue.

---

**EXERCISE 12.2****Create an Amazon S3 Bucket for Final Output JSON**

To create the second bucket for final output, run the following:

```
import boto3
# Variables for the bucket name and the region we will be using.
# Important Note: s3 Buckets are globally unique, as such you need to change the
name of the bucket to something else.
```

---

---

```
# Important Note: If you would like to use us-east-1 as the region, when
# making the s3.create_bucket call, then do not specify any region.

bucketName = "shoe-company-2018-final-json-demo"
bucketRegion = "us-west-1"

# Creates an s3 Resource; this is a higher level API type service for s3.
s3 = boto3.resource('s3')

# Creates a bucket
bucket = s3.create_bucket(ACL='private',Bucket=bucketName,CreateBucketConfiguration={'LocationConstraint': bucketRegion})
```

In the first exercise, you created the initial bucket for ingestion of the .csv file. This bucket will be used for the final JSON output. Again, if you see any errors here, look at the error logs. Verify that the bucket exists inside the Amazon S3 console. You will verify the buckets programmatically in the next exercise.

---

### EXERCISE 12.3

#### Verify List Buckets

To verify the two buckets, use the Python 3 SDK and run the following:

```
import boto3
# Variables for the bucket name and the region we will be using.
# Important Note: Be sure to use the same bucket names you used in the previous
two exercises.
bucketInputName = "shoe-company-2018-ingestion-csv-demo"
bucketOutputName = "shoe-company-2018-final-json-demo"
bucketRegion = "us-west-1"

# Creates an s3 Resource; this is a higher level API type service for s3.
s3 = boto3.resource('s3')

# Get all of the buckets
bucket_iterator = s3.buckets.all()

# Loop through the buckets

for bucket in bucket_iterator:
    if bucket.name == bucketInputName:
        print("Found the input bucket\t:\t", bucket.name)
    if bucket.name == bucketOutputName:
        print("Found the output bucket\t:\t", bucket.name)
```

---

(continued)

**EXERCISE 12.3 (*continued*)**

Here, you are looping through the buckets, and if the two that you created are found, they are displayed. If everything is successful, then you should see output similar to the following:

```
Found the output bucket : shoe-company-2018-final-json-demo  
Found the input bucket  : shoe-company-2018-ingestion-csv-demo
```

---

**EXERCISE 12.4****Prepare the AWS Lambda Function**

To perform the conversion using the AWS CLI and the Python SDK, create the AWS Lambda function. The AWS CLI creates the AWS Lambda function. The Python SDK processes the files inside the AWS Lambda service.

In the following code, change the bucket names to bucket names you defined. The `lambda_handler` function passes the `event` parameter. This allows you to acquire the Amazon S3 bucket name.

Save this code to a file called `lambda_function.py`, and then compress the file.



You can use a descriptive file name; however, remember to update the handler code in Exercise 12.6.

```
import boto3  
import csv  
import json  
import time  
  
# The csv and json modules provide functionality for parsing  
# and writing csv/json files. We can use these modules to  
# quickly perform a data transformation  
# You can read about the csv module here:  
# https://docs.python.org/2/library/csv.html  
# and JSON here:  
# https://docs.python.org/2/library/json.html  
  
# Create an s3 Resource: https://boto3.readthedocs.io/en/latest/guide/resources.html  
s3 = boto3.resource('s3')  
csv_local_file = '/tmp/input-payroll-data.csv'  
json_local_file = '/tmp/output-payroll-data.json'  
  
# Change this value to whatever you named the output s3 bucket in the previous  
exercise
```

---

---

```
output_s3_bucket = 'shoe-company-2018-final-json-demo'

def lambda_handler(event, context):

    # Need to get the bucket name
    bucket_name = event['Records'][0]['s3']['bucket']['name']
    key = event['Records'][0]['s3']['object']['key']

    # Download the file to our AWS Lambda container environment
    try:
        s3.Bucket(bucket_name).download_file(key, csv_local_file)
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they exist and your
              bucket is in the same region as this function.'.format(key, bucket_name))
        raise e

    # Open the csv and json files
    csv_file = open(csv_local_file, 'r')
    json_file = open(json_local_file, 'w')

    # Get a csv DictReader object to convert file to json
    dict_reader = csv.DictReader( csv_file )

    # Create an Employees array for JSON, use json.dumps to pass in the string
    json_conversion = json.dumps({'Employees': [row for row in dict_reader]})

    # Write to our json file
    json_file.write(json_conversion)

    # Close out the files
    csv_file.close()
    json_file.close()

    # Upload finished file to s3 bucket
    try:
        s3.Bucket(output_s3_bucket).upload_file(json_local_file, 'final-output-
payroll.json')
    except Exception as e:
        print(e)
        print('Error uploading object {} to bucket {}. Make sure the file paths
              are correct.'.format(key, bucket_name))
        raise e

    print('Payroll processing completed at: ', time.asctime( time.localtime(time.
time()) ) )
    return 'Payroll conversion from CSV to JSON complete.'
```

---

(continued)

**EXERCISE 12.4 (*continued*)**

After you create the code for the function, upload this file to Amazon S3 with the AWS CLI. This saves the code locally on your desktop and runs the following command. Be sure to compress the file.

```
aws s3 cp lambda_function.zip s3://shoe-company-2018-ingestion-csv-demo
```

If the following command successfully executed, you should see something similar to the following printed to the console:

```
upload: .\lambda_function.zip to s3://shoe-company-2018-ingestion-csv-demo/Lambda_function.zip
```

You may also verify that the file has been uploaded by using the AWS Management Console inside the Amazon S3 service.

---

**EXERCISE 12.5****Create AWS IAM Roles**

In this exercise, create an AWS IAM role so that the AWS Lambda function has the correct permissions to execute the function with the AWS CLI. Create a JSON file of the trust relationship, which allows the AWS Lambda service to assume this particular IAM role with the Security Token Service.

Also create a policy document. A predefined policy document was distributed in the code example that you downloaded in Exercise 12.1. However, if you prefer to create the file manually, you can do so. The following is required for the exercise to work correctly:

```
lambda-trust-policy.json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

---

After the `lambda-trust-policy.json` document has been created, run the following command to create the IAM role:

```
aws iam create-role --role-name PayrollProcessingLambdaRole --description "Provides AWS Lambda with access to s3 and cloudwatch to execute the PayrollProcessing function" --assume-role-policy-document file://lambda-trust-policy.json
```

A JSON object returns. Copy the `RoleName` and `ARN` roles for the next steps.

```
{
    "Role": {
        "AssumeRolePolicyDocument": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Action": "sts:AssumeRole",
                    "Effect": "Allow",
                    "Principal": {
                        "Service": "lambda.amazonaws.com"
                    }
                }
            ]
        },
        "RoleId": "roleidnumber",
        "CreateDate": "2018-05-19T17:30:05.020Z",
        "RoleName": "PayrollProcessingLambdaRole",
        "Path": "/",
        "Arn": "arn:aws:iam::accountnumber:role/PayrollProcessingLambdaRole"
    }
}
```

After you create an AWS role, attach a policy to the role. There are two types of AWS policies: AWS managed and customer managed. AWS creates predefined policies that you can use called *AWS managed policies*. You may create *customer managed policies* specific to your requirements.

For this example, you will use an AWS managed policy built for AWS Lambda called `AWSLambdaExecute`. This provides AWS Lambda access to Amazon CloudWatch Logs and Amazon S3 `GetObject` and `PutObject` API calls.

```
aws iam attach-role-policy --role-name PayrollProcessingLambdaRole --policy-arn arn:aws:iam::aws:policy/AWSLambdaExecute
```

---

(continued)

**EXERCISE 12.5 (*continued*)**

If this command successfully executes, it does not return results. To verify that the IAM role has been properly configured, from the AWS Management Console, go to the IAM service. Click Roles, and search for PayrollProcessingLambdaRole. On the **Permissions** tab, verify that the AWSLambdaExecute policy has been attached. On the **Trust relationships** tab, verify that the trusted entities states the following: “The identify provider(s) lambda.amazonaws.com.”

You have successfully uploaded the Python code that has been compressed to Amazon S3, and you created an IAM role. In the next exercise, you will create the AWS Lambda function.

---

**EXERCISE 12.6****Create the AWS Lambda Function**

In this exercise, create the AWS Lambda function. You can view the AWS Lambda API reference here:

<https://docs.aws.amazon.com/cli/latest/reference/lambda/index.html>



For the --handler parameter, make sure that you specify the name of the .py file you created in Exercise 12.4. Also, make sure that for the S3Key parameter, you specify the name of the compressed file inside the Amazon S3 bucket.

Run this AWS CLI command:

```
aws lambda create-function --function-name PayrollProcessing --runtime python3.7  
--role arn:aws:iam::accountnumber:role/PayrollProcessingLambdaRole --handler  
lambda_function.lambda_handler --description "Converts Payroll CSVs to JSON  
and puts the results in an s3 bucket." --timeout 3 --memory-size 128 --code  
S3Bucket=shoe-company-2018-ingestion-csv-demo,S3Key=lambda_function.zip --tags  
Environment="Production",Application="Payroll" --region us-west-1
```

If the command was successful, you receive a JSON response similar to the following:

```
{  
    "FunctionName": "PayrollProcessing",  
    "FunctionArn": "arn:aws:lambda:us-east-2:accountnumber:function:  
    PayrollProcessing",  
    "Runtime": "python3.7",  
    "Role": "arn:aws:iam::accountnumber:role/PayrollProcessingLambdaRole",  
    "Handler": "payroll.lambda_handler",  
    "CodeSize": 1123,  
    "Description": "Converts Payroll CSVs to JSON and puts the results in an s3  
    bucket.",
```

---

```
"Timeout": 3,
"MemorySize": 128,
"LastModified": "2018-12-10T06:36:27.990+0000",
"CodeSha256": "NUKm2kp/fLzVr58t8XCTw6YGBmxR2E1Q9MHuW11QXfw=",
"Version": "$LATEST",
"TracingConfig": {
    "Mode": "PassThrough"
},
"RevisionId": "ae30524f-26a9-426a-b43a-efa522cb1545"
}
```

You have successfully created the AWS Lambda function. You can verify this from the AWS Management Console and opening the AWS Lambda console.

---

## EXERCISE 12.7

### Give Amazon S3 Permission to Invoke an AWS Lambda Function

In this exercise, use the AWS Lambda CLI add-permission command to invoke the AWS Lambda function.

```
aws lambda add-permission --function-name PayrollProcessing --statement-id lambdas3permission --action lambda:InvokeFunction --principal s3.amazonaws.com --source-arn arn:aws:s3:::shoe-company-2018-ingestion-csv-demo --source-account yourawsaccountnumber --region us-west-1
```

After you run this command and it is successful, you should receive a JSON response that looks similar to the following:

```
{
    "Statement": {
        "Sid": "lambdas3permission",
        "Effect": "Allow",
        "Principal": {
            "Service": "s3.amazonaws.com"
        },
        "Action": "lambda:InvokeFunction",
        "Resource": "arn:aws:lambda:us-east-2:accountnumber:function:PayrollProcessing",
        "Condition": {
            "StringEquals": {
                "AWS:SourceAccount": "accountnumber"
            },
        }
    }
}
```

---

(continued)

**EXERCISE 12.7 (continued)**

```
"ArnLike": {  
    "AWS:SourceArn": "arn:aws:s3:::shoe-company-2018-ingestion-csv-demo"  
}  
}  
}  
}
```

This provides a function policy to the AWS Lambda function that allows the S3 bucket that you created to call the action `lambda:InvokeFunction`. You can verify this by navigating to the AWS Lambda service inside the AWS Management Console. In the **Designer** section, click the key icon to view permissions, and under **Function policy**, you will see the policy you just created.

---

**EXERCISE 12.8****Add the Amazon S3 Event Trigger**

In this exercise, add the trigger for Amazon S3 using AWS CLI for the `s3api` commands. The `notification-config.json` file was provided in the exercise files. Its contents are as follows:

```
{  
    "LambdaFunctionConfigurations": [  
        {  
            "Id": "s3PayrollFunctionObjectCreation",  
            "LambdaFunctionArn": "arn:aws:lambda:us-west-1:accountnumber:function:  
PayrollProcessing",  
            "Events": [  
                "s3:ObjectCreated:*"  
            ],  
            "Filter": {  
                "Key": {  
                    "FilterRules": [  
                        {  
                            "Name": "suffix",  
                            "Value": ".csv"  
                        }  
                    ]  
                }  
            }  
        }  
    ]  
}
```

---

```
        }
    }
]
}

aws s3api put-bucket-notification-configuration --bucket shoe-company-2018-ingestion-csv-demo --notification-configuration file://notification-config.json
```

If the execution is successful, no response is sent. To verify that the trigger has been added to the AWS Lambda function, navigate to the AWS Lambda console inside the AWS Management Console, and verify that there is now an Amazon S3 trigger.

---

### EXERCISE 12.9

#### Test the AWS Lambda Function

To test the AWS Lambda function, use the AWS CLI to upload the CSV file to the Amazon S3 bucket; then check whether the function transforms the data and puts the result file in the output bucket.

```
aws s3 cp input-payroll-data.csv s3://shoe-company-2018-ingestion-csv-demo
```

If everything executes successfully, in the output bucket that you created, you should see the transformed JSON file. You accepted input into one Amazon S3 bucket as a .csv, transformed it to serverless by using AWS Lambda, and then stored the resulting .json file in a separate Amazon S3 bucket. If you do not see the file, retrace your steps through the exercises. It is a good idea to view the Amazon CloudWatch Logs, which can be found on the **Monitoring** tab in the AWS Lambda console. This way, you can determine whether there are any errors.

---

## Review Questions

1. A company currently uses a serverless web application stack, which consists of Amazon API Gateway, Amazon Simple Storage Service (Amazon S3), Amazon DynamoDB, and AWS Lambda. They would like to make improvements to their AWS Lambda functions but do not want to impact their production functions.

How can they accomplish this?

- A. Create new AWS Lambda functions with a different name, and update resources to point to the new functions when they are ready to test.
  - B. Copy their AWS Lambda function to a new region where they can update their resources to the new region when ready.
  - C. Create a new AWS account, and re-create all their serverless infrastructure for their application testing.
  - D. Publish the current version of their AWS Lambda function, and create an alias as PROD. Then, assign PROD to the current version number, update resources with the PROD alias ARN, and create a new version of the updated AWS Lambda function and assign an alias of \$DEV.
2. What is the maximum amount of memory that you can assign an AWS Lambda function?
    - A. AWS runs the AWS Lambda function; it is a managed service, so you do not need to configure memory settings.
    - B. 3008 MB
    - C. 1000 MB
    - D. 9008 MB
  3. What is the default timeout value for an AWS Lambda function?
    - A. 3 seconds
    - B. 10 seconds
    - C. 15 seconds
    - D. 25 seconds
  4. A company uses a third-party service to send checks to its employees for payroll. The company is required to send the third-party service a JSON file with the person's name and the check amount. The company's internal payroll application supports exporting only to CSVs, and it currently has *cron* jobs set up on their internal network to process these files. The server that is processing the data is aging, and the company is concerned that it might fail in the future. It is also looking to have the AWS services perform the payroll function.  
What would be the best serverless option to accomplish this goal?
    - A. Create an Amazon Elastic Compute Cloud (Amazon EC2) and the necessary *cron* job to process the file from CSV to JSON.
    - B. Use AWS Import/Export to create a virtual machine (VM) image of the on-premises server and upload the Amazon Machine Images (AMI) to AWS.

- C. Use AWS Lambda to process the file with Amazon Simple Storage Service (Amazon S3).
  - D. There is no way to process this file with AWS.
5. What is the maximum execution time allowed for an AWS Lambda function?
- A. 60 seconds
  - B. 120 seconds
  - C. 230 seconds
  - D. 300 seconds
6. Which language is *not* supported for AWS Lambda functions?
- A. Ruby
  - B. Python 3.6
  - C. Node.js
  - D. C# (.NET Core)
7. How can you increase the limit of AWS Lambda concurrent executions?
- A. Use the Support Center page in the AWS Management Console to open a case and send a Server Limit Increase request.
  - B. AWS Lambda does not have any limits for concurrent executions.
  - C. Send an email to [limits@amazon.com](mailto:limits@amazon.com) with the subject “AWS Lambda Increase.”
  - D. You cannot increase concurrent executions for AWS Lambda.
8. A company is receiving *permission denied* after its AWS Lambda function is invoked and executes and has a valid trust policy. After investigating, the company realizes that its AWS Lambda function does not have access to download objects from Amazon Simple Storage Service (Amazon S3).  
Which type of policy do you need to correct to give access to the AWS Lambda function?
- A. Function policy
  - B. Trust policy
  - C. Execution policy
  - D. None of the above
9. A company wants to be able to send event payloads to an Amazon Simple Queue Service (Amazon SQS) queue if the AWS Lambda function fails.  
Which of the following configuration options does the company need to be able to do this in AWS Lambda?
- A. Enable a dead-letter queue.
  - B. Define an Amazon Virtual Private Cloud (Amazon VPC) network.
  - C. Enable concurrency.
  - D. AWS Lambda does not support such a feature.

- 10.** A company wants to be able to pass configuration settings as variables to their AWS Lambda function at execution time.

Which feature should the company use?

- A.** Dead-letter queues
- B.** AWS Lambda does not support such a feature.
- C.** Environment variables
- D.** None of the above

# Chapter 13



AWS® Certified Developer Official Study Guide  
By Nick Alteen, Jennifer Fisher, Casey Gerena, Wes Gruver, Asim Jalil,  
Heiward Osman, Marife Pagan, Santosh Patlolla and Michael Roth  
Copyright © 2019 by Amazon Web Services, Inc.

# Serverless Applications

---

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

#### Domain 1: Deployment

- ✓ 1.4 Deploy serverless applications.

#### Domain 2: Security

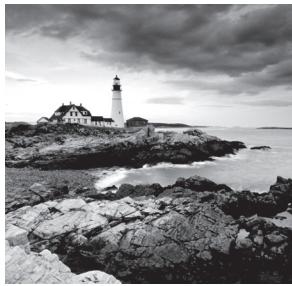
- ✓ 2.1 Make authenticated calls to AWS Services.
- ✓ 2.3 Implement application authentication and authorization.

#### Domain 3: Development with AWS Services

- ✓ 3.1 Write code for serverless applications.
- ✓ 3.2 Translate functional requirements into application design.
- ✓ 3.3 Implement application design into application code.
- ✓ 3.3 Write code that interacts with AWS Services by using APIs, SDKs, and AWS CLI.

#### Domain 5: Monitoring and Troubleshooting

- ✓ 5.1 Write code that you can monitor.



## Introduction to Serverless Applications

In the previous chapter, you learned about AWS Lambda and how you can write functions that run in a serverless manner. A serverless application is typically a combination of AWS Lambda and other Amazon services. You build serverless applications to allow developers to focus on their core product instead of the need to manage and operate servers or runtimes in the cloud or on-premises. This reduces overhead and lets developers reclaim time and energy that can be better spent developing reliable, scalable products and new features for applications.

Serverless applications have the following three main benefits:

- No server management
- Flexible scaling
- 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 Elastic Compute Cloud (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. You do not need to architect for these capabilities, as the services that run the application provide them by default. Additionally, when periods of low traffic occur in the web application, you do not spend money on Amazon EC2 instances that do not run at their full capacity.

## Web Server with Amazon Simple Storage Service (Presentation Tier)

*Amazon Simple Storage Service* (Amazon S3) can store HTML, CSS, images, and JavaScript files within an Amazon S3 bucket, and can host the website like a traditional web server. Though Amazon S3 hosts static websites, today many websites are dynamic applications,

where you can use JavaScript to create HTTP requests. These HTTP requests are sent to a Representational State Transfer (REST) endpoint service called *Amazon API Gateway*, which allows the application to save and retrieve data dynamically.

Amazon API Gateway opens up a variety of application tier possibilities. An internet-accessible HTTPS API can be consumed by any client capable of HTTPS communication. Some common presentation tier examples that you could use for your application's include the following:

**Mobile app** Not only can you integrate with custom business logic via Amazon API Gateway and AWS Lambda, you can use Amazon Cognito to create and manage user identities.

**Static website content hosted in Amazon S3** You can enable your Amazon API Gateway APIs to be cross-origin resource sharing-compliant. This allows web browsers to invoke your APIs directly from within the static web pages.

**Any other HTTPS-enabled client device** Many devices can connect and communicate via HTTPS. There is nothing unique or proprietary about how clients communicate with the APIs that you create with the Amazon API Gateway service; it is pure HTTPS. No specific client software or licenses are required.

Additionally, there are several JavaScript frameworks that are widely available today, such as Angular and React, which allow you to benefit from a *Model-View-Controller* (MVC) architecture.

## Amazon S3 Static Website



For the remainder of this chapter, the example bucket's name is examplebucket. This is for illustration purposes only, as Amazon S3 bucket names must be globally unique.

To create an Amazon S3 static website, you first need to create a bucket. Name the bucket something meaningful, such as examplebucket. When you use virtual hosted-style buckets with Secure Sockets Layer (SSL), the SSL wildcard certificate only matches buckets that do *not* contain periods. To work around this, use HTTP or write your own certificate verification logic. AWS recommends that you do not use periods (.) in bucket names when using virtual hosted-style buckets with SSL.

After you create your Amazon S3 bucket, you must enable and configure it to use static website hosting, index document, error document, and redirection rules (optional) in the AWS Management Console > Amazon S3 Service. Use this examplebucket bucket to host a website.

The Amazon S3 bucket includes the region based on latency, cost, and regulatory requirements. Each object has a unique key. You grant permissions at the object or bucket level.

For the index document, enter the name of your home page's HTML file (typically index.html). Additionally, you may load a custom error page such as error.html. As with

many of the Amazon services, you can make these changes with the AWS Command Line Interface (AWS CLI) or in an AWS software development kit (AWS SDK). To enable this option with the AWS CLI, run the following command:

```
aws s3 website s3://examplebucket/ --index-document index.html --error-document error.html
```

After you enable the Amazon S3 static website hosting feature, enter an endpoint that reflects your AWS Region: examplebucket.s3-website.amazonaws.com.

## Configuring Web Traffic Logs

Amazon S3 allows you to log and capture information such as the number of visitors who access your website. To enable logs, create a new Amazon S3 bucket to store your logs. This excludes your log files from the website-hosting bucket. You can create a logs-examplebucket-com bucket, and inside of that bucket, you can create a folder you call logs/ (or any name you choose). Use this folder to store all of your logs.



The term *folder* is used to describe logs; however, the Amazon S3 data model is a flat structure that allows you to create a bucket, and the bucket stores objects. There is no hierarchy of sub-buckets or subfolders; nevertheless, you can infer a logical hierarchy using key name prefixes and delimiters as the Amazon S3 console does. In other words, you should know that, as a developer, there is technically no such thing as an Amazon S3 folder—it is simply a key.

Now you use Amazon S3 to enable the static website-hosted bucket called examplebucket to log files. You can configure the target bucket for the log files at logs-examplebucket-com, and you can create a target prefix to send log files to a particular prefix key only.

To enable this feature with the AWS CLI, create an access control list that provides access to the log files that you want to create and then apply the logging policy. Here's an example:

```
aws s3api put-bucket-acl --bucket examplebucket --grant-write  
'URI="http://acs.amazonaws.com.cn/groups/s3/LogDelivery"' --grant-read-acp  
'URI="http://acs.amazonaws.com.cn/groups/s3/LogDelivery"'
```

```
aws s3api put-bucket-logging --bucket examplebucket --bucket-logging-status  
file://logging.json
```

Here's the file logging.json:

```
{  
    "LoggingEnabled": {  
        "TargetBucket": "examplebucket",  
        "TargetPrefix": "logs/"  
    }  
}
```

```
"TargetPrefix": "examplebucket/",  
"TargetGrants": [  
    {  
        "Grantee": {  
            "Type": "AmazonCustomerByEmail",  
            "EmailAddress": "user@example.com"  
        },  
        "Permission": "FULL_CONTROL"  
    },  
    {  
        "Grantee": {  
            "Type": "Group",  
            "URI": "http://acs.amazonaws.com/groups/global/AllUsers"  
        },  
        "Permission": "READ"  
    }  
]
```

## Creating Custom Domain Name with Amazon Route 53

Amazon Route 53 is a highly available and scalable cloud Domain Name System (DNS) web service. It is designed to give developers and businesses an extremely reliable and cost-effective way to route end users to internet applications by translating names like `www.example.com` into the numeric IP addresses like `192.0.2.1` that computers use to connect to each other. Amazon Route 53 is fully compliant with IPv6 as well.

You may not want to use the Amazon S3 endpoint such as `bucket-name.s3-website-region.amazonaws.com`. Instead, you may want a more user-friendly URL such as `myexamplewebsite.com`. To accomplish this, purchase a domain name with Amazon Route 53.



You can purchase your domain from another provider and then update the name servers to use Amazon Route 53.

Amazon Route 53 effectively connects user requests to infrastructure running in AWS—such as Amazon EC2 instances, Elastic Load Balancing (ELB) load balancers, or Amazon S3 buckets—and can route users to infrastructure outside of AWS. You can use Amazon

Route 53 to configure DNS health checks to route traffic to healthy endpoints or to monitor independently the health of your application and its endpoints. Amazon Route 53 Traffic Flow makes it easy for you to manage traffic globally through a variety of routing types, including latency-based routing, geolocation, geoproximity, and weighted round-robin, all of which can be combined with DNS failover to enable a variety of low-latency, fault-tolerant architectures.

Using Amazon Route 53 Traffic Flow's simple visual editor, you can easily manage how your end users are routed to your application's endpoints—whether in a single AWS Region or distributed around the globe. Amazon Route 53 also offers domain name registration. You can purchase and manage domain names such as example.com, and Amazon Route 53 will automatically configure DNS settings for your domains.

## Speeding Up Content Delivery with Amazon CloudFront

*Latency* is an increasingly important aspect when you deliver web applications to the end user, as you always want your end user to have an efficient, low-latency experience on your website. Increased latency can result in both decreased customer satisfaction and decreased sales. One way to decrease latency is to use *Amazon CloudFront* to move your content closer to your end users. Amazon CloudFront has two delivery methods to deliver content. The first is a web distribution, and this is for storing of .html, .css, and graphic files. Amazon CloudFront also provides the ability to have an RTMP distribution, which speeds up distribution of your streaming media files using Adobe Flash Media Server's RTMP protocol. An RTMP distribution allows an end user to begin playing a media file before the file has finished downloading from a CloudFront edge location.

To use Amazon CloudFront with your Amazon S3 static website, perform these tasks:

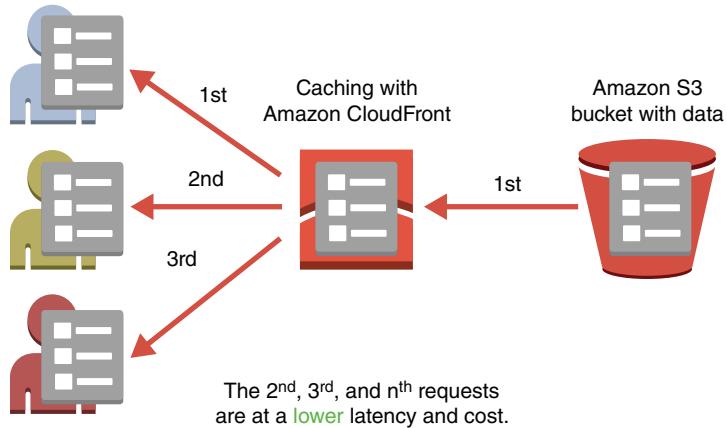
1. Choose a delivery method.

In the example, Amazon S3 is used to store a static web page; thus, you will be using the Web delivery method. However, as mentioned previously, you could also use RTMP for streaming media files.

2. Specify the cache behavior. A cache behavior lets you configure a variety of CloudFront functionality for a given URL path pattern for files on your website.
3. Choose the distribution settings and network that you want to use. For example, you can use all edge locations or only U.S., Canada, and Europe locations.

Amazon CloudFront enables you to cache your data to minimize redundant data-retrieval operations. Amazon CloudFront reduces the number of requests to which your origin server must respond directly. This reduces the load on your origin server and reduces latency because more objects are served from Amazon CloudFront edge locations, which are closer to your users.

The Amazon S3 bucket pushes the first request to Amazon CloudFront's cache. The second, third, and  $n^{\text{th}}$  requests pull from the Amazon CloudFront's cache at a lower latency and cost, as shown in Figure 13.1.

**FIGURE 13.1** Amazon CloudFront cache

The more requests that Amazon CloudFront is able to serve from edge caches as a proportion of all requests (that is, the greater the cache hit ratio), the fewer viewer requests that Amazon CloudFront needs to forward to your origin to get the latest version or a unique version of an object. You can view the percentage of viewer requests that are hits, misses, and errors in the Amazon CloudFront console.

A number of factors affect the cache hit ratio. You can adjust your Amazon CloudFront distribution configuration to improve the cache hit ratio.

Use Amazon CloudFront with Amazon S3 to improve your performance, decrease your application's latency and costs, and provide a better user experience. Amazon CloudFront is also a serverless service, and it fits well with serverless stack services, especially when you use it in conjunction with Amazon S3.

## Dynamic Data with Amazon API Gateway (Logic or App Tier)

This section details how to use dynamic data with the Amazon API Gateway service in a logic tier or app tier.

*Amazon API Gateway* is a fully managed, serverless AWS service, with no server that runs inside your environment to define, deploy, monitor, maintain, and secure APIs at any scale. Clients integrate with the APIs that use standard HTTPS requests. Amazon API Gateway can integrate with a service-oriented multitier architecture with Amazon services,

such as AWS Lambda and Amazon EC2. It also has specific features and qualities that make it a powerful edge for your logic tier. You can use these features and qualities to enhance and build your dynamic web application.

The Amazon API Gateway integration strategy that provides access to your code includes the following:

**Control service** Uses REST to provide access to Amazon services, such as AWS Lambda, Amazon Kinesis, Amazon S3, and Amazon DynamoDB. The access methods include the following:

- Consoles
- CLI
- SDKs
- REST API requests and responses

**Execution service** Uses standard HTTP protocols or language-specific SDKs to deploy API access to backend functionality.



Do not directly expose resources or the API—always use AWS edge services and the Amazon API Gateway service to safeguard your resources and APIs.

## Endpoints

There are three types of *endpoints* for Amazon API Gateway.

**Regional endpoints** Live inside the AWS Region, such as us-west-2.

**Edge optimized endpoints** Use Amazon CloudFront, a content delivery web service with the AWS global network of edge locations as connection points for clients, and integrate with your API.

**Private endpoints** Can live only inside of a *virtual private cloud* (VPC).

You use Amazon API Gateway to help drive down the total response time latency of your API. You can improve the performance of specific API requests with Amazon API Gateway to store responses in an optional in-memory cache. This not only provides performance benefits for API requests that repeat, but it also reduces backend executions, which helps to reduce overall costs.

The API endpoint can be a default host name or a custom domain name. The default host name is as follows:

{api-id}.execute-api.{region}.amazonaws.com

## Resources

Amazon API Gateway consists of resources and methods. A *resource* is an object that provides operations you use to interact with HTTP commands such as GET, POST, or DELETE. If you combine a resource path with a specific operation on a resource, you create a *method*. Users can call API methods to obtain controlled access to resources and to receive a response. You define *mappings* between the method and the backend to maintain control. If the frontend payload does not match the corresponding backend payload, you can create mapping templates to enable them to communicate and return a response.

Before you can interact with a resource, you use a *model* to describe the data format for the request or response. You use the model with the AWS SDK for an API to validate data and generate a mapping template. Models save time and money, as they reduce the likelihood that your API will experience security and reliability issues.

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*. In the Amazon API Gateway console, this base URL is referred to as the *Invoke URL*, and it displays in the API's stage editor after the API deploys.

If you own a pizza restaurant and run a website to display your menu options, you can create a *root resource* called *menu* (for production) that results in /menu with a GET method and returns the JSON values for your entire menu. When individuals visit your website and navigate to the menu, you can return all of this data.

For example, the following:

```
{api-id}.execute-api.region.amazonaws.com/menu
```

will return the dataset through the Amazon API Gateway service:

```
[  
 {  
   "id": 1,  
   "menu-item": "cheese pizza",  
   "price": "14.99"  
,  
 {  
   "id": 2,  
   "menu-item": "pepperoni pizza",  
   "price": "17.99"  
 }  
]
```

With resources, you create paths such as /menu, /specials, and /orders to pull different datasets with HTTP methods.

## HTTP Methods

The *Internet Engineering Task Force* (IETF) is responsible for developing and documenting the HTTP protocol and how it operates. Amazon API Gateway uses the HTTP protocol to process these HTTP methods. Amazon API Gateway supports the following methods:

- GET
- HEAD
- POST
- PUT
- PATCH
- OPTIONS
- DELETE

These methods send and receive data to and from the backend. Serverless data can be sent to AWS Lambda to process.

## Stages

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. For example, stage settings enable caching, customize request throttling, configure logging, define stage variables, or attach a canary release to test. A *canary release* is a software deployment strategy in which a new version of an API is deployed at the same time that the original base version remains deployed as a production release. This means that in a canary deployment, you will have the majority of your traffic route to the current production environment and will have a small portion of your traffic route to the canary environment for testing purposes.

When you create a stage, your API is considered deployed and accessible to whomever you grant access. An advisable API strategy is to create stages for each of your environments such as DEV, TEST, and PROD, so that you can continue to develop and update your API and applications without affecting production.

## Authorizers

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 and only allow users to whom you have granted specific access to your API.



You have a customer relationship management application, and you only want certain users to be able to modify customer data. With authorizers, you create an API and restrict who can call that API with an authorizer in conjunction with AWS Lambda or Amazon Cognito.

## API Keys

With the Amazon API Gateway service, you can generate *API keys* to provide access to your API for external users, use them to sell to your customer base, and use the API call `apikey:create` to create an API key.

## Cross-Origin Resource Sharing

*Cross-origin resource sharing* (CORS) remedies the inability of a client-side web application that runs on one server to be retrieved from another service. This remedy is called a *same-origin policy*, and primarily it prevents malicious actors from calling your APIs from different servers and creates a *denial of service* for your endpoint. While you implement CORS, you still need servers to exchange data for valid reasons, such as to deliver APIs to different users, clients, or customers. You can read the specification at <https://www.w3.org/TR/cors/>.

CORS allows you to set certain HTTP headers to enable cross-origin access to call APIs or services to which you need access. The HTTP headers include the following:

- `Access-Control-Allow-Origin`
- `Access-Control-Allow-Credentials`
- `Access-Control-Allow-Headers`
- `Access-Control-Allow-Methods`
- `Access-Control-Expose-Headers`
- `Access-Control-Max-Age`
- `Access-Control-Request-Headers`
- `Access-Control-Request-Method`
- `Origin`

To use Amazon API Gateway, you must enable the CORS *resource* inside of the Amazon API Gateway console so that your web application makes calls to the Amazon API Gateway service successfully. Without CORS, any calls made to the Amazon API Gateway service will fail.

## Integrating with AWS Lambda

With Amazon API Gateway, you can build *RESTful APIs* without the need to manage a server. Amazon API Gateway gives your application a simple way (HTTPS requests) to leverage the innovation of AWS Lambda directly. Amazon API Gateway forms the bridge that connects your presentation tier and the functions you write in AWS Lambda. After defining the client/server relationship with your API, the contents of the client's HTTPS request can be passed to AWS Lambda for execution, where you can write a function to talk to your database tier. For example, once someone accesses the API endpoint, contents of the request—which includes the request metadata, request headers, and the request body—can be passed to AWS Lambda. This then allows AWS Lambda to request dynamic data from your database tier—for example, Amazon DynamoDB.

## Monitoring Amazon API Gateway with Amazon CloudWatch

Amazon API Gateway also integrates with *Amazon CloudWatch*. Amazon CloudWatch provides preconfigured metrics to help you monitor your APIs and build both dashboards and alarms. At the time of this writing, there are nine metrics available by default with Amazon CloudWatch, as shown in Table 13.1.

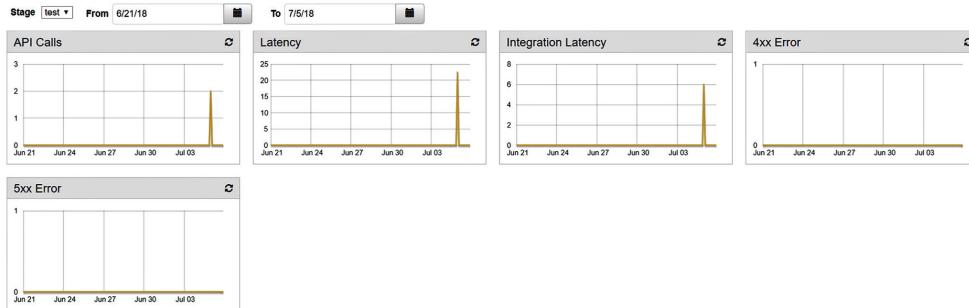
**TABLE 13.1** Amazon CloudWatch Metrics

Metric	Description
4XXError	The number of client-side errors captured in a specified period. The Sum statistic represents this metric, namely, the total count of the 4XXError errors in the given period. The Average statistic represents the 4XXError error rate, namely, the total count of the 4XXError errors divided by the total number of requests during the period. The denominator corresponds to the Count metric.  Unit: Count
5XXError	The number of server-side errors captured in a given period. The Sum statistic represents this metric, namely, the total count of the 5XXError errors in the given period. The Average statistic represents the 5XXError error rate, namely, the total count of the 5XXError errors divided by the total number of requests during the period. The denominator corresponds to the Count metric.  Unit: Count
CacheHitCount	The number of requests served from the API cache in a given period. The Sum statistic represents this metric, namely, the total count of the cache hits in the specified period. The Average statistic represents the cache hit rate, namely, the total count of the cache hits divided by the total number of requests during the period. The denominator corresponds to the Count metric.  Unit: Count
CacheMissCount	The number of requests served from the backend in a given period, when API caching is enabled. The Sum statistic represents this metric, namely, the total count of the cache misses in the specified period. The Average statistic represents the cache miss rate, namely, the total count of the cache hits divided by the total number of requests during the period. The denominator corresponds to the Count metric.  Unit: Count

Metric	Description
Count	The total number API requests in a given period. The Sample-Count statistic represents this metric. Unit: Count
IntegrationLatency	The time between when Amazon API Gateway relays a request to the backend and when it receives a response from the backend. Unit: Millisecond
Latency	The time between when Amazon API Gateway receives a request from a client and when it returns a response to the client. The latency includes the integration latency and other Amazon API Gateway overhead. Unit: Millisecond

See Figure 13.2 for a sample dashboard.

**FIGURE 13.2** Sample dashboard for Amazon API Gateway using Amazon CloudWatch



If you use Amazon CloudWatch with Amazon API Gateway, you can monitor your application from an API standpoint to see whether any issues occur as the application is being used. Particularly, you can view metrics such as CacheMissCount and Latency.

## Other Notable Features

Amazon API Gateway has several notable features.

**Security** Amazon API Gateway exposes HTTPS endpoints only. AWS recommends that you use IAM roles and policies to secure access to the backend, but you can use Lambda authorizers too. The administrator-managed policy is `AmazonAPIGatewayAdministrator`.

**Definition support** The OpenAPI Specification, formerly known as Swagger Specification, is used to define a RESTful interface. If you create a document that conforms to the OpenAPI Specification, you can upload it to Amazon API Gateway to have it create your desired API endpoint. For more information on the OpenAPI Specification you can visit <https://swagger.io/specification>.

**Free tier** Amazon API Gateway has a free tier, and it allows one million API receive calls per month, for free, for the first 12 months.

## User Authentication with Amazon Cognito

A crucial aspect of building web applications is *user authentication*. Nearly every web application today has a user authentication *system*. From banking websites to social media websites, user authentication is a critical component to secure your web and mobile applications. *Amazon Cognito* allows for simple and secure user sign-up, sign-in, and access control mechanisms designed to handle web application authentication.

Amazon Cognito includes the following features:

- Amazon Cognito user pools, which are secure and scalable user directories
- Amazon Cognito identity pools (federated identities), which offer social and enterprise identity federation
- Standards-based Web Identity Federation Authentication through Open Authorization (OAuth) 2.0, Security Assertion Markup Language (SAML) 2.0, and OpenID Connect (OIDC) support
- Multi-factor authentication
- Encryption for data at rest and data in transit
- Access control with AWS Identity and Access Management (IAM) integration
- Easy application integration (prebuilt user interface)
- iOS Object C, Android, iOS Swift, and JavaScript
- Adherence to compliance requirements such as Payment Card Industry Data Security Standard (PCI DSS)

### Amazon Cognito User Pools

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. Whether your users sign in directly or

through a third party, all members of the user pool have a directory profile that you can access through an SDK.

User pools provide the following:

- Sign-up and sign-in services
- A built-in, customizable web user interface (UI) to sign in users
- Social sign-in with Facebook, Google, and Amazon, and sign-in with Security Assertion Markup Language (SAML) identity providers from your user pool
- User directory management and user profiles
- Security features, such as *multi-factor authentication* (MFA), check for compromised credentials, account takeover protection, and phone and email verification
- Customized workflows and user migration through AWS Lambda triggers

After successfully authenticating a user, Amazon Cognito issues *JSON Web Tokens* (JWT) that you can use to secure and authorize access to your own APIs or exchange them for AWS credentials.

With Amazon Cognito, you can choose how you want your users to sign in: with a username, an email address, and/or a phone number. Additionally, user pools allow you to select *attributes*. Attributes are properties that you want to store about your end users, with standard attributes that are created for you, if you enable the option. You can also develop custom attributes.

The standard attributes are as follows:

- address
- birthdate
- email
- family name
- gender
- given name
- locale
- middle name
- name
- nickname
- phone number
- picture
- preferred username
- profile
- zoneinfo
- updated at
- website

## Password Policies

In addition to attributes, you can configure *password policies*. You can set the minimum password length and require specific character types, including uppercase letters and lowercase letters. Furthermore, you can either allow users to sign up and enroll themselves or allow only administrators to create users. If administrators create the account, you can also set the account to expire if it remains unused for a specified period of time.

## Multi-factor Authentication

*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. With Amazon Cognito, you can enable multi-factor authentication to secure your application further. To enable this option with Amazon Cognito, create a role that enables Amazon Cognito to send Short Message Service (SMS) messages to users.

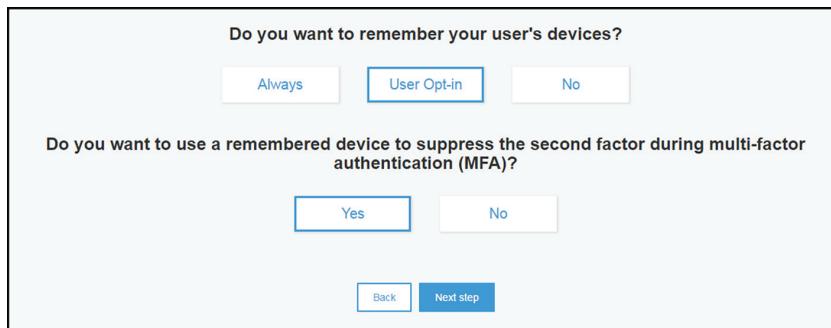
Besides MFA, you can customize your SMS verification messages, email verification messages, and user invitation messages. For example, you could send your end users a welcome message when they verify their account.

## Device Tracking and Remembering

If you enable multi-factor authentication, this increases the security of an application to require a second authentication challenge from the user. However, this does require a new two-factor sign-in after a prolonged absence of activity, even when the user device has not been signed out or shut off.

With device tracking and remembering, you can save that user's device and remember it so that they do not have to provide a token again, as the application has already seen this specific device. Figure 13.3 shows how to enable this feature.

**FIGURE 13.3** Device tracking



The specifics of the configuration terminology include the following:

**Tracked** A *tracked device* is assigned a set of device credentials and consists of a *key* and *secret key pair*. You can view all tracked devices for a specific user on the Users screen of the Amazon Cognito console. In addition, you can view the devices metadata (whether it is remembered, the time it began being tracked, the last authenticated time, and such) and the devices usage.

**Remembered** A *remembered device* is also tracked. During user authentication, the key and secret pair assigned to a remembered device authenticates the device to verify that it is the same device that the user previously used to sign in to the application. You can view remembered devices in the Amazon Cognito console.

**Not remembered** A *not-remembered device*, while still tracked, is treated as if it was never used during the user authentication flow. The device credentials are not used to authenticate the device. The new APIs in the *AWS Mobile SDK* do not expose these devices, but you can see them in the Amazon Cognito console.

The first configuration setting reads “Do you want to remember devices?” and has the following options:

**No (the default)** Devices are neither remembered nor tracked.

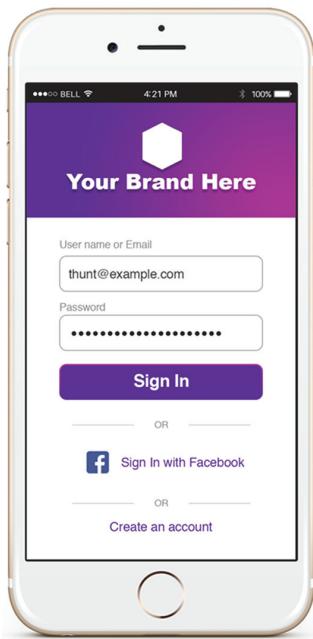
**Always** Every device used with your application is remembered.

**User opt-in** The user’s device is remembered only if that user opts to remember the device. This enables your users to decide whether your application should remember the devices they use to sign in, though all devices are tracked regardless of which setting they choose. This is a useful option when you require a higher security level, but the user may sign in from a shared device. For example, if a user signs in to a banking application from a public computer at a library, the user requires the option to decide whether their device is to be remembered.

The second configuration option is “Do you want to use a remembered device to suppress the second factor during multi-factor authentication (MFA)?” It appears when you select either Always or User Opt-In for the first configuration option. The second factor suppression option enables your application to use a remembered device as a second factor of authentication, and it suppresses the SMS-based challenge in the MFA flow. This feature works together with MFA, and it requires MFA to be enabled for the user pool. The device must first be remembered before it can be used to suppress the SMS-based challenge. Upon the initial sign-in with a new device, the user must complete the SMS challenge. Afterward, the user no longer has to complete the SMS challenge.

## User Interface Customization

An Amazon Cognito user pool includes a prebuilt *user interface* (UI) that you can use inside of your application to build a user authentication flow quickly, as shown in Figure 13.4.

**FIGURE 13.4** Amazon Cognito prebuilt UI

You can modify the UI with the AWS Management Console, the AWS CLI, or the API. You can also upload your own custom logo with a maximum file size of 100 KB. The CSS *classes* you can customize in the prebuilt UI are as follows:

- background-customizable
- banner-customizable
- errorMessage-customizable
- idpButton-customizable
- idpButton-customizable:hover
- inputField-customizable
- inputField-customizable:focus
- label-customizable
- legalText-customizable
- logo-customizable
- submitButton-customizable
- submitButton-customizable:hover
- textDescription-customizable

You can customize the UI and CLI with two commands: `get-ui-customization` to retrieve the customization settings and `set-ui-customization` to set the UI customization, as shown in the following example code:

```
aws cognito-idp get-ui-customization
aws cognito-idp set-ui-customization --user-pool-id <your-user-pool-id>
--client-id <your-app-client-id> --image-file <path-to-logo-image-file>
--css ".label-customizable{ color: <color>;}"
```

## Amazon Cognito Identity Pools

*Amazon Cognito identity pools* allow you to create unique identities and assign permissions for your users. Inside the *identity pool*, you can include the following:

- Users in an Amazon Cognito user pool
- Users who authenticate with external identity providers such as Facebook, Google, or a SAML-based identity provider
- Users authenticated via your own existing authentication process

An identity pool allows you to obtain temporary AWS credentials with permissions that you define either to access other Amazon services directly or to access resources through Amazon API Gateway. Amazon Cognito identity pools help you integrate several authentication providers, such as the following:

- Amazon Cognito user pools
- Amazon.com users
- Facebook
- Google
- Twitter
- OpenID
- SAML
- Custom—supports your own identities such as `(login).(mycompany).(myapp)`

Once you enable the third-party resources that you want to allow to sign in to your apps, you can assign permissions to these users. With the combination of *user pools* and *identity pools*, you can create a serverless user authentication system.

Use this command to create an Amazon Cognito user pool with the CLI:

```
aws cognito-idp create-user-pool --pool-name <value>
```

## Amazon Cognito SDK

You can start developing for Amazon Cognito using the *AWS Mobile SDK*. Amazon Cognito currently supports the following SDKs through the AWS Mobile SDK:

- JavaScript SDK
- iOS SDK
- Android SDK

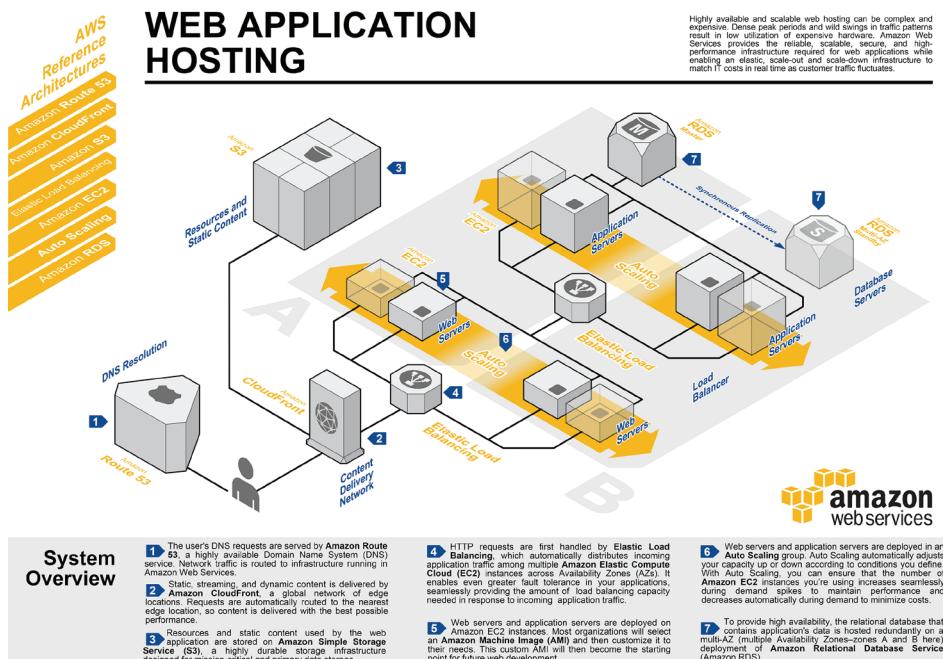
In addition to using the higher-level mobile and JavaScript SDKs, you can also use the lower-level APIs available via the following AWS SDKs to integrate all Amazon Cognito functionality in your applications:

- Java SDK
- .NET SDK
- Node.js SDK
- Python SDK
- PHP SDK
- Ruby SDK

## Standard Three-Tier vs. the Serverless Stack

This chapter has introduced serverless services and their benefits. Now that you know about some of the serverless services that are available in AWS, let's compare a traditional three-tier application against a serverless application architecture. Figure 13.5 shows a typical three-tier web application.

**FIGURE 13.5** Standard three-tier web infrastructure architecture



This architecture uses the following components and services:

**Routing:** Amazon Route 53

**Content distribution network (CDN):** Amazon CloudFront

**Static data:** Amazon S3

**High availability/decoupling:** Application load balancers

**Web servers:** Amazon EC2 with Auto Scaling

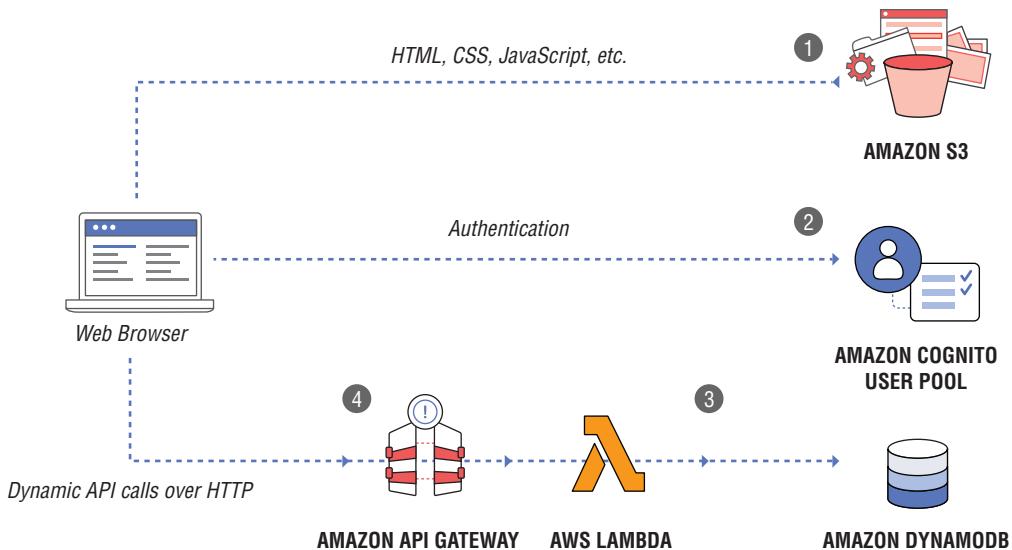
**App servers:** Amazon EC2 with Auto Scaling

**Database:** Amazon RDS in a multi-AZ configuration

Amazon Route 53 provides a DNS service that allows you to take domain names such as examplecompany.com and translate them to an IP address that points to running servers.

The CDN shown in Figure 13.6 is the Amazon CloudFront service, which improves your site performance with the use of its global content delivery network.

**FIGURE 13.6** Serverless web application architecture



Source: <https://aws.amazon.com/getting-started/projects/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/>

Amazon S3 stores your static files such as photos or movie files.

*Application load balancers* are responsible for distributing load across Availability Zones to your Amazon EC2 servers, which run your web application with a service such as Apache or NGINX.

Application servers are responsible for performing business logic prior to storing the data in your database servers that are run by Amazon RDS.

Amazon RDS is the managed database server, and it can run an Amazon Aurora, Microsoft SQL Server, Oracle SQL Server, MySQL, PostgreSQL, or MariaDB database server.

While this architecture is a robust and highly available service, there are several downsides, including the fact that you have to manage servers. You are responsible for patching those servers, preventing downtime associated with those patches, and proper server scaling.

In a typical serverless web application architecture, you also run a web application, but you have zero servers that run inside your AWS account, as shown in Figure 13.6.

Serverless web application architecture services include the following:

**Routing:** Amazon Route 53

**Web servers/static data:** Amazon S3

**User authentication:** Amazon Cognito user pools

**App servers:** Amazon API Gateway and AWS Lambda

**Database:** Amazon DynamoDB

Amazon Route 53 is your DNS, and you can use Amazon CloudFront for your CDN.

You can also use Amazon S3 for your web servers. In this architecture, you use Amazon S3 to host your entire static website. You use JavaScript to make API calls to the Amazon API Gateway service.

For your business or application servers, you use Amazon API Gateway in conjunction with AWS Lambda. This allows you to retrieve and save data dynamically.

You use Amazon DynamoDB as a serverless database service, and you do not provision any Amazon EC2s inside of your Amazon VPC account. Amazon DynamoDB is also a great database service for storing session state for stateful applications. You can use Amazon RDS instead if you need a relational database. However, it would not then be a fully serverless stack. There is a new service released called *Amazon Aurora Serverless*, which is a full RDS MySQL 5.6-compatible service that is completely serverless. This would allow you to run a traditional SQL database, but one that has the benefit of being serverless. Amazon Aurora Serverless is discussed in the next section.

You use Amazon Cognito user pools for user authentication, which provides a secure user directory that can scale to hundreds of millions of users. Amazon Cognito User Pools is a fully managed service with no servers for you to manage. While user authentication was not shown in Figure 13.6, you can use your web server tier to talk to a user directory, such as *Lightweight Directory Access Protocol* (LDAP), for user authentication.

As you can see, while some of the components are the same, you may use them in slightly different ways. By taking advantage of the AWS global network, you can develop fully scalable, highly available web applications—all without having to worry about maintaining or patching servers.

## Amazon Aurora Serverless

Amazon Aurora Serverless is an on-demand, auto-scaling configuration for the Aurora MySQL-compatible edition, where the database automatically starts, shuts down, and scales up or down as needed by your application. This allows you to run a traditional SQL database in the cloud without needing to manage any infrastructure or instances.

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.

**Know what flexible scaling means.** With flexible scaling, you no longer have to disable Amazon Elastic Compute Cloud (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 instances appropriately.

**Know what serverless applications mean.** Serverless applications have built-in availability and fault tolerance. You do not need to architect for these capabilities, as the services that run the application provide them by default. Additionally, when periods of low traffic occur on the web application, you do not spend money on Amazon EC2 instances that do not run at their full capacity.

**Know what services are serverless.** On the exam, it is important to understand which Amazon services are serverless and which ones are not. The following services are serverless:

- Amazon API Gateway
- AWS Lambda
- Amazon SQS
- Amazon SNS
- Amazon Kinesis
- Amazon Cognito
- Amazon Aurora Serverless
- Amazon S3

**Know how to host a serverless web application.** Hosting a serverless application means that you need Amazon S3 to host your static website, which comprises your HTML, JavaScript, and CSS files. For your database infrastructure, you can use Amazon DynamoDB or Amazon Aurora Serverless. For your business logic tier, you can use AWS Lambda. For DNS services, you can utilize Amazon Route 53. If you need the ability to host an API, you can use Amazon API Gateway. Finally, if you need to decrease latency to portions of your application, you can utilize services like Amazon CloudFront, which allows you to host your content at the edge.

## Resources to Review

Serverless Computing and Applications:

<https://aws.amazon.com/serverless/>

Amazon S3 Website Endpoints:

[https://docs.aws.amazon.com/general/latest/gr/rande.html#s3\\_website\\_region\\_endpoints](https://docs.aws.amazon.com/general/latest/gr/rande.html#s3_website_region_endpoints)

Amazon Cognito FAQs:

<https://aws.amazon.com/cognito/faqs/>

Amazon API Gateway FAQ:

<https://aws.amazon.com/api-gateway/faqs/>

AWS Well-Architected Framework—Serverless Applications Lens:

<https://d1.awsstatic.com/whitepapers/architecture/AWS-Serverless-Applications-Lens.pdf>

Serverless Architectures with AWS Lambda:

<https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf>

AWS Serverless Multi-Tier Architectures (Amazon API Gateway and AWS Lambda):

[https://d1.awsstatic.com/whitepapers/AWS\\_Serverless\\_Multi-Tier\\_Architectures.pdf](https://d1.awsstatic.com/whitepapers/AWS_Serverless_Multi-Tier_Architectures.pdf)

Serverless Streaming Architectures and Best Practices:

[https://d1.awsstatic.com/whitepapers/Serverless\\_Streaming\\_Architecture\\_Best\\_Practices.pdf](https://d1.awsstatic.com/whitepapers/Serverless_Streaming_Architecture_Best_Practices.pdf)

Optimizing Enterprise Economics with Serverless Architectures:

<https://d1.awsstatic.com/whitepapers/optimizing-enterprise-economics-serverless-architectures.pdf>

Common Serverless Architectures discussed at re:Invent 2017 (Video):

<https://www.youtube.com/watch?v=xJcm9V2jagc>

AWS Serverless Application Model (AWS SAM) FAQs:

<https://aws.amazon.com/serverless/sam/faqs/>

## Exercises

For this “Exercises” section, expand the OpenPets API Template that comes with Amazon API Gateway and build a frontend with HTML and JavaScript. You use AWS Lambda for some compute processing to save data to an Amazon DynamoDB database.

**EXERCISE 13.1****Create an Amazon S3 Bucket for the Swagger Template**

In this exercise, you use an AWS SAM template and a Swagger template to deploy your infrastructure. You will need to create an Amazon S3 bucket for the Swagger file.

1. Create an Amazon S3 bucket.

```
aws s3 mb s3://my-bucket-name --region us-east-1
```

If the command was successful, you should see output similar to the following, which means the bucket has been created:

```
make_bucket: my-bucket-name
```

2. Upload the **Swagger template**.

```
aws s3 cp petstore-api-swagger.yaml s3://my-bucket-name/petstore-api-swagger.yaml
```

If the file was successfully uploaded, you should be able to navigate to the Amazon S3 bucket and see it. This file is for the Swagger template, and it is used to create the REST API inside the Amazon API Gateway. You have not yet deployed the API.

3. Use AWS SAM to deploy your serverless infrastructure. To package your SAM template, run the following command:

```
aws cloudformation package \
--template-file ./petStoreSAM.yaml \
--s3-bucket my-bucket-name \
--output-template-file petStoreSAM-output.yaml \
--region us-east-1
```

If the command was successful, you should see that the file has been uploaded, and a new file called `petStoreSAM-output.yaml` has been created locally. You have packaged the AWS SAM template and converted it to a full AWS CloudFormation template. You will use this template in the next step to deploy the package to the Amazon API Gateway.

4. Deploy the package.

```
aws cloudformation deploy \
--template-file ./petStoreSAM-output.yaml \
--stack-name petStoreStack \
--capabilities CAPABILITY_IAM \
--parameter-overrides S3BucketName=s3://my-bucket-name/
petstore-api-swagger.yaml \
--region us-east-1
```

---

If the command was successful, you should see that the CloudFormation stack has been deployed. While it is in the process of deploying the resources, you will see something similar to the following:

Waiting for stack create/update to complete

This may take a few minutes. When it is finished deploying, the console displays the following message:

Successfully created/updated stack - petStoreStack

You have now successfully deployed the CloudFormation stack and can view the resources it created inside the AWS Management Console under the **AWS CloudFormation** service.

5. After the stack is created, run the command and write the results down for subsequent steps:

```
aws cloudformation describe-stacks --stack-name petStoreStack --region us-east-1 --query 'Stacks[0].Outputs[0].{PetStoreAPI:OutputValue}'
```

After running this command, the URL for the API is returned. Navigate to this URL to view the default page returned by the PetStore API. You will be changing this in the next exercise.

You have successfully completed the first exercise, created your AWS SAM template, and deployed it using AWS CloudFormation. Now your Amazon API Gateway is active, and you have the URL for accessing it.

---

## EXERCISE 13.2

### Edit the HTML Files

In steps 1 through 5, you are going to update the URL inside your .html files to point to the Amazon API Gateway stage that you have created. You do this so that your web application (.html files) knows the endpoint where to send your pet data.

1. Open index.html in the project folder and locate line 68 to find the variable named api\_gw\_endpoint. Input the value you retrieved from the previous command in Exercise 13.1.

```
var api_gw_endpoint = "https://cdvhqasdfnk444fe.execute-api.us-east-1.amazonaws.com/PetStoreProd/"
```

2. Open pets.html.
- 

(continued)

**EXERCISE 13.2 (*continued*)**

3. Input the value you received from the last command on line 96, and add /pets to the end of the string:

```
var api_gw_endpoint = "https://cdvhqasdfnk444fe.execute-api.us-east-1.amazonaws.com/PetStoreProd/pets"
```

4. Open add-pet.html.

5. Input the value you received from the last command on line 87, and add /pets to the end.

```
var api_gw_endpoint = "https://cdvhqasdfnk444fe.execute-api.us-east-1.amazonaws.com/PetStoreProd/pets"
```

6. Create a new Amazon S3 bucket for your website.

```
aws s3 mb s3://my-bucket-name --region us-east-1
```

7. Copy the project files to the website.

```
aws s3 cp . s3://my-bucket-name --recursive  
aws s3 rm s3://my-bucket-name/sam --recursive
```

Here you are uploading all the files from your project folder to Amazon S3 and then removing the SAM template from the bucket. You do not want others to have access to your template files and AWS Lambda functions. You want others to have access only to the end application.

8. Change the Amazon S3 bucket name inside of the policy.json to your bucket name. This will be on line 12.

9. Enable public read access for the bucket:

```
aws s3api put-bucket-policy --bucket my-bucket-name --policy file://policy.json
```

If successful, this command will not return any information. You are enabling the Amazon S3 bucket to be publicly accessible, meaning that everyone can access your website.

10. Enable the static website.

```
aws s3 website s3://my-bucket-name/ --index-document index.html --error-document index.html
```

The Amazon S3 bucket now acts as a web server and is running your pet store application.

11. Navigate to the website.

```
url: http://my-bucket-name.s3-website-us-east-1.amazonaws.com/index.html
```

12. Navigate Amazon API Gateway, AWS Lambda, Amazon DynamoDB, and the AWS SAM template to view the configuration.

Now that the application has been deployed, you can view all the individual components inside the AWS Management Console.

Inside Amazon API Gateway, you should see the PetStoreAPIGW. If you review the resources, you will see the various HTTP methods that you are allowing for your API.

In AWS Lambda, two functions were created: savePet for saving pets to Amazon DynamoDB and getPets for retrieving pets stored in Amazon DynamoDB.

In Amazon DynamoDB, you should have a table called PetStore. You can view the items in this table, though by default there should be none. After you create your first pet, however, you will be able to see some items in the table.

You can view the AWS SAM template and the AWS CloudFormation stack to see exactly how each of these resources were created.

---



With YAML, tab indentations are extremely important. Make sure that you have a valid YAML template. There are a variety of tools that you can use to validate YAML syntax. You can use the following websites to validate the YAML:

<https://codebeautify.org/yaml-validator>

<http://www.yamllint.com/>

If you want to perform client-side validation and not use a website, a number of IDEs support YAML validation. Refer to your IDE documentation to check for YAML support.

### EXERCISE 13.3

#### Define an AWS SAM Template

In this exercise, you will develop an AWS Lambda function locally and then test that Lambda function using the AWS SAM CLI. To perform this exercise successfully, you must have AWS SAM CLI installed. For information on how to install the AWS SAM CLI, review the following documentation: <https://github.com/awslabs/aws-sam-cli>. The following steps assume that you have a working AWS SAM CLI installation.

- Once you have installed AWS SAM CLI, open your favorite integrated development environment (IDE) and define an AWS SAM template.

---

(continued)

**EXERCISE 13.3 (*continued*)**

2. Enter the following in your template file:

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

```
Description: Welcome to the Pet Store Demo
```

Resources:

PetStore:

```
Type: AWS::Serverless::Function
```

Properties:

```
Runtime: nodejs8.10
```

```
Handler: index.handler
```

3. Save the file as template.yaml.

You have created the SAM template and saved the file locally. In subsequent exercises, you will use this information to execute an AWS Lambda function.

---

**EXERCISE 13.4****Define an AWS Lambda Function Locally**

Now that you have a valid SAM template, you can define your AWS Lambda function locally. In this example, use Nodejs 8.10, but you can use any AWS Lambda supported language.

1. Open your favorite IDE, and type the following Nodejs code:

```
'use strict';  
  
//A simple Lambda function  
exports.handler = (event, context, callback) => {  
  
    console.log('This is our local lambda function');  
    console.log('Creating a PetStore service');  
    callback(null, "Hello " + event.Records[0].dynamodb.NewImage.Message.S + "!  
What kind of pet are you interested in?");  
}
```

2. Save the file as index.js.

You have two files: an index.js and the SAM template. In the next exercise, you will generate an event source that will be used as the trigger for the AWS Lambda function.

---

**EXERCISE 13.5****Generate an Event Source**

Now that you have a valid SAM template and a valid AWS Lambda Nodejs 8.10 function, you can generate an event source.

1. Inside your terminal, type the following to generate an event source:

```
sam local generate-event dynamodb update > event.json
```

This will generate an Amazon DynamoDB update event. For a list of all of the event sources, type the following:

```
sam local generate-event -help
```

2. Modify the event source JSON file (event.json). On line 17, change New Item! to your first and last names.

```
"S": "John Smith"
```

You have now configured the three pieces that you need: the AWS SAM template, the AWS Lambda function, and the event source. In the next exercise, you will be able to run the AWS Lambda function locally.

**EXERCISE 13.6****Run the AWS Lambda Function**

Trigger and execute the AWS Lambda function.

1. In your terminal, type the following to execute the AWS Lambda function:

```
sam local invoke "PetStore" -e event.json
```

You will see the following message:

*Hello Casey Gerena! What kind of pet are you interested in?*

The AWS Lambda Docker image is downloaded to your local environment, and the event.json serves as all of the data that will be received as an event source to the AWS Lambda function. Inside the AWS SAM template, you will have given this function the name PetStore; however, you can define as many functions as you need to in order to build your application.

**EXERCISE 13.7****Modify the AWS SAM template to Include an API Locally**

To make your pet store into an API, modify the template.yaml.

1. Open the template.yaml file, and modify it to look like the following:

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

```
Description: Welcome to the Pet Store Demo
```

```
Resources:
```

```
PetStore:
```

```
Type: AWS::Serverless::Function
```

```
Properties:
```

```
Runtime: nodejs8.10
```

```
Handler: index.handler
```

```
Events:
```

```
PetStore:
```

```
Type: Api
```

```
Properties:
```

```
Path: /
```

```
Method: any
```

2. Save the template.yaml file.

You have modified the AWS SAM template to connect an Amazon API Gateway event for any method (GET, POST, and so on) to the AWS Lambda function. In the next exercise, you will modify the AWS Lambda function to work with the API.

---

**EXERCISE 13.8****Modify Your AWS Lambda Function for the API**

After you have defined an API, modify your AWS Lambda function.

1. Open the index.js file, and make the following changes:

```
'use strict';  
  
//A simple Lambda function  
exports.handler = (event, context, callback) => {  
  
    console.log('DEBUG: This is our local lambda function');
```

---

```
console.log('DEBUG: Creating a PetStore service');

callback(null, {
    statusCode: 200,
    headers: { "x-petstore-custom-header": "custom header from petstore
service" },
    body: '{"message": "Hello! Welcome to the PetStore. What kind of Pet
are you interested in?"}'
})

}
```

**2.** Save the index.js file.

You have modified the AWS Lambda function to respond to an API REST request. However, you have not actually executed anything—you will do that in the next exercise.

### EXERCISE 13.9

#### Run Amazon API Gateway Locally

Now that you have everything defined, run Amazon API Gateway locally.

**1.** Open a terminal and type the following:

```
sam local start-api
```

You will see output that looks like the following. Take note of the URL.

```
2018-10-11 23:05:25 Mounting PetStore at http://127.0.0.1:3000/hello [GET]
2018-10-11 23:05:25 You can now browse to the above endpoints to invoke your
functions. You do not need to restart/reload SAM CLI while working on your
functions changes will be reflected instantly/automatically. You only need to
restart SAM CLI if you update your AWS SAM template
2018-10-11 23:05:25 * Running on http://127.0.0.1:3000/ (Press CTRL+C to quit)
```

**2.** Open a web browser, and navigate to the previous URL.

You will see the following message:

```
Message: "Hello! Welcome to the Pet Store. What kind of Pet are you interested in?"
```

When you navigate to the URL, the local API Gateway forwards the request to AWS Lambda, which is also running locally, provided by index.js. You can now build serverless applications locally. When you are ready to deploy to a development or production environment, deploy the serverless applications to the AWS Cloud with AWS SAM. This allows developers to iterate through their code quickly and make improvements locally.

## Review Questions

1. Which templating engine can you use to deploy infrastructure inside of AWS that is built for serverless technologies?
  - A. AWS CloudFormation
  - B. Ansible
  - C. AWS OpsWorks for Automate Operations
  - D. AWS Serverless Application Model (AWS SAM)
2. What option do you need to enable to call Amazon API Gateway from another server or service?
  - A. You do not need to enable any options. Amazon API Gateway is ready to use as soon as it's deployed.
  - B. Enable cross-origin resource sharing (CORS).
  - C. Deploy a stage.
  - D. Deploy a resource.
3. A company is considering moving to the AWS serverless stack. What are two benefits of serverless stacks? (Select TWO.)
  - A. No server management
  - B. It costs less than Amazon Elastic Compute Cloud (Amazon EC2).
  - C. Flexible scaling
  - D. There are no benefits to serverless stacks.
4. Can you create HTTP endpoints with Amazon API Gateway?
  - A. Yes. You can create HTTP endpoints with Amazon API Gateway.
  - B. No. API Gateway creates FTP endpoints.
  - C. No. API Gateway only supports SSH endpoints.
  - D. No. API Gateway is a secure service that only supports HTTPS.
5. A company is moving to a serverless application, using Amazon Simple Storage Service (Amazon S3), AWS Lambda, and Amazon DynamoDB. They are currently using Amazon CloudFront for their content delivery network (CDN) network. They are concerned that they can no longer use Amazon CloudFront because they will have no Amazon Elastic Compute Cloud (Amazon EC2) instances running. Is their concern valid?
  - A. Their concerns are valid: Amazon CloudFront only supports Amazon EC2.
  - B. Their concerns are valid because all serverless applications are fully dynamic and contain no static information; thus, Amazon CloudFront does not support serverless applications.
  - C. Their concerns are not valid. Amazon CloudFront supports serverless applications
  - D. Their concerns are valid. Amazon CloudFront does support serverless applications; however, it does not support Amazon S3.

6. Amazon Cognito Mobile SDK does *not* support which language/platform?
  - A. iOS
  - B. Android
  - C. JavaScript
  - D. All of these languages/platform are supported.
7. Does Amazon Cognito support Short Message Service (SMS)–based multi-factor authentication (MFA)?
  - A. No. Amazon Cognito does not support SMS-based MFA.
  - B. No. Amazon Cognito does not support SMS-based MFA; however, it does support MFA.
  - C. Yes. Amazon Cognito does support SMS-based MFA.
  - D. None of the above.
8. Does Amazon Cognito support device tracking and remembering?
  - A. Amazon Cognito does not support device tracking and remembering.
  - B. Amazon Cognito supports device tracking but not remembering.
  - C. Amazon Cognito supports device remembering but not tracking.
  - D. Amazon Cognito supports device remembering and tracking.
9. What is the property name that you use to connect an AWS Lambda function to the Amazon API Gateway inside of an AWS Serverless Application Model (AWS SAM) template?
  - A. events
  - B. handler
  - C. context
  - D. runtime
10. A company wants to use a serverless application to run its dynamic website that is currently running on Amazon Elastic Compute Cloud (Amazon EC2) and Elastic Load Balancing (ELB). Currently, the application uses HTML, CSS, and React, and the database is a NoSQL flavor. You are the advisor—is this possible?
  - A. No. This is not possible, because there is no way to run React in AWS. React is a Facebook technology.
  - B. No. This is not possible, because you need an Amazon EC2 to run the web server.
  - C. No. This is not possible, because there is no way to load balance a serverless application.
  - D. Yes. This is possible; however, some refactoring will be required.



# Chapter 14



AWS® Certified Developer Official Study Guide  
By Nick Alteen, Jennifer Fisher, Casey Gerena, Wes Gruver, Asim Jalil,  
Heiward Osman, Marife Pagan, Santosh Patlolla and Michael Roth  
Copyright © 2019 by Amazon Web Services, Inc.

# Stateless Application Patterns

---

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

#### Domain 1: Deployment

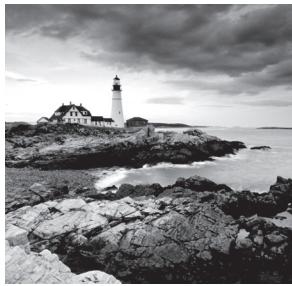
- ✓ 1.4 Deploy serverless applications.

#### Domain 2: Security

- ✓ 2.1 Make authenticated calls to AWS services.
- ✓ 2.2 Implement encryption using AWS services.
- ✓ 2.3 Implement application authentication and authorization.

#### Domain 3: Development with AWS Services

- ✓ 3.2 Translate functional requirements into application design.
- ✓ 3.3 Implement application design into application code.
- ✓ 3.4 Write code that interacts with AWS services by using APIs, SDKs, and AWS CLI.



## Introduction to the Stateless Application Pattern

In previous chapters, you were introduced to compute, networking, databases, and storage on the AWS Cloud. This chapter covers the fully managed services that you use to build stateless applications on AWS. Scalability is an important consideration when you create and deploy applications that are highly available, and stateless applications are easier to scale.

When users or services interact with an application, they often perform a sequence of interactions that form a session. A *stateless application* is one that requires no knowledge of previous interactions and stores no session information. Given the same input, an application can provide the same response to any user.

A stateless application can scale horizontally because requests can be serviced by any of the available compute resources, such as Amazon Elastic Compute Cloud (Amazon EC2) instances or AWS Lambda functions. With no session data sharing, you can add more compute resources as necessary. When that compute capacity is no longer needed, you can safely terminate any individual resource. Those resources do not need to be aware of the presence of their peers, and they only need a way to share the workload among them.

This chapter discusses the AWS services that provide a mechanism for persisting state outside of the application: Amazon DynamoDB, Amazon Simple Storage Service (Amazon S3), Amazon ElastiCache, and Amazon Elastic File System (Amazon EFS).

## Amazon DynamoDB

*Amazon DynamoDB* is a fast and flexible NoSQL database service that applications use that require consistent, single-digit millisecond latency at any scale. A fully managed NoSQL database supports both document and key-value store models. DynamoDB is ideal for mobile, web, gaming, ad tech, and Internet of Things (IoT) applications. DynamoDB provides an effective solution for sharing session states across web servers, Amazon EC2 instances, or computing nodes.

## Using Amazon DynamoDB to Store State

DynamoDB provides fast and predictable performance with seamless scalability. It enables you to offload the administrative burdens of operating and scaling a distributed database, including hardware provisioning, setup and configuration, replication, software patching, or cluster scaling. Also, DynamoDB offers encryption at rest, which reduces the operational tasks and complexity involved in protecting sensitive data.

With DynamoDB, you can create database *tables* that can store and retrieve any amount of data (*collection*) and serve any level of request traffic. You can scale up or scale down the throughput capacity of your tables without downtime or performance degradation and use the *AWS Management Console* to monitor resource utilization and performance metrics. DynamoDB provides on-demand backup capability to create full backups of tables for long-term retention and archives for regulatory compliance. Use DynamoDB to delete expired items from tables automatically to reduce both storage usage and the cost to store irrelevant data.

DynamoDB automatically spreads data and traffic for tables over a sufficient number of servers to handle throughput and storage requirements while maintaining consistent and fast performance. All of your data is stored on solid-state drive (SSDs) and automatically replicates across multiple Availability Zones in an AWS Region, providing built-in high availability and data durability. You can use global tables to keep DynamoDB tables synchronized across AWS Regions, and you can access this service using the DynamoDB console, the AWS Command Line Interface (AWS CLI), a generic web services Application Programming Interface (API), or any programming language that the AWS software development kit (AWS SDK) supports.

Tables, items, and attributes are core components of DynamoDB. A *table* is a collection of items, and each item is a collection of attributes. For example, you could have a table called *Cars*, which stores information about vehicles. DynamoDB uses *primary keys* to identify each item in a table (e.g., Ford) uniquely and *secondary indexes* to provide more querying flexibility (e.g., Mustang). You can use *Amazon DynamoDB Streams* to capture data modification events in DynamoDB tables.

## Primary Key, Partition Key, and Sort Key

When you create a table, you must configure both the *table name* and the *primary key* of the table. The *primary key* uniquely identifies each item in the table so that no two items have the same key. DynamoDB supports two different kinds of primary keys: a partition key and sort key.

A *partition key* is a simple *primary key*, composed of only a *partition key attribute*. The partition key of an item is also known as its *hash attribute*. The term *hash attribute* derives from the use of an internal hash function in DynamoDB that evenly distributes data items across partitions based on their partition key values. DynamoDB uses the partition key's value as input to an internal hash function. The output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item will be stored.



In a table that has only a partition key, no two items can have the same partition key value.

You can also create a *primary key* as a *composite primary key*, consisting of a partition key (first attribute) and a sort key (second attribute).

The *sort key* of an item is also known as its *range attribute*. The term *range attribute* derives from the way that DynamoDB stores items with the same partition key physically close together, in sorted order, by the sort key value.

Each primary key attribute must be a *scalar*, meaning that it can hold only a single value. The only data types allowed for primary key attributes are string, number, or binary. There are no such restrictions for other, nonkey attributes.

## Best Practices for Designing and Using Partition Keys

When DynamoDB stores data, it divides table items into multiple physical partitions primarily based on the partition key values, and it distributes the table data accordingly. The primary key that uniquely identifies each item in a DynamoDB table can be either *simple* (partition key only) or *composite* (partition key combined with a sort key). *Partition key values* determine the logical partitions in which a table's data is stored, which affects the table's underlying physical partitions. Efficient partition key design keeps your workload spread evenly across these partitions.

A single physical DynamoDB partition supports a maximum of 3,000 read-capacity units (RCUs) or 1,000 write-capacity units (WCUs). Provisioned I/O capacity for the table is divided evenly among all physical partitions. Therefore, design your partition keys to spread I/O requests as evenly as possible across the table's partitions to prevent “hot spots” that use provisioned I/O capacity inefficiently.

### Example 1: Hotspot

If a table has a small number of heavily accessed partition key values (possibly even one heavily used partition key value), request traffic is concentrated on a small number of partitions, or only one partition. If the workload is heavily unbalanced, meaning that it is disproportionately focused on one or a few partitions, the requests do not achieve the overall provisioned throughput level.



To achieve the maximum DynamoDB throughput, create tables where the partition key has a large number of distinct values, and values are requested fairly uniformly, as randomly as possible.

## Designing Partition Keys to Distribute Even Workloads

The partition key portion of a table's primary key determines the logical partitions in which a table's data is stored, and it affects the underlying physical partitions.

*Provisioned I/O* capacity for the table is divided evenly among these physical partitions, but a partition key design that does not distribute I/O requests evenly can create “hot” partitions that use your provisioned I/O capacity inefficiently and result in throttling.

The optimal usage of a table's provisioned throughput depends on both the workload patterns of individual items and the partition key design. This does not mean that you must access all partition key values to achieve an efficient throughput level or even that the percentage of accessed partition key values must be high. It does mean that the more distinct partition key values that your workload accesses, the more those requests are spread across the partitioned space. You will use your provisioned throughput more efficiently as the ratio of partition key values accessed to the total number of partition key values increases.

Table 14.1 provides a comparison of the provisioned throughput efficiency of some common partition key schemas.

**TABLE 14.1** Partition Key Schemas

Partition Key Value	Uniformity
User Identification (ID) where the application has many users	Good
Status Code where there are only a few possible status codes	Bad
Item Creation Date rounded to the nearest period (day, hour, or minute)	Bad
Device ID where each device accesses data at relatively similar intervals	Good
Device ID where even if there are many devices being tracked, one is by far more popular than all the others	Bad

If a single table has only a small number of partition key values, consider distributing your write operations across more distinct partition key values, and structure the primary key elements to avoid one “hot” (heavily requested) partition key value that slows overall performance.

Consider a table with a composite primary key. The partition key represents the item's creation date, rounded to the nearest day. The sort key is an item identifier. On a given day, all new items are written to that single partition key value and corresponding physical partition.

If the table fits entirely into a single partition (considering the growth of your data over time), and if your application's read and write throughput requirements do not exceed the read and write capabilities of a single partition, your application does not encounter any unexpected throttling because of partitioning.

However, if you anticipate your table scaling beyond a single partition, architect your application so that it can use more of the table's full provisioned throughput.

## Using Write Shards to Distribute Workloads Evenly

A *shard* is a uniquely identified group of stream records within a stream. To distribute writes better across a partition key space in DynamoDB, expand the space. You can add a random number to the partition key values to distribute the items among partitions, or you can use a number that is calculated based on what you want to query.

### Random Suffixes in Shards

To distribute loads more evenly across a partition key space, add a random number to the end of the partition key values and then randomize the writes across the larger space. For example, if a partition key represents today's date, choose a random number from 1 through 200, and add it as a suffix to the date. This yields partition key values such as 2018-07-09.1, 2014-07-09.2, and so on, through 2018-07-09.200. Because you are randomizing the partition key, the writes to the table on each day are spread evenly across multiple partitions. This results in better parallelism and higher overall throughput.

To read all of the items for a given day, you would have to query the items for all of the suffixes and then merge the results. For example, first issue a Query request for the partition key value 2018-07-09.1, then another Query for 2018-07-09.2, and so on, through 2018-07-09.200. When complete, your application then merges the results from all Query requests.

### Calculated Suffixes in Shards

A *random strategy* can improve write throughput, but it is difficult to read a specific item because you do not know which suffix value was written to the item. To make it easier to read individual items, instead of using a random number to distribute the items among partitions, use a number that you can calculate based on what you want to query.

Consider the previous example, where a table uses today's date in the partition key. Now suppose that each item has an accessible OrderId attribute and that you most often need to find items by OrderId in addition to date. Before your application writes the item to the table, it can calculate a hash suffix based on the OrderId, append it to the partition key date, and generate numbers from 1 through 200 that evenly distribute, similar to what the random strategy produces. You can use a simple calculation, such as the product of the UTF-8 code point values, for the characters in the OrderId, modulo 200, + 1. The partition key value would then be the date concatenated with the calculation result.

With this strategy, the writes are spread evenly across the partition key values and across the physical partitions. You can easily perform a GetItem operation for a particular item and date because you can calculate the partition key value for a specific OrderId value.

To read all of the items for a given day, you must query each of the 2018-07-09.N keys (where N is 1 through 200), and your application then merges the results. With this strategy, you avoid a single "hot" partition key value taking the entire workload.

## Items

Each table contains zero or more items. An *item* is a group of attributes that is uniquely identifiable among all other entities in the table. For example, in a *People* table, each item represents a person, and in a *Cars* table, each item represents one vehicle. Items in DynamoDB are similar to rows, records, or tables in other database systems. However, in DynamoDB, there is no limit to the number of items that you can store in a table.

## Attributes

Each item in a table is composed of one or more attributes. An *attribute* is a fundamental data element, something that does not need to be broken down any further. For example, an item in a *People* table contains attributes called PersonID, LastName, FirstName, and so on. For a *Department* table, an item may have attributes such as DepartmentID, Name, Manager, and so on. Attributes in DynamoDB are similar in many ways to fields or columns in other database systems.

The naming rules for DynamoDB tables are as follows:

- All names must be encoded using UTF-8 and be case-sensitive.
- Table names must be between 3 and 255 characters long and can contain only the following characters:
  - a–z
  - A–Z
  - 0–9
  - \_ (underscore)
  - - (dash)
  - . (period)
- Attribute names must be between 1 and 255 characters long.
- Each item in the table has a unique identifier, or primary key, which distinguishes the item from all others in the table. In a *People* table, the primary key consists of one attribute, PersonID.
- Other than the primary key, the *People* table is *schema-less*, meaning that you are not required to define the attributes or their data types beforehand. Each item can have its own distinct attributes.
- Most of the attributes are *scalar*, meaning that they can have only one value. Strings and numbers are common scalars.
- Some of the items have a nested attribute. For example, in a *People* table, the Address attribute may have nested attributes such as Street, City, and PostalCode. DynamoDB supports nested attributes up to 32 levels deep.

## Data Types

DynamoDB supports several *data types* for attributes within a table.

## Scalar

A *scalar type* can represent exactly one value. The scalar types are number, string, binary, Boolean, and null.

**Number** *Numbers* can be positive, negative, or zero and can have up to 38 digits of precision. Exceeding this limit results in an exception. Numbers are presented as variable length. Leading and trailing zeros are trimmed. All numbers are sent as strings to maximize compatibility across languages and libraries. DynamoDB treats them as number-type attributes for mathematical operations. You can use the number data type to represent a date or a timestamp. One way to do this is with the epoch time, the number of seconds since 00:00:00 Coordinated Universal Time (UTC) on January 1, 1970.

**String** *Strings* are Unicode with UTF-8 binary encoding. The length of a string must be greater than zero, and it is constrained by the maximum DynamoDB item size limit of 400 KB. If a primary key attribute is a string type, the following additional constraints apply:

- For a *simple primary key*, the maximum length of the first attribute value (partition key) is 2,048 bytes.
- For a *composite primary key*, the maximum length of the second attribute value (sort key) is 1,024 bytes.
- DynamoDB collates and compares strings using the bytes of the underlying UTF-8 string encoding. For instance, “a” (0x61) is greater than “A” (0x41).

You can use the string data type to represent a date or a timestamp. One way to do this is to use ISO 8601 strings as follows:

- 2018-04-19T12:34:56Z
- 2018-02-31T10:22:18Z
- 2017-05-08T12:22:46Z

**Binary** *Binary type attributes* can store any binary data, such as compressed text, encrypted data, or images. Whenever DynamoDB compares binary values, it treats each byte of the binary data as unsigned. The length of a binary attribute must be greater than zero, and it is constrained by the maximum DynamoDB item size limit of 400 KB. If a primary key attribute is a binary type, the following additional constraints apply:

- For a simple primary key, the maximum length of the first attribute value (partition key) is 2,048 bytes.
- For a composite primary key, the maximum length of the second attribute value (sort key) is 1,024 bytes.

Applications must encode binary values in base64-encoded format before sending them to DynamoDB. Upon receipt, DynamoDB decodes the data into an unsigned byte array and uses it as the length of the binary attribute.

**Boolean** A *Boolean type attribute* can store one of two values: true or false.

**Null** A *null attribute* is one with an unknown or undefined state.

## Document

There are two document types, list and map, which you can nest within each other to represent complex data structures up to 32 levels deep. There is no limit on the number of values in a list or a map, as long as the item containing the values fits within the DynamoDB item size limit of 400 KB.



An attribute value cannot be an empty string or an empty set; however, empty lists and maps are allowed.

**List** A *list type attribute* can store an ordered collection of values. Lists are enclosed in square brackets [ ... ] and are similar to a JavaScript Object Notation (JSON) array. There are no restrictions on the data types that can be stored in a list element, and the elements in a list element can be of different types. Here is an example of a list with strings and numbers:

```
MyFavoriteThings: ["Thriller", "Purple Rain", 1983, 2]
```

**Map** A *map type attribute* can store an unordered collection of name/value pairs. Maps are enclosed in curly braces { ... } and are similar to a JSON object. There are no restrictions on the data types that you can store in a map element, and elements in a map do not have to be the same type. Maps are ideal for storing JSON documents in DynamoDB. The following example shows a map that contains a string, a number, and a nested list that contains another map:

```
{
    Location: "Labrynth",
    MagicStaff: 1,
    MagicRings: [
        "The One Ring",
        {
            "ElevenKings: { Quantity : 3},
            "DwarfLords: { Quantity : 7},
            "MortalMen: { Quantity : 9}
        }
    ]
}
```



DynamoDB enables you to work with individual elements within maps—even if those elements are deeply nested.

**Set** DynamoDB supports types that represent *sets* of number, string, or binary values. There is no limit on the number of values in a set, as long as the item containing the value fits within the DynamoDB 400 KB item size limit. Each value within a set must be unique. The order of the values within a set is not preserved. Applications must not rely on the order of elements within the set. DynamoDB does not support empty sets.



All of the elements within a set must be of the same type.

## Amazon DynamoDB Tables

DynamoDB global tables provide a fully managed solution for deploying a multiregion, multi-master database, without having to build and maintain your own replication solution. When you create a global table, you configure the AWS Regions where you want the table to be available. DynamoDB performs all of the necessary tasks to create identical tables in these regions and propagate ongoing data changes to all of the regions.

DynamoDB global tables are ideal for massively scaled applications, with globally dispersed users. In such an environment, you can expect fast application performance. Global tables provide automatic multi-master replication to AWS Regions worldwide, so you can deliver low-latency data access to your users no matter where they are located.

There is no practical limit on a table's size. Tables are unconstrained in terms of the number of items or the number of bytes. For any AWS account, there is an initial limit of 256 tables per region.

## Provisioned Throughput

With DynamoDB, you can create *database tables* that store and retrieve any amount of data and serve any level of request traffic. You can scale your table's throughput capacity up or down without downtime or performance degradation, and you can use the AWS Management Console to monitor resource utilization and performance metrics.



For any table or global secondary index, the minimum settings for provisioned throughput are one read capacity unit and one write capacity unit.

AWS places some default limits on the throughput that you can provision. These are the limits unless you request a higher amount.



You can apply all of the available throughput of an account to a single table or across multiple tables.

## Throughput Capacity for Reads and Writes in Tables and Indexes

When you create a *table* or *index* in DynamoDB, you must configure your capacity requirements for read and write activity. If you define the throughput capacity in advance,

DynamoDB can reserve the necessary resources to meet the read and write activity that your application requires, while it ensures consistent, low-latency performance.

Throughput capacity is specified in terms of read capacity units or write capacity units:

- One read capacity unit represents one strongly consistent read per second, or two eventually consistent reads per second, for an item up to 4 KB in size. If you need to read an item larger than 4 KB, DynamoDB must consume additional read capacity units. The total number of read capacity units required depends on both the item size and whether you want an eventually consistent read or strongly consistent read.
- One write capacity unit represents one write per second for an item up to 1 KB in size. If you need to write an item larger than 1 KB, DynamoDB must consume additional write capacity units. The total number of write capacity units required depends on the item size.

For example, if you create a table with five read capacity units and five write capacity units, your application could do the following:

- Perform strongly consistent reads of up to 20 KB per second ( $4\text{ KB} \times 5$  read capacity units)
- Perform eventually consistent reads of up to 40 KB per second (twice as much read throughput)
- Write up to 5 KB per second ( $1\text{ KB} \times 5$  write capacity units)

If your application reads or writes larger items (up to the DynamoDB maximum item size of 400 KB), it will consume more capacity units.

If your read or write requests exceed the throughput settings for a table, DynamoDB can throttle that request. DynamoDB can also throttle read requests exceeds for an index. Throttling prevents your application from consuming too many capacity units. When a request is throttled, it fails with HTTP 400 code (Bad Request) and a ProvisionedThroughputExceededException. The AWS SDKs have built-in support for retrying throttled requests, so you do not need to write this logic yourself.

You can use the AWS Management Console to monitor your provisioned and actual throughput and modify your throughput settings if necessary.

DynamoDB provides the following mechanisms for managing throughput:

- DynamoDB automatic scaling
- Provisioned throughput
- Reserved capacity
- AWS Lambda triggers in DynamoDB streams

## Setting Initial Throughput Settings

Every application has different requirements for reading and writing from a database. When you determine the initial throughput settings for a DynamoDB table, take the following attributes into consideration:

**Item sizes** Some items are small enough that they can be read or written by using a single capacity unit. Larger items require multiple capacity units. By estimating the sizes of the items that will be in your table, you can configure accurate settings for your table's provisioned throughput.

**Expected read and write request rates** In addition to item size, estimate the number of reads and writes to perform per second.

**Read consistency requirements** Read capacity units are based on strongly consistent read operations, which consume twice as many database resources as eventually consistent reads. Determine whether your application requires strongly consistent reads, or whether it can relax this requirement and perform eventually consistent reads instead.



Read operations in DynamoDB are by default eventually consistent, but you can request strongly consistent reads for these operations if necessary.

## Item Sizes and Capacity Unit Consumption

Before you choose read and write capacity settings for your table, understand your data and how your application will access it. These inputs help you determine your table's overall storage and throughput needs and how much throughput capacity your application will require. Except for the primary key, DynamoDB tables are *schemaless*, so the items in a table can all have different attributes, sizes, and data types. The *total size of an item is the sum of the lengths of its attribute names and values*. You can use the following guidelines to estimate attribute sizes:

- Strings are Unicode with UTF-8 binary encoding. The size of a string is as follows:  
$$(\text{length of attribute name}) + (\text{number of UTF-8-encoded bytes})$$
- Numbers are variable length, with up to 38 significant digits. Leading and trailing zeroes are trimmed.
- The size of a number is approximately as follows:  
$$(\text{length of attribute name}) + (1 \text{ byte per two significant digits}) + (1 \text{ byte})$$
- A binary value must be encoded in base64 format before it can be sent to DynamoDB, but the value's raw byte length is used for calculating size. The size of a binary attribute is as follows:  
$$(\text{length of attribute name}) + (\text{number of raw bytes})$$

- The size of a null attribute or a Boolean attribute is as follows:  
$$(\text{length of attribute name}) + (1 \text{ byte})$$
- An attribute of type List or Map requires 3 bytes of overhead, regardless of its contents. The size of a List or Map is as follows:  
$$(\text{length of attribute name}) + \text{sum}(\text{size of nested elements}) + (3 \text{ bytes})$$
- The size of an empty List or Map is as follows:  
$$(\text{length of attribute name}) + (3 \text{ bytes})$$



Choose short attribute names rather than long ones. This helps to optimize capacity unit consumption and reduce the amount of storage required for your data.

## Capacity Unit Consumption for Reads

The following describes how read operations for DynamoDB consume read capacity units:

**GetItem** Reads a single item from a table. To determine the number of capacity units GetItem will consume, take the item size and round it up to the next 4 KB boundary. If you specified a strongly consistent read, this is the number of capacity units required. For an eventually consistent read (the default), take this number and divide it by 2.

For example, if you read an item that is 3.5 KB, DynamoDB rounds the item size to 4 KB. If you read an item of 10 KB, DynamoDB rounds the item size to 12 KB.

**BatchGetItem** Reads up to 100 items, from one or more tables. DynamoDB processes each item in the batch as an individual GetItem request, so DynamoDB first rounds up the size of each item to the next 4-KB boundary and then calculates the total size. The result is not necessarily the same as the total size of all the items.

For example, if BatchGetItem reads a 1.5-KB item and a 6.5-KB item, DynamoDB calculates the size as 12 KB (4 KB + 8 KB), not 8 KB (1.5 KB + 6.5 KB).

**Query** Reads multiple items that have the same partition key value. All of the items returned are treated as a single read operation, whereby DynamoDB computes the total size of all items and then rounds up to the next 4-KB boundary.

For example, suppose that your query returns 10 items whose combined size is 40.8 KB. Amazon DynamoDB rounds the item size for the operation to 44 KB. If a query returns 1,500 items of 64 bytes each, the cumulative size is 96 KB.

**Scan** Reads all items in a table. DynamoDB considers the size of the items that are evaluated, not the size of the items returned by the scan.

If you perform a read operation on an item that does not exist, DynamoDB will still consume provisioned read throughput. A request for a strongly consistent read consumes one read capacity unit, whereas a request for an eventually consistent read consumes 0.5 of a read capacity unit.

For any operation that returns items, request a subset of attributes to retrieve. However, doing so has no impact on the item size calculations. In addition, Query and Scan can return item counts instead of attribute values. Getting the count of items uses the same quantity of read capacity units and is subject to the same item size calculations, because DynamoDB has to read each item to increment the count:

**Read operations and read consistency** The preceding calculations assumed requests for strongly consistent reads. For a request for eventually consistent reads, the operation consumes only half of the capacity units. For example, of an eventually consistent read, if the total item size is 80 KB, the operation consumes only 10 capacity units.

**Read consistency for Scan** A Scan operation performs eventually consistent reads, by default. This means that the Scan results might not reflect changes as the result of recently completed PutItem or UpdateItem operations. If you require strongly consistent reads, when the Scan begins, set the ConsistentRead parameter to true in the Scan request. This ensures that all of the write operations that completed before the Scan began are included in the Scan response. Setting ConsistentRead to true can be useful in table backup or replication scenarios. With DynamoDB streams, to obtain a consistent copy of the data in the table, first use Scan with ConsistentRead set to true. During the Scan, DynamoDB streams record any additional write activity that occurs on the table. After the Scan completes, apply the write activity from the stream to the table.



A Scan operation with ConsistentRead set to true consumes twice as many read capacity units as compared to keeping ConsistentRead at the default value (false).

## Capacity Unit Consumption for Writes

The following describes how DynamoDB write operations consume write capacity units:

**PutItem** Writes a single item to a table. If an item with the same primary key exists in the table, the operation replaces the item. For calculating provisioned throughput consumption, the item size that matters is the larger of the two.

**UpdateItem** Modifies a single item in the table. DynamoDB considers the size of the item as it appears before and after the update. The provisioned throughput consumed reflects the larger of these item sizes. Even if you update only a subset of the item's attributes, UpdateItem will consume the full amount of provisioned throughput (the larger of the “before” and “after” item sizes).

**DeleteItem** Removes a single item from a table. The provisioned throughput consumption is based on the size of the deleted item.

**BatchWriteItem** Writes up to 25 items to one or more tables. DynamoDB processes each item in the batch as an individual PutItem or DeleteItem request (updates are not supported). DynamoDB first rounds up the size of each item to the next 1-KB boundary and then calculates the total size. The result is not necessarily the same as the total size of all the items. For example, if BatchWriteItem writes a 500-byte item and a 3.5-KB item, DynamoDB calculates the size as 5 KB (1 KB + 4 KB), not 4 KB (500 bytes + 3.5 KB).

For PutItem, UpdateItem, and DeleteItem operations, DynamoDB rounds the item size up to the next 1 KB. If you put or delete an item of 1.6 KB, DynamoDB rounds the item size up to 2 KB.

PutItem, UpdateItem, and DeleteItem allow *conditional writes*, whereby you configure an expression that must evaluate to true for the operation to succeed. If the expression evaluates to false, DynamoDB consumes write capacity units from the table.

For an existing item, the number of write capacity units consumed depends on the size of the new item. For example, a failed conditional write of a 1-KB item would consume one write capacity unit. If the new item were twice that size, the failed conditional write would consume two write capacity units.



For a new item, DynamoDB consumes one write capacity unit.

If a ConditionExpression evaluates to false during a conditional write, DynamoDB will consume write capacity from the table based on the following conditions:

- If the item does not currently exist in the table, DynamoDB consumes one write capacity unit.
- If the item does exist, then the number of write capacity units consumed depends on the size of the item. For example, a failed conditional write of a 1-KB item would consume one write capacity unit. If the item were twice that size, the failed conditional write would consume two write capacity units.



Write operations consume write capacity units only. Write operations do not consume read capacity units.

A failed conditional write returns a ConditionalCheckFailedException. When this occurs, you do not receive any information in the response about the write capacity that was consumed. However, you can view the ConsumedWriteCapacityUnits metric for the table in Amazon CloudWatch.

To return the number of write capacity units consumed during a conditional write, use the `ReturnConsumedCapacity` parameter with any of the following attributes:

**Total** Returns the total number of write capacity units consumed.

**Indexes** Returns the total number of write capacity units consumed with subtotals for the table and any secondary indexes that were affected by the operation.

**None** No write capacity details are returned (default).



Unlike a global secondary index, a local secondary index shares its provisioned throughput capacity with its table. Read and write activity on a local secondary index consumes provisioned throughput capacity from the table.

### Capacity Unit Sizes

One *read* capacity unit = one strongly consistent read per second, or two eventually consistent reads per second, for items up to 4 KB in size.

One *write* capacity unit = one write per second, for items up to 1 KB in size.

## Creating Tables to Store the State

Before you store state in DynamoDB, you must create a table. To work with DynamoDB, your application must use several API operations and be organized by category.

### Control Plane

*Control plane operations* let you create and manage DynamoDB tables and work with indexes, streams, and other objects that are dependent on tables.

**CreateTable** Creates a new table. You can create one or more secondary indexes and enable DynamoDB Streams for the table.

**DescribeTable** Returns information about a table, such as its primary key schema, throughput settings, and index information.

**ListTables** Returns the names of all of the tables in a list.

**UpdateTable** Modifies the settings of a table or its indexes, creates or removes new indexes on a table, or modifies settings for a table in DynamoDB Streams.

**DeleteTable** Removes a table and its dependent objects from DynamoDB.

## Data Plane

*Data plane operations* let you perform *create/read/update/delete* (CRUD) actions on data in a table. Some data plane operations also enable you to read data from a secondary index.

### Creating Data

The following data plane operations enable you to perform create actions on data in a table:

**PutItem** Writes a single item to a table. You must configure the primary key attributes, but you do not have to configure other attributes.

**BatchWriteItem** Writes up to 25 items to a table. This is more efficient than multiple PutItem commands because your application needs only a single network round trip to write the items. You can also use BatchWriteItem to delete multiple items from one or more tables.

### Performing Batch Operations

DynamoDB provides the BatchGetItem and BatchWriteItem operations for applications that need to read or write multiple items. Use these operations to reduce the number of network round trips from your application to DynamoDB. In addition, DynamoDB performs the individual read or write operations in *parallel*. Your applications benefit from this parallelism without having to manage concurrency or threading.

The batch operations are wrappers around multiple read or write requests. If a BatchGetItem request contains five items, DynamoDB performs five GetItem operations on your behalf. Similarly, if a BatchWriteItem request contains two put requests and four delete requests, DynamoDB performs two PutItem and four DeleteItem requests.

In general, a batch operation does not fail unless *all* requests in that batch fail. If you perform a BatchGetItem operation, but one of the individual GetItem requests in the batch fails, the BatchGetItem returns the keys and data from the GetItem request that failed. The other GetItem requests in the batch are not affected.

**BatchGetItem** A single BatchGetItem operation can contain up to 100 individual GetItem requests and can retrieve up to 16 MB of data. In addition, a BatchGetItem operation can retrieve items from multiple tables.

**BatchWriteItem** The BatchWriteItem operation can contain up to 25 individual PutItem and DeleteItem requests and can write up to 16 MB of data. The maximum size of an individual item is 400 KB. In addition, a BatchWriteItem operation can put or delete items in multiple tables.



BatchWriteItem does not support UpdateItem requests.

## Reading Data

The following data plane operations enable you to perform read actions on data in a table:

**GetItem** Retrieves a single item from a table. You must configure the primary key for the item that you want. You can retrieve the entire item or only a subset of its attributes.

**BatchGetItem** Retrieves up to 100 items from one or more tables. This is more efficient than calling GetItem multiple times because your application needs only a single network round trip to read the items.

**Query** Retrieves all items that have a specific partition key. You must configure the *partition key* value. You can retrieve entire items or only a subset of their attributes. You can apply a condition to the sort key values so that you retrieve only a subset of the data that has the same partition key.



You can use the Query operation on a table or index if the table or index has both a partition key and a sort key.

**Scan** Retrieves all the items in the table or index. You can retrieve entire items or only a subset of their attributes. You can use a filter condition to return only the values that you want and discard the rest.

## Updating Data

UpdateItem modifies one or more attributes in an item. You must configure the primary key for the item that you want to modify. You can add new attributes and modify or remove existing attributes. You can also perform conditional updates so that the update is successful only when a user-defined condition is met. You can also implement an atomic counter, which increments or decrements a numeric attribute without interfering with other write requests.

## Deleting Data

The following data plane operations enable you to perform delete actions on data in a table:

**DeleteItem** Deletes a single item from a table. You must configure the primary key for the item that you want to delete.

**BatchDeleteItem** Deletes up to 25 items from one or more tables. This is more efficient than multiple DeleteItem calls, because your application needs only a single network round trip. You can also use BatchWriteItem to add multiple items to one or more tables.

## Return Values

In some cases, you may want DynamoDB to return certain attribute values as they appeared before or after you modified them. The PutItem, UpdateItem, and DeleteItem operations have a ReturnValue parameter that you can use to return the attribute values

before or after they are modified. The default value for `ReturnValues` is `None`, meaning that DynamoDB will not return any information about attributes that were modified.

The following are additional settings for `ReturnValues`, organized by DynamoDB API operation:

**PutItem** The `PutItem` action creates a new item or replaces an old item with a new item. You can return the item's attribute values in the same operation by using the `ReturnValues` parameter.

#### **ReturnValues: ALL\_OLD**

- If you overwrite an existing item, `ALL_OLD` returns the entire item as it appeared before the overwrite.
- If you write a nonexistent item, `ALL_OLD` has no effect.

**UpdateItem** The most common use for `UpdateItem` is to update an existing item. However, `UpdateItem` actually performs an *upsert*, meaning that it will automatically create the item if it does not already exist.

#### **ReturnValues: ALL\_OLD**

- If you update an existing item, `ALL_OLD` returns the entire item as it appeared before the update.
- If you update a nonexistent item (upsert), `ALL_OLD` has no effect.

#### **ReturnValues: ALL\_NEW**

- If you update an existing item, `ALL_NEW` returns the entire item as it appeared after the update.
- If you update a nonexistent item (upsert), `ALL_NEW` returns the entire item.

#### **ReturnValues: UPDATED\_OLD**

- If you update an existing item, `UPDATED_OLD` returns only the updated attributes as they appeared before the update.
- If you update a nonexistent item (upsert), `UPDATED_OLD` has no effect.

#### **ReturnValues: UPDATED\_NEW**

- If you update an existing item, `UPDATED_NEW` returns only the affected attributes as they appeared after the update.
- If you update a nonexistent item (upsert), `UPDATED_NEW` returns only the updated attributes as they appear after the update.

**DeleteItem** The `DeleteItem` deletes a single item in a table by primary key. You can perform a conditional delete operation that deletes the item if it exists, or if it has an expected attribute value.

#### **ReturnValues: ALL\_OLD**

- If you delete an existing item, `ALL_OLD` returns the entire item as it appeared before you deleted it.
- If you delete a nonexistent item, `ALL_OLD` does not return any data.

## Requesting Throttle and Burst Capacity

If your application performs reads or writes at a higher rate than your table can support, DynamoDB begins to *throttle* those requests. When DynamoDB throttles a read or write, it returns a ProvisionedThroughputExceededException to the caller. The application can then take appropriate action, such as waiting for a short interval before retrying the request.

The AWS SDKs provide built-in support for retrying throttled requests; you do not need to write this logic yourself. The DynamoDB console displays CloudWatch metrics for your tables so that you can monitor throttled read requests and write requests. If you encounter excessive throttling, consider increasing your table's provisioned throughput settings.

In some cases, DynamoDB uses *burst capacity* to accommodate reads or writes in excess of your table's throughput settings. With burst capacity, unexpected read or write requests can succeed where they otherwise would be throttled. Burst capacity is available on a best-effort basis, and DynamoDB does not verify that this capacity is always available.

## Amazon DynamoDB Secondary Indexes: Global and Local

A *secondary index* is a data structure that contains a subset of attributes from a table. The index uses an alternate key to support Query operations in addition to making queries against the primary key. You can retrieve data from the index using a Query. A table can have multiple secondary indexes, which give your applications access to many different Query patterns.

You can create one or more secondary indexes on a table. DynamoDB does not require indexes, but indexes give your applications more flexibility when you query your data. After you create a secondary index on a table, you can read or scan data from the index in much the same way as you do from the table.

DynamoDB supports the following kinds of indexes:

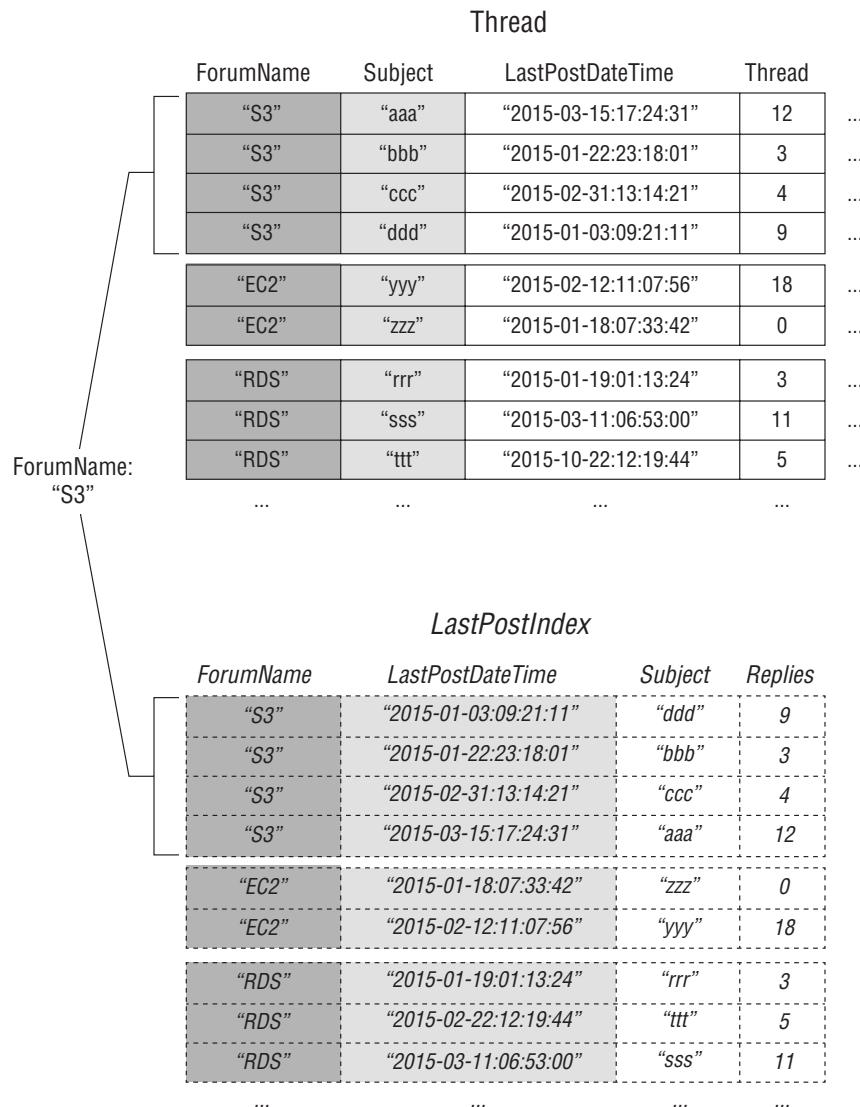
*Global secondary index* A *global secondary index* is one with a partition key and sort key that can be different from those on the table.

*Local secondary index* A *local secondary index* is one that has the same partition key as the table but a different sort key.



You can define up to five global secondary indexes and five local secondary indexes per table. You can also scan an index as you would a table.

Figure 14.1 shows a local secondary index for a DynamoDB table of forum posts. The local secondary index allows you to query based on the date and time of the last post to a subject, as opposed to the subject itself.

**FIGURE 14.1** Amazon DynamoDB indexes

Every secondary index is associated with exactly one table from which it obtains its data; it is the *base table* for the index. DynamoDB maintains indexes automatically. When you add, update, or delete an item in the base table, DynamoDB makes the change to the item in any indexes that belong to that table. When you create an index, you configure which attributes copy (project) from the base table to the index. At a minimum, DynamoDB projects the key attributes from the base table into the index.

When you create an index, you define an *alternate key* (partition key and sort key) for the index. You also define the attributes that you want to project from the base table into the index. DynamoDB copies these attributes into the index along with the primary key attributes from the base table. You can Query or Scan the index like a table.

Consider your application's requirements when you determine which type of index to use. Table 14.2 shows the main differences between a global secondary index and a local secondary index.

**TABLE 14.2** Global vs. Secondary Indexes

Characteristic	Global Secondary Index	Local Secondary Index
Key Schema	The primary key can be simple (partition key) or composite (partition key and sort key).	The primary key must be composite (partition key and sort key).
Key Attributes	The index partition key and sort key (if present) can be any base table attributes of type string, number, or binary.	The partition key of the index is the same attribute as the partition key of the base table. The sort key can be any base table attribute of type string, number, or binary.
Size Restrictions Per Partition Key Value	No size restrictions.	No size restrictions.
Online Index Operations	Create at the same time that you create a table. You can also add a new global secondary index to an existing table or delete an existing global secondary index.	Create at the same time that you create a table. You cannot add a local secondary index to an existing table, nor can you delete any local secondary indexes that currently exist.
Queries and Partitions	Query over the entire table, across all partitions.	Query over a single partition, as specified by the partition key value in the query.
Read Consistency	Query on eventual consistency only.	Query eventual consistency or strong consistency.
Provisioned Throughput Consumption	Every global secondary index has its own provisioned throughput settings for read and write activity. Queries, scans, and updates consume capacity units from the index, not from the base table.	Query or scan consumes read capacity units from the base table. Writes and write updates consume write capacity units from the base table.
Projected Attributes	Queries or scans can only request the attributes that project into the index. DynamoDB will <i>not</i> fetch any attributes from the table.	Queries or scans can request attributes that do not project into the index. DynamoDB will automatically fetch those attributes from the table.

If you write an item to a table, you do not have to configure the attributes for any global secondary index sort key. A table with many global secondary indexes incurs higher costs for write activity than tables with fewer indexes. For maximum query flexibility, you can create up to five global secondary indexes and up to five local secondary indexes per table.

To create more than one table with secondary indexes, you must do so sequentially. Create the first table and wait for it to become active, then create the next table and wait for it to become active, and so on. If you attempt to create more than one table with a secondary index at a time, DynamoDB responds with a `LimitExceededException` error.

For each secondary index, you must configure the following:

**Type of index** The type of index to be created can be either a global secondary index or a local secondary index.

**Name of index** The naming rules for indexes are the same as those for table. The name must be unique for the base table, but you can use the same name for indexes that you associate with different base tables.

**Index key schema** Every attribute in the index key schema must be a top-level attribute of type string, number, or binary. Other data types, including documents and sets, are not allowed. Other requirements for the key schema depend on the type of index:

**Global secondary index** For a global secondary index, the partition key can be any scalar attribute of the base table. A sort key is optional, and it can be any scalar attribute of the base table.

**Local secondary index** For a local secondary index, the partition key must be the same as the base table's partition key, and the sort key must be a non-key base table attribute.

**Additional attributes** These attributes are in addition to the table's key attributes, which automatically project into every index. You can project attributes of any data type, including scalars, documents, and sets.

**Global secondary index** For a *global secondary index*, you must configure *read and write capacity unit settings*. These provisioned throughput settings are independent of the base table's settings.

**Local secondary index** For a *local secondary index*, you do not need to configure *read and write capacity unit settings*. Any read and write operations on a local secondary index draw from the provisioned throughput settings of its base table.

To generate a detailed list of secondary indexes on a table, use the `DescribeTable` operation. `DescribeTable` returns the name, storage size, and item counts for every secondary index on the table. These values refresh approximately every six hours.

Use the `Query` or `Scan` operation to access the data in a secondary index. You configure the *base table name, index, attributes to return in the results*, and any *condition expressions or filters* that you want to apply. DynamoDB returns the results in ascending or descending order.



When you delete a table, all indexes associated with that table are deleted.

## Global Secondary Indexes

Some applications may need to perform many kinds of queries, using a variety of different attributes as query criteria. To support these requirements, you can create one or more global secondary indexes and then issue query requests against these indexes.

To illustrate, Figure 14.2 displays the GameScores table, which tracks users and scores for a mobile gaming application. Each item in GameScores has a partition key (`UserId`) and a sort key (`GameTitle`). Figure 14.2 shows the organization of the items.

**FIGURE 14.2** Game scores

GameScores						
UserId	GameTitle	TopScore	TopScoreDateTime	Wins	Losses	
“101”	“Galaxy Invaders”	5842	“2015-09-15:17:24:31”	21	72	...
“101”	“Meteor Blasters”	1000	“2015-10-22:23:18:01”	12	3	...
“101”	“Starship X”	24	“2015-08-31:13:14:21”	4	9	...
“102”	“Alien Adventure”	192	“2015-07-12:11:07:56”	32	192	...
“102”	“Galaxy Invaders”	0	“2015-09-18:07:33:42”	0	5	...
“103”	“Attack Ships”	3	“2015-10-19:01:13:24”	1	8	...
“103”	“Galaxy Invaders”	2317	“2015-09-11:06:53:00”	40	3	...
“103”	“Meteor Blasters”	723	“2015-10-19:01:13:24”	22	12	...
“103”	“Starship X”	42	“2015-07-11:06:53:00”	4	19	...
...	...	...	...	...	...	...

To write a leaderboard application to display top scores for each game, you could generate a query that specifies the key attributes (`UserId` and `GameTitle`). While this would be efficient for the application to retrieve data from `GameScores` based on `GameTitle` only, it would need to use a `Scan` operation. As you add more items to the table, `Scan` operations of all the data becomes slow and inefficient, making it difficult to answer questions based on Figure 14.2, such as the following:

- What is the top score ever recorded for the game Meteor Blasters?
- Which user had the highest score for Galaxy Invaders?
- What was the highest ratio of wins versus losses?

To better implement queries on non-key attributes, create a global secondary index. A global secondary index contains a selection of attributes from the base table, but you organize them by a primary key that is different from that of the table. The index key does not require any of the key attributes from the table, nor does it require the same key schema as a table.

Every global secondary index must have a partition key and can have an optional sort key. The index key schema can be different from the base table schema. You could have a table with a simple primary key (partition key) and create a global secondary index with a composite primary key (partition key and sort key) or vice versa. The index key attributes can consist of any top-level string, number, or binary attributes from the base table but not other scalar types, document types, and set types.



You can project other base table attributes into the index. When you query the index, DynamoDB can retrieve these projected attributes efficiently; however, global secondary index queries cannot fetch attributes from the base table. *In a DynamoDB table, each key value must be unique.* However, the key values in a global secondary index do not need to be unique. A global secondary index tracks data items only where the key attribute or attributes actually exist.

## Attribute Projections

A *projection* is the set of attributes the secondary index copies from a table. While the partition key and sort key of the table project into the index, you can also project other attributes to support your application's Query requirements. When you query an index, DynamoDB accesses any attribute in the projection as if those attributes were in a table of their own.

When you create a secondary index, configure the attributes that project into the index. DynamoDB provides the following options:

**KEYS\_ONLY** Each item in the index consists only of the table partition key and sort key values, plus the index key values, and this results in the smallest possible secondary index.

**INCLUDE** Each item in the index consists only of the table partition key and sort key values plus the index key values, and it includes other non-key attributes that you configure.

**ALL** Includes all attributes from the source table, including other non-key attributes that you configure. Because the table data is duplicated in the index, an ALL projection results in the largest possible secondary index.

When you choose the attributes to project into a global secondary index, consider the provisioned throughput costs and the storage costs:

- Before accessing a few attributes with the lowest possible latency, consider projecting only those attributes into a global secondary index. The smaller the index, the less it costs to store it and the lower your write costs will be.

- If your application will frequently access non-key attributes, consider projecting those attributes into a global secondary index. The additional storage costs for the global secondary index offset the cost of performing frequent table scans.
- When you’re accessing most of the non-key attributes frequently, project these attributes, or even the entire base table, into a global secondary index. This provides maximum flexibility; however, your storage cost would increase or even double.
- If your application needs to query a table infrequently but must perform many writes or updates against the data in the table, consider projecting KEYS\_ONLY. The global secondary index would be of minimal size but would still be available for query activity.

## Querying a Global Secondary Index

Use the Query operation to access one or more items in a global secondary index. The query must specify the *name of the base table*, the *name of the index*, the *attributes* the query results return, and any query *conditions* that you want to apply. DynamoDB can return the results in ascending or descending order.

Consider the following example in which a query requests game data for a leaderboard application:

```
{
  "TableName": "GameScores",
  "IndexName": "GameTitleIndex",
  "KeyConditionExpression": "GameTitle = :v_title",
  "ExpressionAttributeValues": {
    ":v_title": {"S": "Meteor Blasters"}
  },
  "ProjectionExpression": "UserId, TopScore",
  "ScanIndexForward": false
}
```

In this query, the following actions occur:

- DynamoDB accesses GameTitleIndex, using the GameTitle partition key to locate the index items for Meteor Blasters. All index items with this partition key are next to each other for rapid retrieval.
- Within this game, DynamoDB uses the index to access the UserID and TopScore for this game.
- The query results return in descending order, as the ScanIndexForward parameter is set to false.

## Scanning a Global Secondary Index

You can use the Scan operation to retrieve the data from a global secondary index. Provide the *base table name* and the *index name* in the request. With a Scan operation, DynamoDB

reads the data in the index and returns it to the application. You can also request only some of the data and to discard the residual data. To do this, use the `FilterExpression` parameter of the `Scan` operation.

## Synchronizing Data between Tables and Global Secondary Indexes

DynamoDB automatically synchronizes each global secondary index with its base table. When an application writes or deletes items in a table, any global secondary indexes on that table update asynchronously by using an eventually consistent model. Though applications seldom write directly to an index, understand the following the implications of how DynamoDB maintains these indexes:

- When you create a global secondary index, you configure one or more index key attributes and their data types.
- When you write an item to the base table, the data types for those attributes must match the index key schema's data types.
- When you put or delete items in a table, the global secondary indexes on that table update in an eventually consistent fashion.



### Long Global Index Propagations

Under normal conditions, changes to the table data propagate to the global secondary indexes within a fraction of a second. However, if an unlikely failure scenario occurs, longer propagation delays may occur. Because of this, your applications need to anticipate and handle situations where a query on a global secondary index returns results that are not current.

## Considerations for Provisioned Throughput of Global Secondary Indexes

When you create a global secondary index, you must configure read and write capacity units for the workload that you expect on that index. The provisioned throughput settings of a global secondary index are separate from those of its base table. A `Query` operation on a global secondary index consumes read capacity units from the index, not the base table.

When you put, update, or delete items in a table, the global secondary indexes on that table are updated. These index updates consume write capacity units from the index, not from the base table.

To view the provisioned throughput settings for a global secondary index, use the `DescribeTable` operation, and detailed information about the table's global secondary indexes return.



If you query a global secondary index and exceed its provisioned read capacity, your request throttles. If you perform heavy write activity on the table but a global secondary index on that table has insufficient write capacity, then the write activity on the table throttles.

To avoid potential throttling, the provisioned write capacity for a global secondary index should be equal to or greater than the write capacity of the base table because new updates write to both the base table and global secondary index.

## Read Capacity Units

Global secondary indexes support eventually consistent reads, each of which consume one-half of a *read capacity unit*. For example, a single global secondary index query can retrieve up to 8 KB ( $2 \times 4$  KB) per read capacity unit. For global secondary index queries, DynamoDB calculates the provisioned read activity in the same way that it does for queries against tables, except that the calculation is based on the sizes of the index entries instead of the size of the item in the base table. The number of read capacity units is the sum of all projected attribute sizes across all returned items; the result is then rounded up to the next 4-KB boundary.

The maximum size of the results returned by a Query operation is 1 MB; this includes the sizes of all of the attribute names and values across all returned items.

For example, if a global secondary index contains items with 2,000 bytes of data and a query returns 8 items, then the total size of the matching items is  $2,000 \text{ bytes} \times 8 \text{ items} = 16,000 \text{ bytes}$ ; this is then rounded up to the nearest 4-KB boundary. Because global secondary index queries are eventually consistent, the total cost is  $0.5 \times (16 \text{ KB}/4 \text{ KB})$ , or two read capacity units.

## Write Capacity Units

When you add, update, or delete an item in a table and a global secondary index is affected by this, then the global secondary index consumes provisioned write capacity units for the operation. The total provisioned throughput cost for a write consists of the sum of the write capacity units consumed by writing to the base table and those consumed by updating the global secondary indexes. If a write to a table does not require a global secondary index update, then no write capacity is consumed from the index.

For a table write to succeed, the provisioned throughput settings for the table and all of its global secondary indexes must have enough write capacity to accommodate the write; otherwise, the write to the table will throttle.

## Factors Affecting Cost of Writes

The cost of writing an item to a global secondary index depends on the following factors:

- If you write a new item to the table that defines an indexed attribute or you update an existing item to define a previously undefined indexed attribute, one write operation is required to put the item into the index.

- If an update to the table changes the value of an indexed key attribute (from A to B), two writes are required—one to delete the previous item from the index and another write to put the new item into the index.
- If an item was present in the index, but a write to the table caused the indexed attribute to be deleted, one write is required to delete the old item projection from the index.
- If an item is not present in the index before or after the item is updated, there is no additional write cost for the index.
- If an update to the table changes the value of only projected attributes in the index key schema but does not change the value of any indexed key attribute, then one write is required to update the values of the projected attributes into the index.

All of these factors assume that the size of each item in the index is less than or equal to the 1-KB item size for calculating write capacity units. Larger index entries require additional write capacity units. Minimize your write costs by considering which attributes your queries must return and projecting only those attributes into the index.

## Considerations for Storing Global Secondary Indexes

When an application writes an item to a table, DynamoDB automatically copies the correct subset of attributes to any global secondary indexes in which those attributes should appear. Your account is charged for storing the item in the base table and also for storing attributes in any global secondary indexes on that table.

The amount of space used by an index item is the sum of the following:

- Size in bytes of the base table primary key (partition key and sort key)
- Size in bytes of the index key attribute
- Size in bytes of the projected attributes (if any)
- 100 bytes of overhead per index item

To estimate the storage requirements for a global secondary index, estimate the average size of an item in the index and then multiply by the number of items in the base table that have the global secondary index key attributes.

If a table contains an item for which a particular attribute is not defined but that attribute is defined as an index partition key or sort key, DynamoDB does not write any data for that item to the index.

## Managing Global Secondary Indexes

Global secondary indexes require you to create, describe, modify, delete, and detect index key violations.

## Creating a Table with Global Secondary Indexes

To create a table with one or more global secondary indexes, use the `CreateTable` operation with the `GlobalSecondaryIndexes` parameter. For maximum query flexibility, create up to *five* global secondary indexes per table. *Specify one attribute to act as the index*

*partition key*. You can specify another attribute for the index sort key. It is not necessary for either of these key attributes to be the same as a key attribute in the table.

Each index key attribute must be a scalar of type string, number, or binary, and cannot be a document or a set. You can project attributes of any data type into a global secondary index, including scalars, documents, and sets. You must also provide ProvisionedThroughput settings for the index, consisting of ReadCapacityUnits and WriteCapacityUnits. These provisioned throughput settings are separate from those of the table but behave in similar ways.

## Viewing the Status of Global Secondary Indexes on a Table

To view the status of all the global secondary indexes on a table, use the `DescribeTable` operation. The `GlobalSecondaryIndexes` portion of the response shows all indexes on the table, along with the current status of each (`IndexStatus`).

The `IndexStatus` for a global secondary index is as follows:

**Creating** Index is currently being created, and it is not yet available for use.

**Active** Index is ready for use, and the application can perform `Query` operations on the index.

**Updating** Provisioned throughput settings of the index are being changed.

**Deleting** Index is currently being deleted, and it can no longer be used.

When DynamoDB has finished building a global secondary index, the index status changes from `Creating` to `Active`.

## Adding a Global Secondary Index to an Existing Table

To add a global secondary index to an existing table, use the `UpdateTable` operation with the `GlobalSecondaryIndexUpdates` parameter, and provide the following information:

- An *index name*, which must be unique among all of the indexes on the table.
- The *key schema* of the index. Configure *one* attribute for the index *partition key*. You can configure another attribute for the index *sort key*. It is not necessary for either of these key attributes to be the same as a key attribute in the table. The data types for each schema attribute must be scalar: string, number, or binary.
- The attributes to project from the table into the index include the following:

**KEYS\_ONLY** Each item in the index consists of only the table partition key and sort key values, plus the index key values.

**INCLUDE** In addition to the attributes described in `KEYS_ONLY`, the secondary index includes other non-key attributes that you configure.

**ALL** The index includes all attributes from the source table.

- The provisioned throughput settings for the index, consisting of `ReadCapacityUnits` and `WriteCapacityUnits`. These provisioned throughput settings are separate from those of the table.



You can create only *one* global secondary index per `UpdateTable` operation, and you cannot cancel a global secondary index creation process.

## Resource Allocation

DynamoDB allocates the compute and storage resources to build the index. During the resource allocation phase, the `IndexStatus` attribute is `CREATING` and the `Backfilling` attribute is `false`. Use the `DescribeTable` operation to retrieve the status of a table and all of its secondary indexes.

While the index is in the resource allocation phase, you cannot delete its parent table, nor can you modify the provisioned throughput of the index or the table. You cannot add or delete other indexes on the table; however, you can modify the provisioned throughput of these other indexes.

## Backfilling

For each item in the table, DynamoDB determines which set of attributes to write to the index based on its projection (`KEYS_ONLY`, `INCLUDE`, or `ALL`). It then writes these attributes to the index. During the backfill phase, DynamoDB tracks items that you add, delete, or update in the table and the attributes in the index.

During the backfilling phase, the `IndexStatus` attribute is `CREATING` and the `Backfilling` attribute is `true`. Use the `DescribeTable` operation to retrieve the status of a table and all of its secondary indexes.

While the index is backfilling, you cannot delete its parent table. However, you can still modify the provisioned throughput of the table and any of its global secondary indexes.

When the index build is complete, its status changes to `Active`. You are not able to query or scan the index until it is `Active`.

### Restrictions and Limitations of Backfilling

During the backfilling phase, some writes of violating index items may succeed while others are rejected. This can occur if the data type of an attribute value does not match the data type of an index key schema data type or if the size of an attribute exceeds the maximum length for an index key attribute.

Index key violations do not interfere with global secondary index creation; however, when the index becomes `Active`, the violating keys will not be present in the index. After backfilling, all writes to items that violate the new index's key schema will be rejected. To detect and resolve any key violations that may have occurred, run the Violation Detector tool after the backfill phase completes.

While the resource allocation and backfilling phases are in progress, the index is in the `CREATING` state. During this time, DynamoDB performs read operations on the table; you are not charged for this read activity.

You cannot cancel an in-flight global secondary index creation.

## Detecting and Correcting Index Key Violations

Throughout the backfill phase of the global secondary index creation, DynamoDB examines each item in the table to determine whether it is eligible for inclusion in the index, because noneligible items cause index key violations. In these cases, the items remain in the table, but the index will not have a corresponding entry for that item.

An *index key violation* occurs if:

- There is a *data type mismatch* between an attribute value and the index key schema data type. For example, in Figure 14.1, if one of the items in the GameScores table had a TopScore value of type “string,” and you add a global secondary index with a number-type partition key of TopScore, the item from the table would violate the index key.
- An attribute value from the table exceeds the maximum length for an index key attribute. The *maximum length of a partition key is 2,048 bytes*, and the *maximum length of a sort key is 1,024 bytes*. If any of the corresponding attribute values in the table exceed these limits, the item from the table violates the index key.

If an index key violation occurs, the backfill phase continues without interruption; however, *any violating items are not included in the index*. After the backfill phase completes, all writes to items that violate the new index’s key schema will be rejected.

## Deleting a Global Secondary Index from a Table

You use the `UpdateTable` operation to delete a global secondary index. While the global secondary index is being deleted, there is *no effect* on any read or write activity in the parent table, and you can still modify the provisioned throughput on other indexes. You can delete only one global secondary index per `UpdateTable` operation.



When you delete a table (`DeleteTable`), all of the global secondary indexes on that table are deleted.

## Local Secondary Indexes

Some applications query data by using only the base table’s primary key; however, there may be situations where an *alternate sort key* would be helpful. To give your application a choice of sort keys, create one or more local secondary indexes on a table and issue `Query` or `Scan` requests against these indexes.

For example, Figure 14.3 is useful for an application such as discussion forums. The figure shows how the items in the table would be organized.

**FIGURE 14.3** Forum thread table

Thread				
ForumName	Subject	LastPostDateTime	Replies	
“S3”	“aaa”	“2015-03-15:17:24:31”	12	...
“S3”	“bbb”	“2015-01-22:23:18:01”	3	...
“S3”	“ccc”	“2015-02-31:13:14:21”	4	...
“S3”	“ddd”	“2015-01-03:09:21:11”	9	...
“EC2”	“yyy”	“2015-02-12:11:07:56”	18	...
“EC2”	“zzz”	“2015-01-18:07:33:42”	0	...
“RDS”	“rrr”	“2015-01-19:01:13:24”	3	...
“RDS”	“sss”	“2015-03-11:06:53:00”	11	...
“RDS”	“ttt”	“2015-10-22:12:19:44”	5	...
...	...	...	...	...

DynamoDB stores all items with the same partition key value contiguously. In this example, given a particular ForumName, a Query operation could immediately locate the threads for that forum. Within a group of items with the same partition key value, the items are sorted by sort key value. If the sort key (Subject) is also provided in the Query operation, DynamoDB can narrow the results that are returned, such as returning the threads in the S3 forum that have a Subject beginning with the letter a.

Requests may require more complex data-access patterns, such as the following:

- Which forum threads receive the most views and replies?
- Which thread in a particular forum contains the largest number of messages?
- How many threads were posted in a particular forum, within a particular time period?

To answer these questions, the Query action would not be sufficient. Instead, you must scan the entire table. For a table with millions of items, this would consume a large amount of provisioned read throughput and time. However, you can configure one or more local secondary indexes on non-key attributes, such as Replies or LastPostDateTime.

A *local secondary index* maintains an *alternate sort key* for a given partition key value. A local secondary index also contains a copy of some, or all, of the attributes from its base table. You configure which attributes project into the local secondary index when you create the table. *The data in a local secondary index is organized by the same partition key as the base table but with a different sort key.* This enables you to access data items efficiently across this different dimension. For greater Query or Scan flexibility, create up to five local secondary indexes per table.

If an application locates the threads that have been posted within the last three months but lacks a local secondary index, the application must scan the entire thread table and discard any posts that were not listed within the specified time frame. With a local secondary index, a Query operation could use `LastPostDateTime` as a sort key and find the data quickly.

Figure 14.4 shows a local secondary index named `LastPostIndex`. The partition key is the same as that of the Thread table (see Figure 14.3), but the sort key is `LastPostDateTime`.

**FIGURE 14.4** Last post index

<i>LastPostIndex</i>		
<i>ForumName</i>	<i>LastPostDateTime</i>	<i>Subject</i>
“S3”	“2015-01-03:09:21:11”	“ddd”
	“2015-01-22:23:18:01”	“bbb”
	“2015-02-31:13:14:21”	“ccc”
	“2015-03-15:17:24:31”	“aaa”
“EC2”	“2015-01-18:07:33:42”	“zzz”
	“2015-02-12:11:07:56”	“yyy”
“RDS”	“2015-01-19:01:13:24”	“rrr”
	“2015-02-22:12:19:44”	“ttt”
	“2015-03-11:06:53:00”	“sss”
...	...	...

Every local secondary index must meet the following conditions:

- The partition key is the same as that of its base table.
- The sort key consists of exactly one scalar attribute.
- The sort key of the base table projects into the index, where it acts as a non-key attribute.

In Figure 14.4, the partition key is `ForumName`, and the sort key of the local secondary index is `LastPostDateTime`. In addition, the sort key value from the base table (`Subject`) projects into the index, but it is not a part of the index key. If an application needs a list that is based on `ForumName` and `LastPostDateTime`, it can issue a Query request against `LastPostIndex`. The query results sort by `LastPostDateTime` and can return in ascending or descending order. The query can also apply key conditions, such as returning only items that have a `LastPostDateTime` within a particular time span.

Every local secondary index automatically contains the partition and sort keys from its base table, so you can project non-key attributes into the index. When you query the index, DynamoDB can retrieve these projected attributes efficiently. When you query a local secondary index, the Query operation can also retrieve attributes that do not project into the index. DynamoDB automatically collects these attributes from the base table but at a greater latency and with higher provisioned throughput costs.



For any local secondary index, you can store up to 10 GB of data per distinct partition key value.

## Creating a Local Secondary Index

To create one or more local secondary indexes on a table, use the `LocalSecondaryIndexes` parameter of the `CreateTable` operation. You create local secondary indexes when you create the table. When you delete a table, any local secondary indexes on that table are also deleted. *Configure one non-key attribute to act as the sort key of the local secondary index.* The local secondary indexes attribute is scalar and includes string, number, binary, document types, and set types. You can project attributes of any data type into a local secondary index.



For tables with local secondary indexes, there is a 10-GB size limit per partition key value. A table with local secondary indexes can store any number of items, as long as the total size for any one partition key value does not exceed 10 GB.

## Querying a Local Secondary Index

In a DynamoDB table, the combined partition key value and sort key value for each item must be unique. However, in a local secondary index, the sort key value does not need to be unique for a given partition key value. If there are multiple items in the local secondary index that have the same sort key value, a Query operation returns all items with the same partition key value. In the response, the items that the query locates do not return in any particular order.

You can query a local secondary index using eventually consistent or strongly consistent reads. To configure which type of consistency you want, use the `ConsistentRead` parameter of the `Query` operation. A strongly consistent read from a local secondary index returns the latest updated values. If the `Query` operation must collect additional attributes from the base table, those attributes will be consistent with respect to the index.

## Scanning a Local Secondary Index

You can use the `Scan` function to retrieve all data from a local secondary index. Provide the *base table name* and the *index name* in the request. With a `Scan` function, DynamoDB reads the data in the index and returns it to the application. You can also scan for specific data to return and discard the other data using the `FilterExpression` parameter of the `Scan` API.

## Item Writes and Local Secondary Indexes

DynamoDB automatically keeps all local secondary indexes synchronized with their respective base tables. Applications seldom write directly to an index. However, understand the implications of how DynamoDB maintains these indexes.

*When you create a local secondary index, configure an attribute to serve as the sort key for the index and configure a data type for that attribute.* Whenever you write an item to the base table, if the item defines an index key attribute, its type must match the index key schema's data type.

There is no requirement for a one-to-one relationship between the items in a base table and the items in a local secondary index. This behavior can be advantageous for many applications, because a table with many local secondary indexes incurs higher costs for write activity than tables with fewer indexes.

## Provisioned Throughput for Local Secondary Indexes

When you create a table in DynamoDB, you provision read and write capacity units for the table's expected workload, which includes read and write activity on the table's local secondary indexes.

### Read Capacity Units

When you query a local secondary index, the number of read capacity units consumed depends on how you access the data. As with table queries, an index query can use eventually consistent reads or strongly consistent reads, depending on the value of `ConsistentRead`. One strongly consistent read consumes one read capacity unit, but an eventually consistent read consumes only half of that. By choosing eventually consistent reads, you can reduce your read capacity unit charges.

For index queries that request only index keys and projected attributes, DynamoDB calculates the provisioned read activity in the same way that it does for queries against tables. However, the calculation is based on the sizes of the index entries instead of the size of the item in the base table. The number of read capacity units is the sum of all projected attribute sizes across all items returned. The result is then rounded up to the next 4-KB boundary.

For index queries that read attributes do not project into the local secondary index, and DynamoDB must collect those attributes from the base table in addition to reading the projected attributes from the index. These collections occur when you include any non-projected [per DynamoDB documentation] attributes in the `Select` or `ProjectionExpression` parameters of the `Query` operation. Fetching causes additional latency in query responses, and it incurs a higher provisioned throughput cost. In addition to the reads from the local secondary index, you are charged for read capacity units for every base table item fetched. This charge is for reading each entire item from the table, not only the requested attributes.

The maximum size of the results returned by a `Query` operation is 1 MB. This includes the sizes of all of the attribute names and values across all items returned. However, if a query against a local secondary index causes DynamoDB to fetch item attributes from the

base table, the maximum size of the data in the results may be lower. In this case, the result size is the sum of the following factors:

- The size of the matching items in the index, rounded up to the next 4 KB
- The size of each matching item in the base table, with each item individually rounded up to the next 4 KB

Using this formula, the maximum size of the results returned by a Query operation is still 1 MB.

### Example 2: Query Read Capacity Units for Local Secondary Index

A table has items the size of 300 bytes. There is a local secondary index on that table, but only 200 bytes of each item projects into the index. If you query this index, the query requires table fetches for each item, and the query returns four items. DynamoDB sums up the following:

- Size of the matching items in the index:  $200 \text{ bytes} \times 4 \text{ items} = 800 \text{ bytes}$ ; this rounds up to 4 KB.
- Size of each matching item in the base table:  $(300 \text{ bytes, rounds up to 4 KB}) \times 4 \text{ items} = 16 \text{ KB}$ .

The total size of the data in the result is therefore 20 KB.

## Write Capacity Units

When you add, update, or delete items in a table, the local secondary indexes consume provisioned write capacity units for the table. The total provisioned throughput cost for a write is the sum of write capacity units consumed by the write to the table and those consumed by the update of the local secondary indexes.

The cost of writing an item to a local secondary index depends on the following factors:

- If you write a new item to the table that defines an indexed attribute or you update an existing item to define a previously undefined indexed attribute, one write operation is required to put the item into the index.
- If an update to the table changes the value of an indexed key attribute (from A to B), two writes are required, one to delete the previous item from the index and another write to put the new item into the index.
- If an item was present in the index but a write to the table caused the indexed attribute to be deleted, one write is required to delete the old item projection from the index.
- If an item is not present in the index before or after the item update, there is no additional write cost for the index.

All of these factors assume that the size of each item in the index is less than or equal to the 1-KB item size for calculating write capacity units. Larger index entries require additional write capacity units. You can minimize your write costs by considering which attributes your queries must return and project only those attributes into the index.

## Storage for Local Secondary Indexes

When an application writes an item to a table, DynamoDB automatically copies the correct subset of attributes to any local secondary indexes in which those attributes should appear. Your account is charged for storing the item in the base table and also for storing attributes in any local secondary indexes on that table.

The amount of space used by an index item is the sum of the following elements:

- Size in bytes of the base table primary key (partition and sort key)
- Size in bytes of the index key attribute
- Size in bytes of the projected attributes (if any)
- 100 bytes of overhead per index item

To estimate the storage requirements for a local secondary index, estimate the average size of an item in the index and then multiply by the number of items in the base table.

If a table contains an item where a particular attribute is not defined but that attribute is defined as an index sort key, then DynamoDB does not write any data for that item to the index.

## Amazon DynamoDB Streams

*Amazon DynamoDB Streams* captures data modification events in DynamoDB tables. The data about these events appear in the stream in near real time and in the order that the events occurred.

Each event represents a stream record. When you enable a stream on a table, DynamoDB captures information about every modification to data items in the table. The stream captures an image of the entire item, including all of its attributes. A *stream record* contains information about a data modification to a single item in a DynamoDB table including the primary key attributes of the items. You can configure the stream so that the stream records capture additional information, such as the “before” and “after” images of modified items. Finally, a stream record is written when an item is deleted from the table, and each stream record also contains the name of the table, the event timestamp, and other metadata.



Stream records have a lifetime of 24 hours, after which they are deleted automatically from the stream.

A DynamoDB stream is a time-ordered flow of information of item-level modifications (create, update, or delete) to items in a DynamoDB table.

DynamoDB Streams does the following:

- Each stream record appears exactly *once* in the stream.
- For each item that is modified in a DynamoDB table, the stream records appear in the *same sequence* as the actual modifications to the item.

Many applications benefit from the ability to capture changes to items stored in a DynamoDB table when such changes occur. The following are common scenarios:

- An application in one AWS Region modifies the data in a Amazon DynamoDB table. A second application in another AWS Region reads these data modifications and writes the data to another table, creating a replica that stays in sync with the original table.
- A popular mobile app modifies data in a DynamoDB table at the rate of thousands of updates per second. Another application captures and stores data about these updates, providing near-real-time usage metrics for the mobile app.
- A global multiplayer game has a multi-master topology, storing data in multiple AWS Regions. Each master stays in sync by consuming and replaying the changes that occur in the remote regions.
- An application automatically sends notifications to the mobile devices of all friends in a group as soon as one friend uploads a new picture.
- A new customer adds data to a DynamoDB table. This event invokes another application that sends a welcome email to the new customer.

Whenever an application creates, updates, or deletes items in the table, Amazon DynamoDB Stream writes a stream record with the primary key attribute, or attributes, of the items that were modified. A stream record contains information about a data modification to a single item in a DynamoDB table. Applications can access this log and view the data items as they appeared before and after they were modified, in near real time.



DynamoDB Streams writes stream records in near-real time, so you can build applications that consume these streams and act based on the contents.

## DynamoDB Cross-Region Replication

You can create tables that automatically replicate across two or more AWS Regions with full support for multi-master writes. Using cross-region replication, you can build fast, massively scaled applications for a global user base without having to manage the replication process.

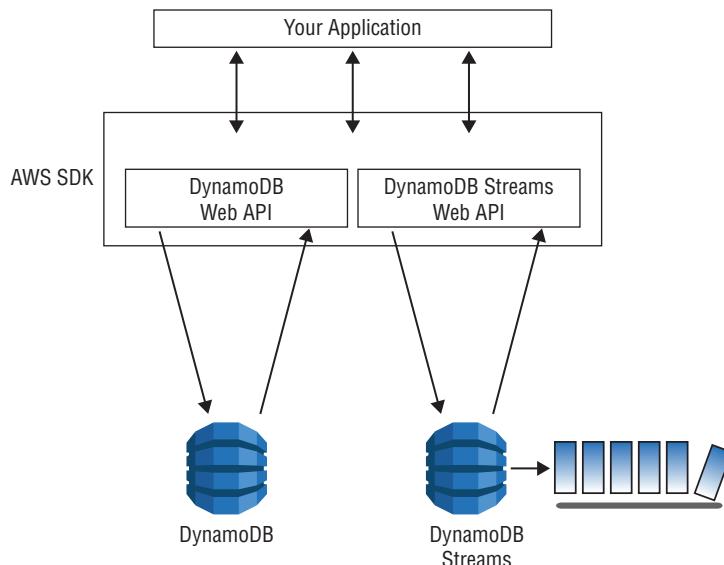
## DynamoDB Stream Endpoints

AWS maintains separate endpoints for DynamoDB and DynamoDB Streams. To work with database tables and indexes, your application must access a DynamoDB endpoint. To

read and process DynamoDB Streams records, your application must access a DynamoDB Streams endpoint in the same AWS Region.

Figure 14.5 shows the DynamoDB endpoint flow.

**FIGURE 14.5** DynamoDB Streams endpoints



The naming convention for DynamoDB Streams endpoints is `streams.dynamodb.<region>.amazonaws.com`. For example, if you use the endpoint `dynamodb.us-west-2.amazonaws.com` to access DynamoDB, use the endpoint `streams.dynamodb.us-west-2.amazonaws.com` to access DynamoDB Streams.

The AWS SDKs provide separate clients for DynamoDB and DynamoDB Streams. Depending on your requirements, your application can access a DynamoDB endpoint, a DynamoDB Streams endpoint, or both at the same time. To connect to both endpoints, your application must instantiate two clients: one for DynamoDB and one for DynamoDB Streams.

## Enabling a Stream

You can enable a stream on a new table when you create it, enable or disable a stream on an existing table, or change the settings of a stream. DynamoDB Streams operates asynchronously, so there is no performance impact on a table if you enable a stream.

You can also use the `CreateTable` or `UpdateTable` APIs to enable or modify a stream. The `StreamSpecification` parameter determines how the stream is configured:

**StreamEnabled** Specifies whether a stream for the table is enabled (true) or disabled (false)

**StreamViewType** Specifies the information that will be written to the stream whenever data in the table is modified:

**KEYS\_ONLY** Only the key attributes of the modified item

**NEW\_IMAGE** The entire item as it appears after it was modified

**OLD\_IMAGE** The entire item as it appeared before it was modified

**NEW\_AND\_OLD\_IMAGES** Both the new and the old images of the item

You can enable or disable a stream at any time. However, if you attempt to enable a stream on a table that already has a stream, you will receive a `ResourceInUseException`. If you attempt to disable a stream on a table that does not have a stream, you will receive a `ValidationException`.

When you set `StreamEnabled` to `true`, DynamoDB creates a new stream with a unique stream descriptor. If you disable and then re-enable a stream on the table, a new stream is created with a different stream descriptor.

The *Amazon Resource Name* (ARN) uniquely identifies every stream. The following is an example of defining an ARN for a stream on a DynamoDB table named `TestTable`:

```
arn:aws:dynamodb:us-west-2:111122223333:table/TestTable/stream/  
2015-05-11T21:21:33.291
```

To determine the latest stream descriptor for a table, issue a DynamoDB `DescribeTable` request and look for the `LatestStreamArn` element in the response.

## Reading and Processing a Stream

To read and process a stream, your application must connect to a DynamoDB Streams endpoint and issue API requests. A stream consists of *stream records*. Each stream record represents a single data modification in the DynamoDB table to which the stream belongs. Each stream record is assigned a sequence number, reflecting the order in which the record was published to the stream.

Stream records are organized into groups called *shards*. Each shard acts as a container for multiple stream records and contains information required for accessing and iterating through these records. The stream records within a shard are removed automatically after 24 hours. If you disable a stream, any shards that are open are closed.

Shards are ephemeral, meaning that they can be both created and deleted automatically as necessary. Any shard can automatically split into multiple new shards, and a shard may split in response to high levels of write activity on its parent table to enable applications to process records from multiple shards in parallel.

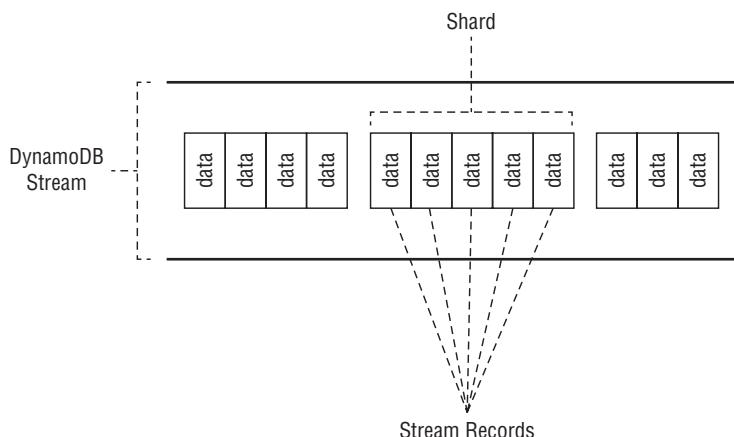
It is equally possible for a parent shard to have only one child shard. Because shards have a parent-and-children lineage, an application must always process a parent shard before it processes a child shard. This ensures that the stream records process in the correct order.

If you use the DynamoDB Streams *Kinesis Adapter*, this processing is handled for you. Your application processes the shards and stream records in the correct order, and it

automatically handles new or expired shards and shards that split while the application is running.

Figure 14.6 shows the relationship between a stream, shards in the stream, and stream records in the shards.

**FIGURE 14.6** Stream and shard relationship



To access a stream and process the stream records, do the following:

1. Identify the unique ARN of the stream that you want to access.
2. Determine which shard or shards in the stream contain the stream records of interest.
3. Access the shard or shards and retrieve the stream records that you want.



If you perform a PutItem or UpdateItem operation that does not change any data in an item, then DynamoDB Streams will not write a stream record for that operation.



No more than two processes should be reading from the same stream's shard at the same time. Having more than two readers per shard may result in throttling.

## Read Capacity for DynamoDB Streams

DynamoDB is available in multiple AWS Regions around the world. Each region is independent and isolated from other AWS Regions. For example, a table called *People* in the us-east-2 Region and a table named *People* in the us-west-2 Region are two entirely separate tables.

Every AWS Region consists of multiple, distinct locations called *Availability Zones*. Each Availability Zone is isolated from failures in other Availability Zones and provides inexpensive, low-latency network connectivity to other Availability Zones in the same region. This allows rapid replication of your data among multiple Availability Zones in a region.

When an application writes data to a DynamoDB table and receives an HTTP 200 response (OK), all copies of the data are updated. The data is eventually consistent across all storage locations, usually within one second or less.

**Eventually consistent reads** DynamoDB supports eventually consistent reads and strongly consistent reads. When you read data from a DynamoDB table, the response may not reflect the results of a recently completed write operation and may include some stale data. If you repeat your read request after a short period, the response returns the latest data.

**Strongly consistent reads** DynamoDB uses eventually consistent reads unless you specify otherwise. Read operations, such as GetItem, query, and Scan, provide a ConsistentRead parameter. If you set this parameter to true, DynamoDB uses strongly consistent reads during the operation. When you request a strongly consistent read, DynamoDB returns a response with the most up-to-date data, reflecting the updates from all prior write operations that were successful. A strongly consistent read may not be available if there is a network delay or outage.

## DynamoDB Streams API

The DynamoDB Streams API provides the following operations:

**ListStreams** Returns a list of stream descriptors for the current account and endpoint, or you can request only the stream descriptors for a particular table name.

**DescribeStream** Returns information about a stream, such as its ARN, and where your application can begin to read the first few stream records. The output includes a list of shards associated with the stream, including the shard IDs.

**GetShardIterator** Returns a shard iterator, which describes a location within a shard, to retrieve the records from the stream. You can request that the iterator provide access to the oldest point, the newest point, or a particular point in the stream.

**GetRecords** Retrieves one or more stream records by using a given shard iterator. Provides the shard iterator returned from a GetShardIterator request.

## Data Retention Limit for DynamoDB Streams

All data in DynamoDB Streams is subject to a 24-hour lifetime. You can retrieve and analyze the last 24 hours of activity for any given table; however, data older than 24 hours is susceptible to trimming (removal) at any moment.

If you disable a stream on a table, the data in the stream continues to be readable for 24 hours. After this time, the data expires, and the stream records are deleted automatically. There is no mechanism for manually deleting an existing stream; you must wait until the retention limit expires (24 hours) and all of the stream records are deleted.

## AWS Lambda Triggers in DynamoDB Streams

DynamoDB integrates with AWS Lambda, so you can create *triggers* (code that executes automatically) that automatically respond to events in DynamoDB Streams. With triggers, you can build applications that react to data modifications in DynamoDB tables.

### Example 3: DynamoDB Table Update Using AWS Lambda and Amazon Resource Name

In Figure 14.2, you have a mobile gaming app that writes to a *GameScores* table. Whenever the *TopScore* attribute of the *GameScores* table updates, a corresponding stream record writes to the table's stream. This event triggers a Lambda function that posts a congratulatory message on a social media network.

If you enable DynamoDB Streams on a table, you can associate the stream Amazon ARN with a Lambda function that you write. Immediately after an item in the table is modified, a new record appears in the table's stream. Lambda polls the stream and invokes your Lambda function synchronously when it detects new stream records.

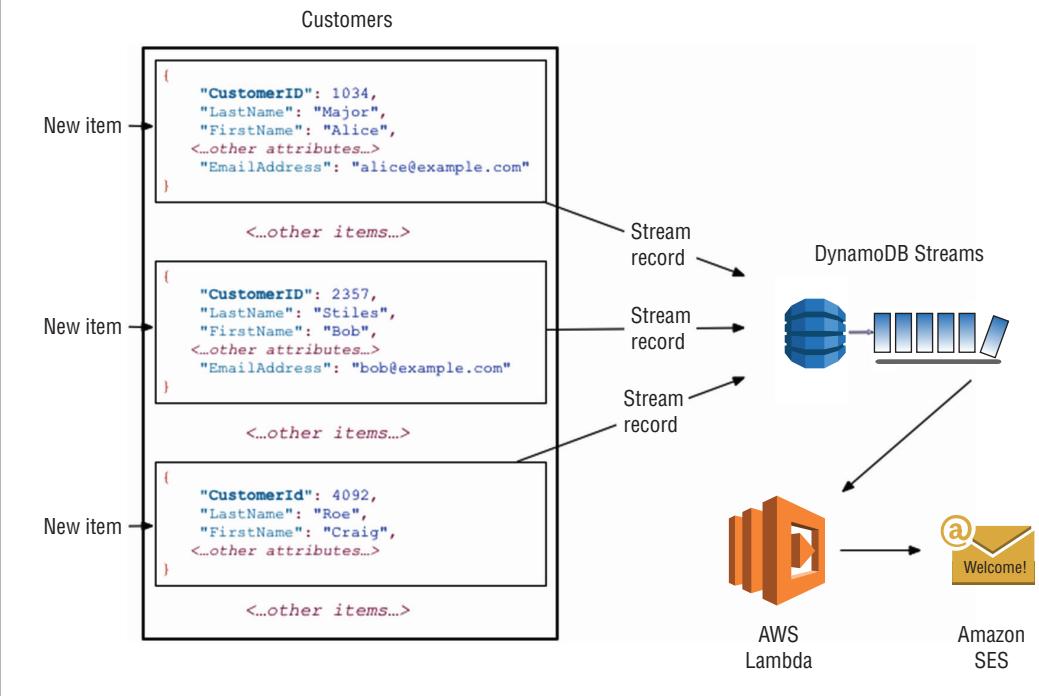
The Lambda function can perform any actions that you configure, such as sending a notification or initiating a workflow. For instance, you can write a Lambda function to copy each stream record to persistent storage, such as Amazon Simple Storage Service (Amazon S3), to create a permanent audit trail of write activity in your table.

Query so you can create *triggers* that automatically respond to events in DynamoDB Streams. With triggers, you can build applications that react to data modifications in DynamoDB tables.

If you enable DynamoDB Streams on a table, you can associate the stream ARN with a Lambda function that you write. Immediately after an item in the table is modified, a new record appears in the table's stream. Lambda polls the stream and invokes your Lambda function synchronously when it detects new stream records.

### Example 4: Lambda Email Trigger

A *Customers* table, such as the one shown in Figure 14.7, contains customer information for a company. If you want to send a “welcome” email to each new customer, enable a stream on that table and then associate the stream with a Lambda function. The Lambda function executes whenever a new stream record appears, but it processes only new items added to the Customers table. For any item that has an *EmailAddress* attribute, the Lambda function invokes *Amazon Simple Email Service* (Amazon SES) to send an email to that address. In Figure 14.7, the last customer, Craig Roe, will not receive an email because he does not have an *EmailAddress*.

**FIGURE 14.7** AWS Lambda Customers table

## Amazon DynamoDB Auto Scaling

*Amazon DynamoDB automatic scaling* actively manages throughput capacity for tables and global secondary indexes. With automatic scaling, you can define a range (upper and lower limits) for read and write capacity units and define a target utilization percentage within that range. DynamoDB automatic scaling seeks to maintain your target utilization, even as your application workload increases or decreases.

With DynamoDB automatic scaling, a table or a global secondary index can increase its provisioned read and write capacity to handle sudden increases in traffic without throttling. When the workload decreases, DynamoDB automatic scaling can decrease the throughput so that you do not pay for unused provisioned capacity.

If you use the AWS Management Console to create a table or a global secondary index, DynamoDB automatic scaling is enabled by default. You can manage automatic scaling settings at any time by using the console, the AWS CLI, or one of the AWS SDKs.

## Managing Throughput Capacity Automatically with AWS Auto Scaling

Many database workloads are cyclical in nature or are difficult to predict in advance. In a social networking application, where most of the users are active during daytime hours, the database must be able to handle the daytime activity. But there is no need for the same levels of throughput at night. If a new mobile gaming app is experiencing rapid adoption and becomes too popular, the app could exceed the available database resources, resulting in slow performance and unhappy customers. These situations often require manual intervention to scale database resources up or down in response to varying usage levels.

DynamoDB uses the *AWS Application Auto Scaling* service to adjust provisioned throughput capacity dynamically in response to actual traffic patterns. This enables a table or a global secondary index to increase its provisioned read and write capacity to handle sudden increases in traffic, without throttling. When the workload decreases, Application Auto Scaling decreases the throughput so that you do not pay for unused provisioned capacity.

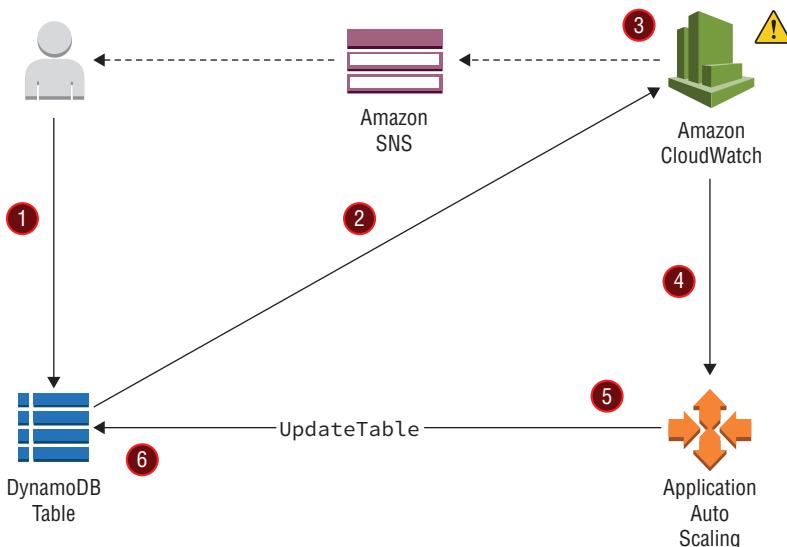
Application Auto Scaling does not scale down your provisioned capacity if the consumed capacity of your table becomes zero. To scale down capacity manually, perform one of the following actions:

- Send requests to the table until automatic scaling scales down to the minimum capacity.
- Change the policy and reduce the maximum provisioned capacity to the same size as the minimum provisioned capacity.

With Application Auto Scaling, you can create a scaling policy for a table or a global secondary index. The scaling policy specifies whether you want to scale read capacity or write capacity (or both), and the minimum and maximum provisioned capacity unit settings for the table or index.

The scaling policy also contains a target utilization that is the percentage of consumed provisioned throughput at a point in time. Application Auto Scaling uses a target tracking algorithm to adjust the provisioned throughput of the table (or index) upward or downward in response to actual workloads so that the actual capacity utilization remains at or near your target utilization.

DynamoDB automatic scaling also supports global secondary indexes. Every global secondary index has its own provisioned throughput capacity, separate from that of its base table. When you create a scaling policy for a global secondary index, Application Auto Scaling adjusts the provisioned throughput settings for the index to ensure that its actual utilization stays at or near your desired utilization ratio, as shown in Figure 14.8.

**FIGURE 14.8** DynamoDB Auto Scaling

## How DynamoDB Auto Scaling Works

The steps in Figure 14.8 summarize the automatic scaling process:

1. Create an Application Auto Scaling policy for your DynamoDB table.
2. DynamoDB publishes consumed capacity metrics to Amazon CloudWatch.
3. If the table's consumed capacity exceeds your target utilization (or falls below the target) for a specific length of time, CloudWatch triggers an alarm. You can view the alarm on the AWS Management Console and receive notifications using Amazon Simple Notification Service (Amazon SNS).
4. The CloudWatch alarm invokes Application Auto Scaling to evaluate your scaling policy.
5. Application Auto Scaling issues an `UpdateTable` request to adjust your table's provisioned throughput.
6. DynamoDB processes the `UpdateTable` request, increasing or decreasing the table's provisioned throughput capacity dynamically so that it approaches your target utilization.

DynamoDB automatic scaling modifies provisioned throughput settings only when the actual workload stays elevated or depressed for a sustained period of several minutes. The

Application Auto Scaling target tracking algorithm seeks to keep the target utilization at or near your chosen value over the long term. Sudden, short-duration spikes of activity are accommodated by the table’s built-in burst capacity.

## Burst Capacity

DynamoDB provides some flexibility in your per-partition throughput provisioning by providing *burst capacity*. Whenever you are not fully using a partition’s throughput, Amazon DynamoDB reserves a portion of that unused capacity for later bursts of throughput to handle usage spikes.

DynamoDB currently retains up to 5 minutes (300 seconds) of unused read and write capacity. During an occasional burst of read or write activity, these extra capacity units can be consumed quickly—even faster than the per-second provisioned throughput capacity that you have defined for your table. However, do not rely on burst capacity being available at all times, as DynamoDB can also consume burst capacity for background maintenance and other tasks without prior notice.

To enable DynamoDB automatic scaling, you create a scaling policy. This *scaling policy* specifies the table or global secondary index that you want to manage, which capacity type to manage (read or write capacity), the upper and lower boundaries for the provisioned throughput settings, and your target utilization.

When you create a scaling policy, Application Auto Scaling creates a pair of CloudWatch alarms on your behalf. Each pair represents your upper and lower boundaries for provisioned throughput settings. These CloudWatch alarms are triggered when the table’s actual utilization deviates from your target utilization for a sustained period of time.

When one of the CloudWatch alarms is triggered, Amazon SNS sends you a notification (if you have enabled it). The CloudWatch alarm then invokes Application Auto Scaling, which notifies DynamoDB to adjust the table’s provisioned capacity upward or downward, as appropriate.

## Considerations for DynamoDB Auto Scaling

Before you begin using DynamoDB automatic scaling, be aware of the following:

- DynamoDB automatic scaling can increase read capacity or write capacity as often as necessary in accordance with your automatic scaling policy. All DynamoDB limits remain in effect.
- DynamoDB automatic scaling does not prevent you from manually modifying provisioned throughput settings. These manual adjustments do not affect any existing CloudWatch alarms that are related to DynamoDB automatic scaling.
- If you enable DynamoDB automatic scaling for a table that has one or more global secondary indexes, AWS highly recommends that you also apply automatic scaling uniformly to those indexes. You can apply this by choosing *Apply same settings* to global secondary indexes in the AWS Management Console.

## Provisioned Throughput for DynamoDB Auto Scaling

If you are not using DynamoDB automatic scaling, you must manually define your throughput requirements. *Provisioned throughput* is the maximum amount of capacity that an application can consume from a table or index. If your application exceeds your provisioned throughput settings, it is subject to request throttling.

### Example 5: Determining the Provisioned Throughput Setting

Suppose that you want to read 80 items per second from a table, where the items are 3 KB in size, and you want strongly consistent reads with each read requiring one provisioned read capacity unit. To determine this, divide the item size of the operation by 4 KB and then round up to the nearest whole number:

$$3 \text{ KB}/4 \text{ KB} = 0.75, \text{ or } 1 \text{ read capacity unit}$$

Knowing this, you must set the table's provisioned read throughput to 80 read capacity units:

$$1 \text{ read capacity unit per item} \times 80 \text{ reads per second} = 80 \text{ read capacity units}$$

If you want to write 100 items per second to your table, and the items are 512 bytes in size, each write requires one provisioned write capacity unit. To determine this, divide the item size of the operation by 1 KB and then round up to the nearest whole number:

$$512 \text{ bytes}/1 \text{ KB} = 0.5, \text{ or } 1$$

To accomplish this, set the table's provisioned write throughput to 100 write capacity units:

$$1 \text{ write capacity unit per item} \times 100 \text{ writes per second} = 100 \text{ write capacity units}$$

## Partitions and Data Distribution

DynamoDB stores data in partitions. A *partition* is an allocation of storage for a table, backed by solid-state drives (SSDs) and automatically replicated across multiple Availability Zones within an AWS Region. Partition management is handled entirely by DynamoDB, so you do not have to manage partitions yourself. When you create a table, the initial status of the table is CREATING. During this phase, DynamoDB allocates sufficient partitions to the table so that it can handle your provisioned throughput requirements. You can begin writing and reading table data after the table status changes to ACTIVE.

DynamoDB allocates additional partitions to a table in the following situations:

- If you increase the table's provisioned throughput settings beyond what the existing partitions can support
- If an existing partition fills to capacity and more storage space is required

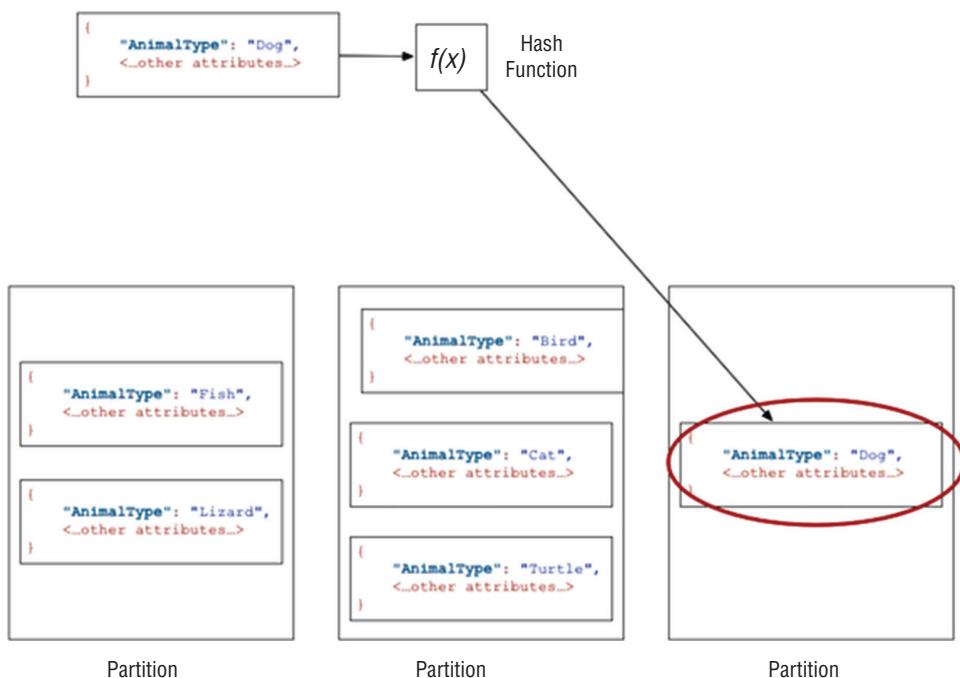
Partition management occurs automatically in the background, and it is transparent to your applications. Your table remains available throughout and fully supports your provisioned throughput requirements. Global secondary indexes in DynamoDB are also composed of partitions. The data in a global secondary index is stored separately from the data in its base table, but index partitions behave similarly to table partitions.

## Data Distribution: Partition Key

If your table has a simple primary key (partition key only), DynamoDB stores and retrieves each item based on its partition key value. To write an item to the table, DynamoDB uses the value of the partition key as input to an internal hash function. The output value from the hash function determines the partition in which the item will be stored. To read an item from the table, you must configure the partition key value for the item. DynamoDB uses this value as input to its hash function, yielding the partition in which the item can be found.

Figure 14.9 shows a table named *Pets*, which spans multiple partitions. The table's primary key is *AnimalType* (only this key attribute is shown). DynamoDB uses its hash function to determine where to store a new item, in this case based on the hash value of the string *Dog*. The items are not stored in sorted order. Each item's location is determined by the hash value of its partition key.

**FIGURE 14.9** Data distribution and partition





DynamoDB is optimized for uniform distribution of items across a table's partitions, regardless of the number of partitions. Choose a partition key with a large number of distinct values relative to the number of items in the table.

## Data Distribution: Partition Key and Sort Key

If the table has a composite primary key (partition key and sort key), DynamoDB calculates the hash value of the partition key in the same way, but it stores the items with the same partition key value physically close together, ordered by sort key value.

To write an item to the table, DynamoDB calculates the hash value of the partition key to determine which partition should contain the item. In that partition, there could be several items with the same partition key value, so DynamoDB stores the item among the others with the same partition key in ascending order by sort key.

To read an item from the table, configure both the partition key value and sort key value. DynamoDB calculates the partition key's hash value, yielding the partition in which the item can be found.

You can read multiple items from the table in a single Query operation, if the desired items have the same partition key value. DynamoDB returns all items with that partition key value. You can apply a condition to the sort key that only returns items within a certain range of values.

## Optimistic Locking with Version Number

*Optimistic locking* is a strategy to ensure that the client-side item that you are updating or deleting is the same as the item in DynamoDB. If you use this strategy, then all writes on your database are protected from being accidentally overwritten.



DynamoDB global tables use a "last writer wins" reconciliation between concurrent updates. If you use Global Tables, last writer policy wins. In this case, the locking strategy does not work as expected.

With optimistic locking, each item has an attribute that acts as a version number. If you retrieve an item from a table, the application records the version number of that item. You can update the item, but only if the version number on the server side has not changed. If there is a version mismatch, then someone else has modified the item before you did, and the update attempt fails because you have an outdated version of the item. If this happens, you retrieve the current item and then attempt to update it again.

To support optimistic locking, the AWS SDK for Java provides the `@AmazonDynamoDBVersionAttribute` annotation. In the mapping class for your table, designate one property to store the version number and mark it using the annotation. When you save an object, the corresponding item in the DynamoDB table has an attribute that stores the version number. The Amazon DynamoDBMapper assigns a version number when you first save the object, and it automatically increments the version number each time you update the

item. Your update or delete requests succeed only if the client-side object version matches the corresponding version number of the item in the Amazon DynamoDB table.

`ConditionalCheckFailedException` occurs if the following conditions are true:

- You use optimistic locking with `@AmazonDynamoDBVersionAttribute`, and the version value on the server is different from the value on the client side.
- You configure your own conditional constraints while saving data by using `AmazonDynamoDBMapper` with `AmazonDynamoDBSaveExpression`, and these constraints failed.

## Disabling Optimistic Locking

To disable optimistic locking, change the `AmazonDynamoDBMapperConfig.SaveBehavior` enumeration value from `UPDATE` to `CLOBBER`. Do this by creating an `AmazonDynamoDBMapperConfig` instance that skips version checking and then use this instance for your requests. You can also set locking behavior for a specific operation only. For example, the following Java snippet uses the `DynamoDBMapper` to save a catalog item. It specifies `DynamoDBMapperConfig.SaveBehavior` by adding the optional `DynamoDBMapperConfig` parameter to the `save` method.

```
DynamoDBMapper mapper = new DynamoDBMapper(client);

// Load a catalog item.
CatalogItem item = mapper.load(CatalogItem.class, 101);
item.setTitle("This is a new title for the item");
...
// Save the item.
mapper.save(item,
    new DynamoDBMapperConfig(
        DynamoDBMapperConfig.SaveBehavior.CLOBBER));
```

## DynamoDB Tags

You can label DynamoDB resources with tags. *Tags* allow you to categorize your resources in different ways: by purpose, owner, environment, or other criteria. Tags help you to identify a resource quickly based on the tags that you have assigned to it, and they help you to see your AWS bills broken down by tags.

Tables that have tags automatically tag local secondary indexes and global secondary indexes. Currently, you cannot tag DynamoDB Streams. AWS offerings and services, such as Amazon EC2, Amazon S3, DynamoDB, and more, support tags. Efficient tagging can provide cost insights by enabling you to create reports across services that carry a specific tag.

## Tag Restrictions

*Each tag consists of a key and a value*, both of which you define. Each DynamoDB table can have only one tag with the same key, so if you attempt to add an existing tag (the same key), the existing tag value updates to the new value.

The following restrictions apply:

- Tag keys and values are case-sensitive.
- The maximum key length is 128 Unicode characters, and the maximum value length is 256 Unicode characters. The allowed character types are letters, white space, and numbers, plus the following special characters: + - = . \_ : /.
- The maximum number of tags per resource is 50.
- The AWS-assigned tag names and values are automatically assigned the aws: prefix, which you cannot manually assign.
- AWS-assigned tag names do not count toward the tag limit of 50.
- User-assigned tag names have the prefix user: in the cost allocation report.
- You cannot tag a resource at the same time that you create it.



Tagging is a separate action that you can perform only after you create the resource. You cannot backdate the application of a tag.

## DynamoDB Items

A DynamoDB *item* is a collection of attributes that is uniquely identifiable among all other entities in the table, and each item has a *name* and a *value*. An attribute value can be a scalar, a set, or a document type. Each table contains zero or more items. For example, in a *People* table, each item represents a person, and in a “cars” table, each item represents one vehicle. Items in DynamoDB are similar to rows, records, or tables in other database systems, but in DynamoDB, there is no limit to the number of items that you can store in a table.

## Atomic Counters

You can use the `UpdateItem` operation to implement an *atomic counter*, which is a numeric attribute that increments, unconditionally, without interfering with other write requests. With an atomic counter, the updates are not independent, and the numeric value increments each time that you call `UpdateItem`.

You can use an atomic counter to track the number of visitors to a website. In this case, your application would increment a numeric value, regardless of its current value. If an `UpdateItem` operation fails, the application may retry the operation. This would risk updating the counter twice, but most can tolerate a slight overcounting or undercounting of website visitors.

An atomic counter would not be appropriate where overcounting or undercounting cannot be tolerated, as in a banking application. In this case, it is safer to use a conditional update instead of an atomic counter.



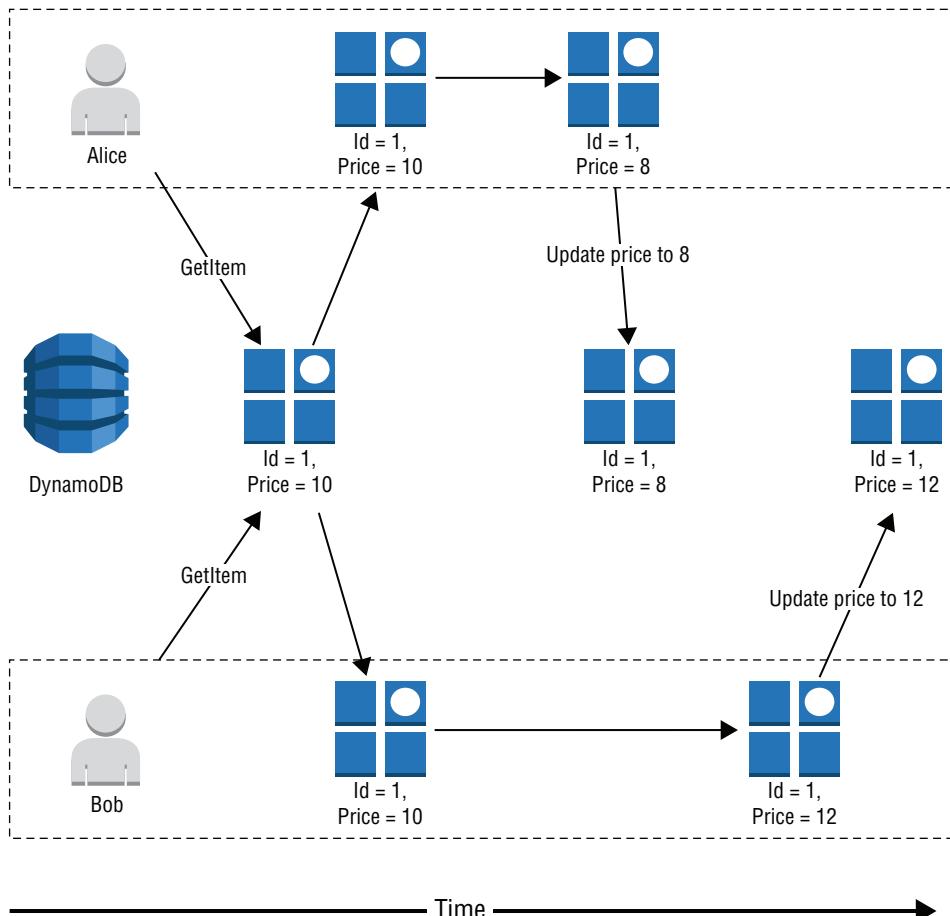
All write requests are applied in the order in which they were received.

## Conditional Writes

By default, the DynamoDB write operations (`PutItem`, `UpdateItem`, `DeleteItem`) are unconditional. Each of these operations overwrites an existing item that has the specified primary key. DynamoDB supports *conditional writes* for these operations. A conditional write succeeds only if the item attributes meet one or more expected conditions; otherwise, it returns an error.

Conditional writes are helpful in many situations, including cases in which multiple users attempt to modify the same item. You may want a `PutItem` operation to succeed only if there is not already an item with the same primary key. Alternatively, you could prevent an `UpdateItem` operation from modifying an item if one of its attributes has a certain value. Consider Figure 14.10 in which two users (Alice and Bob) are working with the same item from a DynamoDB table.

**FIGURE 14.10** Conditional write success



Suppose that Alice updates the Price attribute to 8.

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"1"}}' \
--update-expression "SET Price = :newval" \
--expression-attribute-values file://expression-attribute-values.json
```

The arguments for --expression-attribute-values write to the file expression-attribute-values.json.

```
{
  ":newval":{"N":"8"}
}
```

Now suppose that Bob issues a similar UpdateItem request later but changes the Price to 12. For Bob, the --expression-attribute-values parameter looks like this:

```
{
  ":newval":{"N":"12"}
}
```

Bob's request succeeds, but Alice's earlier update is lost.

To request a conditional PutItem, DeleteItem, or UpdateItem, you configure a condition expression. A *condition expression* is a string containing attribute names, conditional operators, and built-in functions where the entire expression must evaluate to true; otherwise, the operation fails.

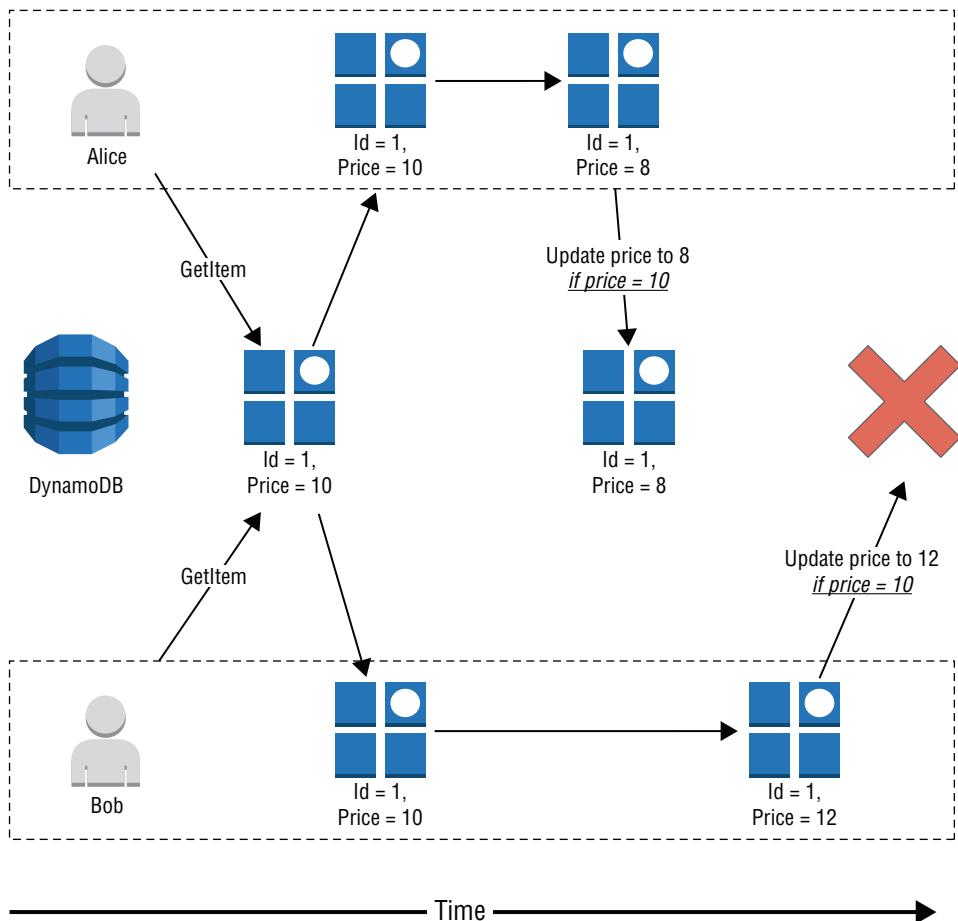
Now consider Figure 14.11, showing how conditional writes would prevent Alice's update from being overwritten.

Alice first attempts to update Price to 8 but only if the current Price is 10.

```
aws dynamodb update-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"1"}}' \
--update-expression "SET Price = :newval" \
--condition-expression "Price = :currval" \
--expression-attribute-values file://expression-attribute-values.json
```

The arguments for --expression-attribute-values write to the file expression-attribute-values.json.

```
{
  ":newval":{"N":"8"}, 
  ":currval":{"N":"10"}
}
```

**FIGURE 14.11** Conditional write success

Alice's update succeeds because the condition evaluates to true.

Next, Bob attempts to update the Price to 12 but only if the current Price is 10. For Bob, the --expression-attribute-values parameter looks like the following:

```
{
  ":newval": {"N": "12"},  

  ":currval": {"N": "10"}  

}
```

Because Alice has previously changed the Price to 8, the condition expression evaluates to false and Bob's update fails.

## Time to Live

*Time to Live* (TTL) for DynamoDB enables you to define when items in a table expire so that they can be automatically deleted from the database.

AWS provides TTL at no extra cost to you as a way to reduce both storage usage and the cost of storing irrelevant data without using provisioned throughput. With TTL enabled on a table, you can set a timestamp for deletion on a per-item basis and limit storage usage to only those records that are relevant.

TTL is useful if you have continuously accumulating data that loses relevance after a specific time, such as session data, event logs, usage patterns, and other temporary data. If you have sensitive data that must be retained only for a certain amount of time according to contractual or regulatory obligations, TTL helps you to make sure that data is removed promptly and on schedule.

### Enabling Time to Live

When you enable TTL on a table, a background job checks the TTL attribute of items to determine whether they are expired. TTL compares the current time in epoch time format to the time stored in the `Time to Live` attribute of an item. If the epoch time value stored in the attribute is less than the current time, the item is marked as expired and later deleted.



The epoch time format is the number of seconds elapsed since 12:00:00 a.m. on January 1, 1970, UTC.

DynamoDB deletes expired items on a best-effort basis to ensure availability of throughput for other data operations. DynamoDB typically deletes expired items within 48 hours of expiration. The exact duration within which an item is deleted after expiration is specific to the nature of the workload and the size of the table.

Items that have expired but not deleted still show up in reads, queries, and scans. These items can be updated, and successful updates to change or remove the expiration attribute will be honored. As items are deleted, they are immediately removed from local secondary and global secondary indexes in the same eventually consistent way as a standard delete operation.

### Before Using Time to Live

Before you enable TTL on a table, consider the following:

- Make sure that any existing timestamp values in the specified `Time to Live` attribute are correct and in the right format.
- Items with an expiration time greater than five years in the past are not deleted.
- If data recovery is a concern, back up your table.
- For a 24-hour recovery window, use DynamoDB Streams.
- For a full backup, use AWS Data Pipeline.

- Use the AWS CloudFormation to set TTL when you create a DynamoDB table.
- Use Identity and Access Management (IAM) policies to prevent unauthorized updates to the TTL attribute or configuration of the TTL feature. If you allow access to only specified actions in your existing IAM policies, ensure that your policies update to allow DynamoDB:UpdateTimeToLive for roles that need to enable or disable TTL on tables.
- Consider whether you must complete any post-processing of deleted items. The stream's records of TTL deletes are marked, and you use AWS Lambda function to monitor the records.

When your program sends a request, DynamoDB attempts to process it. If the request is successful, DynamoDB returns an HTTP success status code (200 OK), along with the results from the requested operation.

If the request is unsuccessful, DynamoDB returns an error. Each error has three components:

- An HTTP status code (such as 400)
- An exception name (such as ResourceNameNotFound)
- An error message (such as Requested resource not found: Table: tablename not found)

The AWS SDKs resolve propagating errors in your application so that you can take appropriate action. For example, in a Java program, write try-catch logic to handle a ResourceNotFoundException.

## Error Handling in Your Application

For your application to run smoothly, you must add logic to catch and respond to errors. Typical approaches include using try-catch blocks or if-then statements. The AWS SDKs perform their own retries and error checking. If you encounter an error while using one of the AWS SDKs, the error code and description help you troubleshoot it. You may also see a Request ID in the response, which can be helpful when working with AWS Support to diagnose an issue.

### Error Retries and Exponential Backoff

Numerous components on a network, such as DNS servers, switches, load balancers, and others, can return error responses anywhere in the life of a given request. The usual technique for dealing with these error responses in a networked environment is to implement retries in the client application. This technique increases the reliability of the application and reduces operational costs for the developer.

As each AWS SDK automatically implements retry logic, you can modify the retry parameters to suit your needs. For example, consider a Java application that requires a fail-fast strategy with no retries allowed in case of an error. With the AWS SDK for Java, you

could use the `ClientConfiguration` class and provide a `maxErrorRetry` value of 0 to turn off the retries.

If you are not using an AWS SDK, attempt to retry original requests that receive server errors (5xx). However, client errors (4xx), other than a `ThrottlingException` or a `ProvisionedThroughputExceededException`, indicate the need to revise the request itself or to correct the problem before trying again.

In addition to simple retries, each AWS SDK implements the *exponential backoff algorithm* for better flow control. The concept behind exponential backoff is to use progressively longer waits between retries for consecutive error responses. For example, you can set the wait to up to 50 milliseconds before the first retry, up to 100 milliseconds before the second retry, up to 200 milliseconds before third retry, and so on.

However, if the request has not succeeded after a minute, the request size may exceed your provisioned throughput and not the request rate. Set the maximum number of retries to stop at around one minute. If the request is not successful, investigate your provisioned throughput options.

## Capacity Units Consumed by Conditional Writes

If a `ConditionExpression` generates an evaluation of `false` during a conditional write, DynamoDB consumes write capacity from the table. If the item does not currently exist in the table, DynamoDB consumes one write capacity unit. If the item does exist, then the number of write capacity units consumed depends on the size of the item. A failed conditional write of a 1-KB item would consume one write capacity unit. If the item were twice that size, the failed conditional write would consume two write capacity units.

A failed conditional write returns a `ConditionalCheckFailedException`. When this occurs, you will not receive information in the response about the write capacity that was consumed. However, you can view the `ConsumedWriteCapacityUnits` metric for the table in Amazon CloudWatch.

To return the number of write capacity units consumed during a conditional write, you use the `ReturnConsumedCapacity` parameter with the following attributes:

`Total` Returns the total number of write capacity units consumed.

`Indexes` Returns the total number of write capacity units consumed with subtotals for the table and any secondary indexes that were affected by the operation.

`None` No write capacity details are returned (default).



Write operations consume only write capacity units; they do not consume read capacity units.

Unlike a global secondary index, a local secondary index shares its provisioned throughput capacity with its table. Read and write activity on a local secondary index consumes provisioned throughput capacity from the table.

## Configuring Item Attributes

This section describes how to refer to item attributes in an expression and projection expression. You can work with any attribute, even if it is deeply nested within multiple lists and maps.

### Item Attributes

You can work with any attribute in an expression, even if it is deeply nested within multiple lists and maps.

### Top-Level Attributes

If an attribute is not embedded within another attribute, the attribute is top level. Top-level attributes include the following:

- Id
- Title
- Description
- BicycleType
- Brand
- Price
- Color
- ProductCategory
- InStock
- QuantityOnHand
- RelatedItems
- Pictures
- ProductReviews
- Comment
- Safety.Warning

All of the top-level attributes are scalars, except for Color (list), RelatedItems (list), Pictures (map), and ProductReviews (map).

### Nested Attributes

A nested attribute is embedded within another attribute. To access a nested attribute, you use *dereference operators*:

- [n] for list elements
- .(dot) for map elements

## Accessing List Elements

The dereference operator for a list element is `[n]`, where `n` is the element number. List elements are zero-based, so `[0]` represents the first element in the list, `[1]` represents the second, and so on. For example:

- `MyList[0]`
- `AnotherList[12]`
- `ThisList[5][11]`

The element `ThisList[5]` is itself a nested list. Therefore, `ThisList[5][11]` refers to the twelfth element in that list.

The number within the square brackets must be a non-negative integer. Therefore, the following expressions are invalid:

- `MyList[-1]`
- `MyList[0.4]`

## Accessing Map Elements

The dereference operator for a map element is a dot (`.`). Use a dot as a separator between elements in a map. For example:

- `MyMap.nestedField`
- `MyMap.nestedField.deeplyNestedField`

## Document Paths

In an expression, you use a document path to tell DynamoDB where to find an attribute. For a top-level attribute, the document path is the attribute name. For a nested attribute, you construct the document path by using dereference operators.

The following are examples of document paths:

- Top-level *scalar* attribute: `ProductDescription`.
- Top-level *list* attribute returns the entire list, not only some of the elements: `RelatedItems`.
- Third element from the `RelatedItems` list (remember that list elements are zero-based): `RelatedItems[2]`.
- Front-view *picture* of the product: `Pictures.FrontView`.
- All of the five-star reviews: `ProductReviews.FiveStar`.
- First of the five-star reviews: `ProductReviews.FiveStar[0]`.

You can use any attribute name in a document path if the first character is `a-z` or `A-Z` and the second character (if present) is `a-z`, `A-Z`, or `0-9`. If an attribute name does not meet this requirement, define an expression attribute name as a placeholder.



The maximum depth for a document path is 32. Therefore, the number of dereference operators in a path cannot exceed this limit.

## Expressions

In DynamoDB, you can use *expressions* to denote the attributes that you want to read from an item. To indicate any conditions that must be met (conditional update) and to indicate how the attributes are to be updated, you can also use expressions when writing an item.



For backward-compatibility, DynamoDB also supports conditional parameters that do not use expressions. New applications should use expressions instead of the legacy parameters.

## Item Projection Expressions

To read data from a table, use operations such as `GetItem`, `Query`, or `Scan`. DynamoDB returns *all* of the item attributes by default. To acquire select attributes, use a projection expression.

A *projection expression* is a string that identifies the attributes that you want to collect. To retrieve a single attribute, specify its name. For multiple attributes, the names must be comma-separated.

The following are examples of projection expressions:

- Single top-level attribute:  
`Title`
- Three top-level attributes; DynamoDB retrieves the entire `Color` set:  
`Title, Price, Color`
- Four top-level attributes' DynamoDB returns the entire contents of `RelatedItems` and `ProductReviews`:  
`Title, Description, RelatedItems, ProductReviews`

### Attribute Names in a Projection Expression

You can use any attribute name in a projection expression, where the first character is a-z or A-Z and the second character (if present) is a-z, A-Z, or 0-9. If an attribute name does not meet this requirement, you must define an expression attribute name as a placeholder.

## Expression Attribute Names

An *expression attribute name* is a placeholder that you use in an expression as an alternative to an actual attribute name. An expression attribute name must begin with a # and be followed by one or more alphanumeric characters. There are several situations in which you use expression attribute names.

**Reserved words** On certain occasions, you might need to write an expression containing an attribute name that conflicts with a DynamoDB reserved word. Refer to <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ReservedWords.html>.

### Example 6: Reserved Words

```
aws dynamodb get-item \  
--table-name ProductCatalog \  
--key '{"Id":{"N":"123"}}' \  
--projection-expression "Comment"
```



If an attribute name begins with a number or contains a space, a special character, or a reserved word, then you *must* use an expression attribute name to replace that attribute's name in the expression.

**Attribute names containing dots** In an expression, a dot (.) is interpreted as a separator character in a document path. However, DynamoDB also enables you to use a dot character as part of an attribute name, which can be ambiguous. To illustrate, suppose that you want to retrieve the Safety.Warning attribute from a table.

To work around this, replace Comment with an expression attribute name, such as #c. The # (pound sign) is required, and it indicates that this is a placeholder for an attribute name. Suppose that you want to access Safety.Warning by using a projection-expression:

```
aws dynamodb get-item \  
--table-name ProductCatalog \  
--key '{"Id":{"N":"123"}}' \  
--projection-expression "#Safety.Warning"
```

DynamoDB returns an empty result, rather than the expected string ("Always wear a helmet") when DynamoDB interprets a dot in an expression as a document path separator. In this case, you must define an expression attribute names (such as #sw) as a substitute for Safety.Warning. Use the following projection-expression:

```
aws dynamodb get-item \  
--table-name ProductCatalog \  
--key '{"Id":{"N":"123"}}' \  
--projection-expression "#sw" \  
--expression-attribute-names '{"#sw":"Safety.Warning"}'
```

DynamoDB would then return the correct result.

**Nested attributes** Suppose that you want to access the nested attribute `ProductReviews.OneStar`, using the following projection-expression:

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"123"}}' \
--projection-expression "ProductReviews.OneStar"
```

The result contains all of the one-star product reviews, which is expected.

But what if you want to use a projection-expression attribute instead? For example, you want to define `#pr1star` as a substitute for `ProductReviews.OneStar`:

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"123"}}' \
--projection-expression "#pr1star" \
--expression-attribute-names '{"#pr1star":"ProductReviews.OneStar"}'
```

DynamoDB returns an empty result instead of the expected map of one-star reviews when DynamoDB interprets a dot in an expression attribute value as a character within an attribute's name. When DynamoDB evaluates the expression attribute name `#pr1star`, it determines that `ProductReviews.OneStar` refers to a *scalar attribute*, which is not what was intended.

The correct approach is to define an `expression-attribute-names` attribute for each element in the document path:

**#pr:** `ProductReviews`  
**#1star:** `OneStar`

You then use `#pr.#1star` for the projection expression:

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"123"}}' \
--projection-expression "#pr.#1star" \
--expression-attribute-names '{"#pr":"ProductReviews", "#1star":"OneStar"}'
```

DynamoDB returns the correct result.

**Repeat attribute names** Expression attribute names are helpful when you must refer to the same attribute name repeatedly. For example, consider the following expression for retrieving reviews from a `ProductCatalog` item:

```
aws dynamodb get-item \
--table-name ProductCatalog \
--key '{"Id":{"N":"123"}}' \
--projection-expression "ProductReviews.FiveStar, ProductReviews.ThreeStar, \
ProductReviews.OneStar"
```

To make this more concise, replace `ProductReviews` with an expression attribute name, such as `#pr`. The revised expression looks like the following:

```
aws dynamodb get-item \  
  --table-name ProductCatalog \  
  --key '{"Id":{"N":"123"}}' \  
  --projection-expression "#pr.FiveStar, #pr.ThreeStar, #pr.OneStar" \  
  --expression-attribute-names '{"#pr":"ProductReviews"}'
```

If you define an expression attribute name, you must use it consistently throughout the entire expression. Also, you cannot omit the `#` symbol.

## Expression Attribute Values

If you must compare an attribute with a value, define an expression attribute value as a placeholder. *Expression attribute values* are substitutes for the actual values that you want to compare. These are values that you might not know until runtime. Use expression attribute values with condition expressions, update expressions, and filter expressions. An expression attribute value must begin with a colon (`:`) followed by one or more alphanumeric characters.

For example, you want to return all of the `ProductCatalog` items that are available in black and cost \$500 or less. You could use a `Scan` operation with a `filter-expression`, as in this AWS CLI example:

```
aws dynamodb scan \  
  --table-name ProductCatalog \  
  --filter-expression "contains(Color, :c) and Price <= :p" \  
  --expression-attribute-values file://values.json
```

The arguments for `--expression-attribute-values` are stored in the file `values.json`:

```
{  
  ":c": { "S": "Black" },  
  ":p": { "N": "500" }  
}
```



Because a `Scan` operation reads every item in a table, avoid using `Scan` with large tables. The filter expression is applied to the `Scan` results, and items that do not match the filter expression are discarded.

If you define an `expression-attribute-values` attribute, you must use it consistently throughout the entire expression. Also, you cannot omit the colon (`:`) symbol.

## Condition Expressions

To manipulate data in a DynamoDB table, use the `PutItem`, `UpdateItem`, and `DeleteItem` operations. You can also use `BatchWriteItem` to perform multiple `PutItem` or `DeleteItem` operations in a single call.

For these data manipulation operations, configure a condition expression to determine which items to modify. If the condition expression evaluates to true, the operation succeeds; otherwise, the operation fails.

The following AWS CLI examples include condition expressions that use the ProductCatalog table. The partition key for this table is `Id`; there is no sort key. The `PutItem` operation creates a sample ProductCatalog item in the examples:

```
aws dynamodb put-item \
--table-name ProductCatalog \
--item file://item.json
```

The arguments for `--item` are stored in the file `item.json`.

```
{
  "Id": {"N": "456" },
  "ProductCategory": {"S": "Sporting Goods" },
  "Price": {"N": "650" }
}
```

## Update Expressions

To update an existing item in a table, use the `UpdateItem` operation, provide the key of the item that you want to update, and use an update expression, indicating the attributes that you want to modify and the values that you want to assign to them.

An *update expression* specifies how `UpdateItem` modifies the attributes of an item, such as setting a scalar value or removing elements from a list or a map. An update expression consists of one or more clauses. Each clause begins with a `SET`, `REMOVE`, `ADD`, or `DELETE` keyword. You can include any of these clauses in an update expression, in any order. *However, each action keyword can appear only once.* Each clause contains one or more actions, separated by commas.

Each of the following actions represents a data modification:

**SET** Updates the expression to add one or more attributes to an item. If any of these attributes already exists, it is overwritten by the new value.

**REMOVE** Updates the expression to remove one or more attributes from an item. To perform multiple Remove actions, separate the attributes by commas and use `Remove` to delete individual elements from a list.

**ADD** Updates the expression to add a new attribute and its values or values to an item. If the attribute already exists, then the behavior of `ADD` depends on the attribute's data type:

- If the attribute is a number and the value you are adding is also a number, then the value is mathematically added to the existing attribute. If the value is a negative number, then it is subtracted from the existing attribute.
- If the attribute is a set, and the value you are adding is also a set, then the value is appended to the existing set.

**DELETE** Deletes the expression.

### Example 7: Update Expression

```
update-expression ::=  
  [ SET action [, action] ... ]  
  [ REMOVE action [, action] ... ]  
  [ ADD action [, action] ... ]  
  [ DELETE action [, action] ... ]
```

## Working with Queries

The Query operation finds items based on primary key values. You can query any table or secondary index that has a composite primary key (a partition key and a sort key).

You must provide the name of the partition key attribute and a single value for that attribute. Query returns all the items with that partition key value. You can provide a sort key attribute and use a comparison operator to refine the search results.

### Key Condition Expression

To specify the search criteria, use a key condition expression. A *key condition expression* is a string that determines the items to be read from the table or index. You must configure the partition key name and value as an equality condition.

You can use any attribute name in a key condition expression as long as the first character is a–z or A–Z and the second character (if present) is a–z, A–Z, or 0–9.

In addition, the attribute name must not be a DynamoDB reserved word. If an attribute name does not meet these requirements, then define an expression attribute name as a placeholder.

For items with a given partition key value, DynamoDB stores these items close together, sorted by the sort key value. In an aws dynamodb Query operation, DynamoDB retrieves the items in sorted order and then processes the items using KeyConditionExpression and any FilterExpression that might be present. At that point, the aws dynamodb query results are sent to the client.

An aws dynamodb query operation generally returns a result set. If no matching items are found, the result set is empty. Query results sort by the sort key value. If the data type of the sort key is Number, the results return in numeric order; otherwise, the results are returned in the order of UTF-8 bytes. The sort order is ascending by default. If you want to reverse the order, set the ScanIndexForward parameter to false. A single Query operation can retrieve a maximum of 1 MB of data. This limit applies before any FilterExpression is applied to the results. If LastEvaluatedKey is present in the response and it is non-null, then paginate the result set.

**Example 8: Query Thread Table for *ForumName* (Partition Key)**

```
aws dynamodb query \
--table-name Thread \
--key-condition-expression "ForumName = :name" \
--expression-attribute-values '{":name":{"S":"Amazon DynamoDB"}}'
```

**Filter Expressions for *Query***

You can use a filter expression to refine the *Query* results further. A *filter expression* determines which items within the `aws dynamodb query` results return. All other results are discarded. A *filter expression* is applied after an `aws dynamodb query` finishes, but before the results are returned. Therefore, a query consumes the same amount of read capacity, regardless of whether a filter expression is used. An `aws dynamodb query` operation can retrieve a maximum of 1 MB of data. This limit applies before the filter expression is evaluated. A filter expression cannot contain partition key or sort key attributes. Configure those attributes in the key condition expression, not the filter expression.

**Example 9: Query the Thread Table for Partition Key and Sort Key**

```
aws dynamodb query \
--table-name Thread \
--key-condition-expression "ForumName = :fn" \
--filter-expression "#v >= :num" \
--expression-attribute-names '{"#v": "Views"}' \
--expression-attribute-values file://values.json
```

**Read Consistency for *Query***

A *Query* operation performs eventually consistent reads by default. This means that the *Query* results might not reflect changes as the result of recently completed `PutItem` or `UpdateItem` operations. If you require strongly consistent reads, set the `ConsistentRead` parameter to `true` in the *Query* request.

**DynamoDB Encryption at Rest**

DynamoDB offers fully managed encryption at rest. DynamoDB encryption at rest provides enhanced security by encrypting your data at rest. The service uses an AWS Key Management Service (AWS KMS) managed encryption key for DynamoDB. This functionality reduces the operational burden and complexity involved in protecting sensitive data.

Enable encryption for any tables that contain sensitive data. You can enable encryption at rest using the AWS Management Console, AWS CLI, or the DynamoDB API.



You can enable encryption at rest only when you create a *new* DynamoDB table. You cannot enable encryption at rest on an existing table. *After encryption at rest is enabled, you cannot disable it.*

## How Encryption Works

DynamoDB encryption at rest provides an additional layer of data protection by securing your data from unauthorized access to the underlying storage. Organizational and industry policies, or government regulations and compliance requirements, might require the use of encryption at rest to protect your data. You can use encryption to increase the data security of the applications that you deploy to the cloud.

With encryption at rest, you can enable encryption for all of your DynamoDB data at rest, including the data that is persisted in your DynamoDB tables, local secondary indexes, and global secondary indexes. Encryption at rest encrypts your data by using 256-bit AES encryption, also known as AES-256 encryption. It works at the table level and encrypts both the base table and its indexes.

Encryption at rest automatically integrates with AWS KMS for managing the service default key to encrypt your tables. If a service default key does not exist when you create your encrypted DynamoDB table, AWS KMS automatically creates a new key. Encrypted tables that you create in the future use this key. AWS KMS combines secure, highly available hardware and software to provide a key management system scaled for the cloud.

Using the same AWS KMS service default key that encrypts the table, the following elements are also encrypted:

- DynamoDB base tables
- Local secondary indexes
- Global secondary indexes

After you encrypt your data, DynamoDB handles decryption of your data transparently with minimal impact on performance. You do not need to modify your applications to use encryption.



DynamoDB cannot read your table data unless it has access to the service default key stored in your AWS KMS account. DynamoDB uses envelope encryption and key hierarchy to encrypt data. Your AWS KMS encryption key is used to encrypt the root key of this key hierarchy.



DynamoDB does not call AWS KMS for every DynamoDB operation. The key refreshes once every 5 minutes per client connection with active traffic.

## Considerations for Encryption at Rest

Before you enable encryption at rest on a DynamoDB table, consider the following:

- When you enable encryption for a table, all the data stored in that table is encrypted. You cannot encrypt only a subset of items in a table.
- DynamoDB uses a service default key for encrypting all of your tables. If this key does not exist, it is created for you. Remember, you cannot disable service default keys.
- Encryption at rest encrypts data only while it is static (at rest) on a persistent storage media. If data security is a concern for data in transit or data in use, you must take the following additional measures:

**Data in transit:** Protect your data while it is actively moving over a public or private network by encrypting sensitive data on the client side or by using encrypted connections, such as HTTPS, Secure Socket Layer (SSL), Transport Layer Security (TLS), and File Transfer Protocol Secure (FTPS).

**Data in use:** Protect your data before sending it to DynamoDB by using client-side encryption.

**On-demand backup and restore:** You can use on-demand backup and restore with encrypted tables, and you can create a backup of an encrypted table. The table that is restored with this backup has encryption enabled.



Currently, you cannot enable encryption at rest for DynamoDB Streams. If encryption at rest is a compliance/regulatory requirement, turn off DynamoDB Streams for encrypted tables.

## IAM Policy Conditions for Fine-Grained Access Control

When you grant permissions in DynamoDB, you can configure conditions that determine how a permissions policy takes effect. In DynamoDB, you can specify conditions when granting permissions by using an IAM policy. For example, you can set the following configurations:

- Grant permissions to allow users read-only access to certain items and attributes in a table or a secondary index.
- Grant permissions to allow users write-only access to certain attributes in a table, based on the identity of that user.

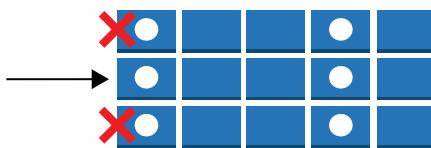
In DynamoDB, you can specify conditions in an IAM policy by using condition keys.

## Examples of Permissions

In addition to controlling access to DynamoDB API actions, you can also control access to individual data items and attributes. For example, you can do the following:

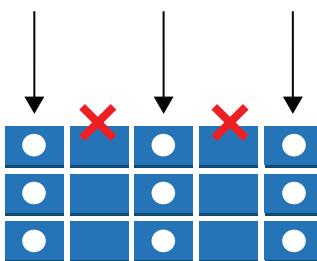
- Grant permissions on a table but restrict access to specific items in that table based on certain primary key values. An example might be a social networking application for games, where all users' game data is stored in a single table, but no users can access data items that they do not own, as shown in Figure 14.12.

**FIGURE 14.12** Granting permissions on a table



- Hide information so that only a subset of attributes is visible to the user. An example might be an application that displays flight data for nearby airports based on the user's location. Airline names, arrival and departure times, and flight numbers are displayed. However, attributes such as pilot names or number of passengers are hidden, as shown in Figure 14.13.

**FIGURE 14.13** Hiding information on a table



To implement this kind of fine-grained access control, write an IAM permissions policy that specifies conditions for accessing security credentials and the associated permissions and then apply the policy to IAM users, groups, or roles. Your IAM policy can restrict access to individual items in a table, access to the attributes in those items, or both at the same time.



You can use web identity federation to control access by users who are authenticated by Login with Amazon, Facebook, or Google.

Use the *IAM condition element* to implement a fine-grained access control policy. By adding a condition element to a permissions policy, you can allow or deny access to items and attributes in DynamoDB tables and indexes, based on your particular business requirements.

For example, in Figure 14.1, the game lets players select from and play a variety of games. The application uses a DynamoDB table named GameScores to track high scores and other user data. Each item in the table is uniquely identified by a user ID and the name of the game that the user played. The GameScores table has a primary key consisting of a partition key (`UserId`) and sort key (`GameTitle`). Users have access only to game data associated with their user ID. A user who wants to play a game must belong to an IAM role named GameRole, which has a security policy attached to it.

To manage user permissions in this application, you could write a permissions policy such as the following:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "AllowAccessToOnlyItemsMatchingUserID",  
            "Effect": "Allow",  
            "Action": [  
                "dynamodb:GetItem",  
                "dynamodb:BatchGetItem",  
                "dynamodb:Query",  
                "dynamodb:PutItem",  
                "dynamodb:UpdateItem",  
                "dynamodb:DeleteItem",  
                "dynamodb:BatchWriteItem"  
            ],  
            "Resource": [  
                "arn:aws:dynamodb:us-west-2:123456789012:table/GameScores"  
            ],  
            "Condition": {  
                "ForAllValues:StringEquals": {  
                    "dynamodb:LeadingKeys": [  
                        "${www.amazon.com:user_ID}"  
  
                    ],  
                    "dynamodb:Attributes": [  
                        "UserId",  
                        "GameTitle",  
                        "Wins",  
                        "Losses",  
                        "Score"  
                    ]  
                }  
            }  
        }  
    ]  
}
```

```
        "TopScore",
        "TopScoreDateTime"
    ],
},
"StringEqualsIfExists": {
    "dynamodb:Select": "SPECIFIC_ATTRIBUTES"
}
}
]
}
```

In addition to granting permissions for specific DynamoDB actions (Action element) on the GameScores table (Resource element), the Condition element uses the condition keys specific to DynamoDB that limit the permissions as follows:

**dynamodb:LeadingKeys** This condition key enables users to access only the items where the partition key value matches their user ID. This ID, \${www.amazon.com:user\_ID}, is a substitution variable.

**dynamodb:Attributes** This condition key limits access to the specified attributes so that only the actions listed in the permissions policy can return values for these attributes. In addition, the StringEqualsIfExists clause ensures that the application provides a list of specific attributes to act upon, and that the application cannot request all attributes.

When an IAM policy is evaluated, the result is either true (access is allowed) or false (access is denied). If any part of the Condition element is false, the entire policy evaluates to false and access is denied.



If you use dynamodb:Attributes, you must configure the names of all the *primary key* and *index key attributes* for the table and any secondary indexes that the policy lists. Otherwise, DynamoDB is unable to use these key attributes to perform the requested action.

## Configure Conditions with Condition Keys

AWS provides a set of predefined condition keys for all AWS offerings and services that support IAM for access control. For example, use the aws:SourceIp condition key to check the requester's IP address before allowing an action to be performed.



Condition keys are case-sensitive.

Table 14.3 displays the DynamoDB service-specific condition keys that apply to DynamoDB.

**TABLE 14.3** DynamoDB Condition Keys

DynamoDB Condition Key	Description
dynamodb:LeadingKeys	Represents the first key attribute of a table and is the partition key for a simple primary key (partition key) or a composite primary key (partition key and sort key). In addition, you must use the ForAllValues modifier when using LeadingKeys in a condition.
dynamodb:Select	Represents the Select parameter of a Query or Scan request using the following values: <ul style="list-style-type: none"> <li>▪ ALL_ATTRIBUTES</li> <li>▪ ALL_PROJECTED_ATTRIBUTES</li> <li>▪ SPECIFIC_ATTRIBUTES</li> <li>▪ COUNT</li> </ul>
dynamodb:Attributes	Represents a list of the attribute names in a request or the attributes that return from a request. Attributes values are named the same way and have the same meaning as the parameters for certain DynamoDB API actions: <ul style="list-style-type: none"> <li>▪ AttributesToGet Used by: BatchGetItem, GetItem, Query, Scan</li> <li>▪ AttributeUpdates Used by: UpdateItem</li> <li>▪ Expected Used by: DeleteItem, PutItem, UpdateItem</li> <li>▪ Item Used by: PutItem</li> <li>▪ ScanFilter Used by: Scan</li> </ul>
dynamodb:ReturnValues	Represents the ReturnValues parameter of a request: <ul style="list-style-type: none"> <li>▪ ALL_OLD</li> <li>▪ UPDATED_OLD</li> <li>▪ ALL_NEW</li> <li>▪ UPDATED_NEW</li> <li>▪ NONE</li> </ul>
dynamodb:ReturnConsumedCapacity	Represents the ReturnConsumedCapacity parameter of a request: <ul style="list-style-type: none"> <li>▪ TOTAL</li> <li>▪ NONE</li> </ul>

## On-Demand Backup and Restore

You can create on-demand backups and enable point-in-time recovery for your DynamoDB tables. DynamoDB on-demand backups enable you to create full backups of your tables for long-term retention and archival for regulatory compliance. You can back up and restore your DynamoDB table data anytime either in the AWS Management Console or as an API call. Backup and restore actions execute with zero impact on table performance or availability.

On-demand backup and restore scales without degrading the performance or availability of your applications. With this distributed technology, you can complete backups in seconds regardless of table size. You can create backups that are consistent across thousands of partitions without worrying about schedules or long-running backup processes. All backups are cataloged, discoverable, and retained until explicitly deleted.

In addition, on-demand backup and restore operations do not affect performance or API latencies. Backups are preserved regardless of table deletion. You can create table backups using the console, the AWS CLI, or the DynamoDB API.

The backup and restore functionality works in the same AWS Region as the source table. DynamoDB on-demand backups are available at no additional cost beyond the normal pricing that is associated with the backup storage size.

### Backups

When you create an on-demand backup, a time marker of the request is cataloged. The backup is created asynchronously by applying all changes until the time of the request to the last full table snapshot. Backup requests process instantaneously and become available for restore within minutes. Each time you create an on-demand backup, the entire table data is backed up. You can make an unlimited number of on-demand backups.

All backups in DynamoDB work without consuming any provisioned throughput on the table. DynamoDB backups do not enable causal consistency across items; however, the skew between updates in a backup is usually much less than a second. While a backup is in progress, you cannot perform certain operations, such as pausing or canceling the backup action, deleting the source table of the backup, or disabling backups on a table. However, you can use AWS Lambda functions to schedule periodic or future backups.



DynamoDB backups also include global secondary indexes, local secondary indexes, streams, and provisioned read and write capacity, in addition to the data.

### Restores

A table restores without consuming any provisioned throughput on the table. The destination table is set with the same provisioned read capacity units and write capacity units as the source table, as recorded at the time that you request the backup. The restore process also restores the local secondary indexes and the global secondary indexes.

You can only restore the entire table data to a new table from a backup. Restore times vary based on the size of the DynamoDB table that is being restored. You can write to the restored table only after it becomes active, and you cannot overwrite an existing table during a restore operation. You can use IAM policies for access control.

## Point-in-Time Recovery

You can enable *point-in-time recovery* (PITR) and create on-demand backups for your DynamoDB tables. Point-in-time recovery helps protect your DynamoDB tables from accidental write or delete operations. With point-in-time recovery, you do not have to worry about creating, maintaining, or scheduling on-demand backups. DynamoDB maintains incremental backups of your table. In addition, point-in-time operations do not affect performance or API latencies. You can enable point-in-time recovery using the AWS Management Console, AWS CLI, or the DynamoDB API.

## How Point-in-Time Recovery Works

When it is enabled, point-in-time recovery provides continuous backups until you explicitly turn it off. After you enable point-in-time recovery, you can restore to any point in time within `EarliestRestorableDateTime` and `LatestRestorableDateTime`. `LatestRestorableDateTime` is typically 5 minutes before the current time. The point-in-time recovery process always restores to a new table. For `EarliestRestorableDateTime`, you can restore your table to any point in time during the last 35 days. The retention period is a fixed 35 days (five calendar weeks) and cannot be modified. Any number of users can execute up to four concurrent restores (any type of restore) in a given account.

When you restore using point-in-time recovery, DynamoDB restores your table data to a new table and to the state based on the selected date and time (day:hour:minute:second). In addition to the data, the following are also included on the newly restored table using point-in-time recovery:

- Global secondary indexes
- Local secondary indexes
- Provisioned read and write capacity
- Encryption settings

After restoring a table, you must manually set up the following on the restored table:

- Scaling policies
- IAM policies
- Amazon CloudWatch metrics and alarms
- Tags
- Stream settings
- TTL settings
- Point-in-time recovery settings

## Considerations for Point-in-Time Recovery

Before you enable point-in-time recovery on a DynamoDB table, consider the following:

- If you disable point-in-time recovery and then later re-enable it on a table, reset the start time for which you can recover that table. As a result, you can only immediately restore that table using the `LatestRestorableDateTime`.
- If you must recover a deleted table that had point-in-time recovery enabled, you must contact AWS Support to restore that table within the 35-day recovery window.
- You can enable point-in-time recovery on each local replica of a global table. When you restore the table, the backup restores to an independent table that is not part of the global table.
- You can enable point-in-time recovery on an encrypted table.
- AWS CloudTrail logs all console and API actions for point-in-time recovery to enable logging, continuous monitoring, and auditing.

# Amazon ElastiCache

*Amazon ElastiCache* is a web service that makes it easy to set up, manage, and scale distributed in-memory cache environments on the AWS Cloud. It provides a high-performance, resizable, and cost-effective in-memory cache while removing the complexity associated with deploying and managing a distributed cache environment.

You can use ElastiCache to store the application state. Applications often store session data in memory, but this approach does not scale well. To address scalability and provide a shared data storage for sessions that can be accessible from any individual web server, abstract the HTTP sessions from the web servers themselves. A common solution is to leverage an *in-memory key-value* store. ElastiCache supports the following open-source in-memory caching engines:

- *Memcached* is an open source, high-performance, distributed memory object caching system that is widely adopted by and protocol-compliant with ElastiCache.
- *Redis* is an open source, in-memory data structure store that you can use as a database cache and message broker. ElastiCache supports Master/Slave replication and Multi-AZ replication that you can use to achieve cross-Availability Zone redundancy.

ElastiCache is an in-memory cache. Caching frequently used data is one of the most important performance optimizations that you can make in your applications. Compared to retrieving data from an in-memory cache, querying a database is a much more expensive operation. By storing frequently accessed data in-memory, you can greatly improve the speed and responsiveness of read-intensive applications. For instance, application state for a web application can be stored in an in-memory cache, as opposed to storing state data in a database.

While key-value data stores are fast and provide submillisecond latency, the added network latency and cost are the drawbacks. An added benefit of leveraging key-value stores is that they can also cache any data, not only HTTP sessions, which helps boost the overall performance of your applications.

## Considerations for Choosing a Distributed Cache

One consideration when choosing a distributed cache for session management is determining the number of nodes necessary to manage the user sessions. You can determine this number by how much traffic is expected and how much risk is acceptable. *In a distributed session cache, the sessions are divided by the number of nodes in the cache cluster. In the event of a failure, only the sessions that are stored on the failed node are affected.* If reducing risk is more important than cost, adding additional nodes to reduce further the percentage of stored sessions on each node may be ideal even when fewer nodes are sufficient.

Another consideration may be whether the sessions must be replicated. Some key-value stores offer replication through read replicas. If a node fails, the sessions are not entirely lost. Whether replica nodes are important in your individual architecture may inform you as to which key-value store you should use. ElastiCache offerings for in-memory key-value stores include ElastiCache for Redis, which supports replication, and ElastiCache for Memcached, which does not support replication.

There are a number of ways to store sessions in key-value stores. Many application frameworks provide libraries that can abstract some of the integration required to Get/Set those sessions in memory. In other cases, you can write your own session handler to persist the sessions directly.

ElastiCache makes it easy to deploy, operate, and scale an in-memory cache in the cloud. ElastiCache improves the performance of web applications by enabling you to retrieve information from fast, managed, in-memory caches instead of relying entirely on slower disk-based databases.

Use Memcached if you require the following:

- Use a simple data model
- Run large nodes with multiple cores or threads
- Scale out or scale in
- Partition data across multiple shards
- Cache objects, such as a database

Use Redis if you require the following:

- Work with complex data types
- Sort or rank in-memory datasets
- Persist the key store
- Replicate data from the primary to one or more read replicas for read-intensive applications

- Automate failover if the primary node fails
- Publish and subscribe (pub/sub): the client is informed of events on the server
- Back up and restore data

Use Table 14.4 to determine which product best fits your needs.

**TABLE 14.4** Memcached or Redis

Capability	Memcached	Redis
Simple cache to offload DB burden	✓	✓
Ability to scale horizontally	✓	
Multithreaded performance	✓	
Advanced data types		✓
Sorting/ranking datasets		✓
Pub/sub capability		✓
Multi-AZ with auto-failover		✓
Persistence		✓

## ElastiCache Terminology

This section describes some of the key terminology that ElastiCache uses.

### Nodes

A *node* is the smallest building block of an ElastiCache deployment. A node is a fixed-size chunk of secure, network-attached RAM. Each node runs an instance of Memcached or Redis, depending on which you select when you create the cluster.

### Clusters

Each ElastiCache deployment consists of one or more nodes in a *cluster*. When you create a cluster, you may choose from many different nodes based on the requirements of both your solution case and your capacity. One Memcached cluster can be as large as 20 nodes. Redis clusters consist of a single node; however, you can group multiple clusters into a Redis replication group.

The individual node types are derived from a subset of the Amazon EC2 instance type families, such as t2, m3, and r3. The t2 cache node family is ideal for development and

low-volume applications with occasional bursts, but certain features may not be available. The *m3* family is a mix of memory and compute, whereas the *r3* family is optimized for memory-intensive workloads.

Based on your requirements, you may decide to have a few large nodes or many smaller nodes in your cluster or replication group. As demand for your application fluctuates, you may add or remove nodes over time. Each node type has a preconfigured amount of memory, with a small portion of that memory reserved for both the caching engine and operating system.

Though it is unlikely, always plan for the possible failure of an individual cache node. For a Memcached cluster, decrease the impact of the failure of a cache node by using a larger number of nodes with a smaller capacity instead of a few large nodes.

If ElastiCache detects the failure of a node, it provisions a replacement and then adds it back to the cluster. During this time, your database experiences increased load because any requests that would have been cached now need to be read from the database. For Redis clusters, ElastiCache detects failures and replaces the primary node. If you enable a Multi-AZ replication group, a read replica automatically is promoted to primary automatically to primary.

**Replication group** A *replication group* is a collection of Redis clusters with one primary read/write cluster and up to five secondary, read-only clusters called *read replicas*. Each read replica maintains a copy of the data from the primary cluster. Asynchronous replication mechanisms keep the read replicas synchronized with the primary cluster. Applications can read from any cluster in the replication group. Applications can write only to the primary cluster. Read replicas enhance scalability and guard against data loss.

**Endpoint** An *endpoint* is the unique address your application uses to connect to an ElastiCache node or cluster. Memcached and Redis have the following characteristics with respect to endpoints:

- A Memcached cluster has its own endpoint and a configuration endpoint.
- A standalone Redis cluster has an endpoint to connect to the cluster for both reads and writes.
- A Redis replication group has two types of endpoints.
  - The *primary endpoint* connects to the primary cluster in the replication group.
  - The *read endpoint* points to a specific cluster in the replication group.

## Cache Scenarios

ElastiCache caches data as key-value pairs. An application can retrieve a value corresponding to a specific key. An application can store an item in cache by a specific key, value, and an expiration time. *Time to live* (TTL) is an integer value that specifies the number of seconds until the key expires.

A *cache hit* occurs when an application requests data from the cache, the data is both present and not expired in the cache, and it returns to the application. A *cache miss* occurs if an application requests data from the cache, and it is not present in the cache (returning a

null). In this case, the application requests and receives the data from the database and then writes the data to the cache.

## Strategies for Caching

The strategy or strategies that you want to implement for populating and maintaining your cache depend on what data you are caching and the access patterns to that data. For example, you would likely not want to use the same strategy for a top-10 leaderboard on a gaming site, Facebook posts, and trending news stories.

### Lazy Loading

*Lazy loading* loads data into the cache only when necessary. Whenever your application requests data, it first makes the request to the ElastiCache cache. If the data exists in the cache and it is current, ElastiCache returns the data to your application. If the data does not exist in the cache or the data in the cache has expired, your application requests the data from your data store, which returns the data to your application. Your application then writes the data received from the store to the cache so that it can be retrieved more quickly the next time that it is requested.

#### Advantages of Lazy Loading

- Only requested data is cached.

Because most data is never requested, lazy loading avoids filling up the cache with data that is not requested.

- Node failures are not fatal.

When a new, empty node replaces a failed node, the application continues to function, though with increased latency. As requests are made to the new node, each missed cache results in a query of the database and adding the data copy to the cache so that subsequent requests are retrieved from the cache.

#### Disadvantages of Lazy Loading

- There is a cache miss penalty.

Each cache miss results in three trips:

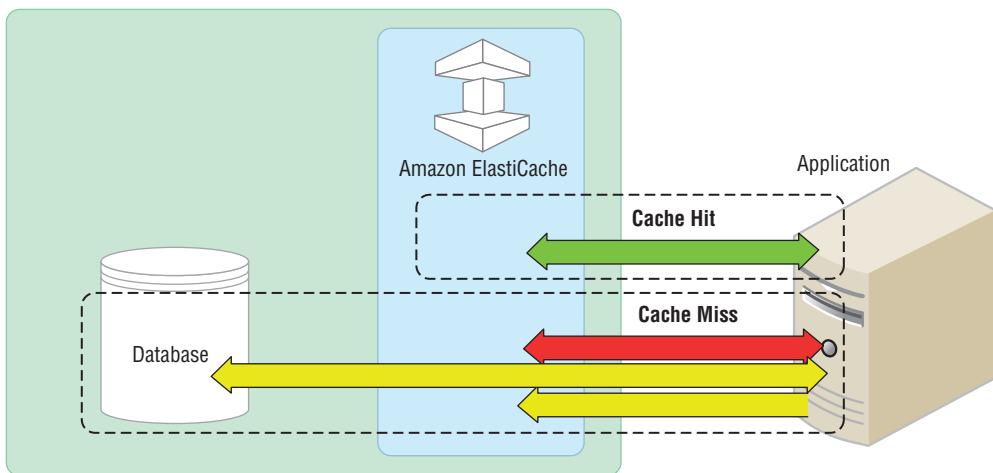
- Initial request for data from the cache
- Querying of the database for the data
- Writing the data to the cache

This can cause a noticeable delay in data getting to the application.

- Stale data.

The application may receive stale data because another application may have updated the data in the database behind the scenes.

Figure 14.14 summarizes the advantages and disadvantages of lazy loading.

**FIGURE 14.14** Lazy loading caching

### Write-Through

The *write-through* strategy adds data or updates data in the cache whenever data is written to the database.

#### Advantages of Write-Through

- The data in the cache is never stale.  
Because the data in the cache updates every time it is written to the database, the data in the cache is always current.

#### Disadvantages of Write-Through

- Write penalty  
Every write involves two trips: a write to the cache and a write to the database.
- Missing data  
When a new node is created either to scale up or replace a failed node, the node does not contain all data. Data continues to be missing until it is added or updated in the database. In this scenario, you might choose to use a lazy caching approach to repopulate the cache.
- Unused data  
Because most data is never read, there can be a lot of data in the cluster that is never read.
- Cache churn  
The cache may be updated often if certain records are updated repeatedly.

## Data Access Patterns

Retrieving a flat key from an in-memory cache is faster than the most performance-tuned database query. Analyze the access pattern of the data before you determine whether you should store it in an in-memory cache.

### Example 10: Cache Static Elements

An example of data to cache is a list of products in a catalog. For a high-volume web application, the list of products could be returned thousands of times per second. Though it may seem like a good idea to cache the most frequently requested items, your application may also benefit from caching items that are not frequently accessed.

You should not store certain data elements in an in-memory cache. For instance, if your application produces a unique page on every request, you probably do not want to cache the page results. However, though the page is different every time, it makes sense to cache the aspects of the page that are static.

## Scaling Your Environment

As your workloads evolve over time, you can use ElastiCache to change the size of your environment to meet the requirements of your workloads. To meet increased levels of write or read performance, expand your cluster horizontally by adding cache nodes. To scale your cache vertically, select a different cache node type.

**Scale horizontally** ElastiCache functionality enables you to scale the size of your cache environment horizontally. This functionality differs depending on the cache engine you select. With Memcached, you can partition your data and scale horizontally to 20 nodes or more. A Redis cluster consists of a single cache node that handles read and write transactions. You can create additional clusters to include a Redis replication group. Although you can have only one node handle write commands, you can have up to five read replicas handle read-only requests.

**Scale vertically** The ElastiCache service does not directly support vertical scaling of your cluster. You can create a new cluster with the desired cache node types and begin redirecting traffic to the new cluster.



Understand that a new Memcached cluster starts empty. By comparison, you can initialize a Redis cluster from a backup.

## Replication and Multi-AZ

Replication is an effective method for providing speedy recovery if a node fails and for serving high quantities of read queries beyond the capacities of a single node. ElastiCache clusters running Redis support both. In contrast, cache clusters running Memcached are standalone in-memory services that do not provide any data redundancy-protection services. Cache clusters running Redis support the notion of replication groups. A *replication group* consists of up to six clusters, with five of them designated as read replicas. By using a replication group, you can scale horizontally by developing code in the application to offload reads to one of the five replicas.

### Multi-AZ Replication Groups

With ElastiCache, you can provision a *Multi-AZ replication group* that allows your application to raise the availability and reduce the loss of data. Multi-AZ streamlines the procedure of dealing with a failure by automating the replacement and failover from the primary node.

If the primary node goes down or is otherwise unhealthy, Multi-AZ selects a replica and promotes it to become the new primary; then a new node is provisioned to replace the failed one. ElastiCache updates the DNS entry of the new primary node to enable your application to continue processing without any changes to the configuration of the application and with only minimal disruption. ElastiCache replication is handled asynchronously, meaning that there will be a small delay before the data is available on all cluster nodes.

## Backup and Recovery

ElastiCache clusters that run Redis support *snapshots*. Use *snapshots* to persist your data from your in-memory key-value stores to disk. Each snapshot is a full clone of the data that you can use to recover to a specific point in time or to create a copy for other purposes. Snapshots are not available to clusters that use the Memcached caching engine. This is because Memcached is a purely in-memory, key-value store, and it always starts empty. ElastiCache uses the native backup capabilities of Redis and generates a standard Redis database backup file, which is stored in Amazon S3.

Snapshots need memory and compute resources to perform, and this can possibly have a performance impact in heavily used clusters. ElastiCache attempts different backup techniques depending on the amount of memory currently available. As a best practice, set up a replication group and perform the snapshot against one of the read replicas instead of creating the snapshot against the primary node. You can automate the creation of snapshots on a schedule, or you can manually initiate a snapshot. Additionally, you can configure a window when a snapshot will be completed and then configure how many days of backups you want to save. Manual snapshots are stored indefinitely until you delete them.

It does not matter whether the snapshot was created manually or automatically. You can use the snapshot to provision a new cluster. The new cluster has the same configuration

as the source cluster by default, but you can override these settings. You can also restore a snapshot from the \*.rdb file that is generated from any other Redis compatible cluster. The Redis \*.rdb file is a binary representation of the in-memory store. This binary file is sufficient to restore the Redis state completely.

## Control Access

The primary way to configure access to your ElastiCache cluster is by restricting connectivity to your cluster through a security group. You can define a security group and add one or more inbound rules that restrict the source traffic. When a cache cluster is deployed inside a virtual private cloud, every node is assigned a private IP address within one or more subnets that you choose. You cannot access individual nodes from the internet or from Amazon EC2 instances outside of the Amazon Virtual Private Cloud (Amazon VPC). You can use the access control lists (ACLs) to constrain network inbound traffic.

Access to manage the configuration and infrastructure of the cluster is controlled separately from access to the actual Memcached or Redis service endpoint. Using the IAM service, you can define policies that control which AWS users can manage the ElastiCache infrastructure.

The ability to configure the cluster and govern the infrastructure is handled independently from access to the actual cache cluster endpoint, which is managed by using the IAM service. Using IAM, you can set up policies that determine which users can manage the ElastiCache infrastructure.

# Amazon Simple Storage Service

There are situations when storing state requires the storage of larger files. This may be the case when your application deals with user uploads or interim results of batch processes. In such cases, consider using Amazon Simple Storage Service (Amazon S3) as your store.

Amazon S3 is storage for the internet. It is designed to make web-scale computing easier for developers. Amazon S3 has a simple web services interface that you can use to store and retrieve any amount of data, at any time, from anywhere on the web. The service aims both to maximize benefits of scale and to pass those benefits on to developers.

## Amazon S3 Core Concepts

Amazon S3 is a stateless application that does not save client data that generates in one session for use in the next session with that client. Each session starts as if it was the first time, and responses are not dependent on data from a previous session. This means that the server does not store any state about the client session. Instead, the session data is stored on the client and passed to the server as requested.

## Buckets

A *bucket* is a container for objects stored in Amazon S3. Every object is contained in a bucket. For example, if the object named `photos/car.jpg` is stored in the `anitacrandle` bucket, then it is addressable using the URL `http://anitacrandle.s3.amazonaws.com/photos/car.jpg`.

Buckets serve several purposes:

- They organize the Amazon S3 namespace at the highest level.
- They identify the account responsible for storage and data transfer charges.
- They play a role in access control.
- They serve as the unit of aggregation for usage reporting.

## Creating a Bucket

Amazon S3 provides APIs for creating and managing buckets. By default, you can create up to 100 buckets in each of your accounts. If you need more buckets, increase your bucket limit by submitting a service limit increase.

When you create a bucket, provide a name for the bucket, and then choose the AWS Region where you want to create the bucket. You can store any number of objects in a bucket.

Create a bucket by using any of the following methods:

- Amazon S3 console
- Programmatically, using the AWS SDKs

When using the AWS SDKs, first create a client and then use the client to deliver a request to create a bucket. When you create the client, you can configure an AWS Region. US East (N. Virginia) is the default region. You can also configure an AWS Region in your request to create the bucket.

If you create a client specific to the US East (N. Virginia) Region, the client uses this endpoint to communicate with: Amazon S3: `s3.amazonaws.com`. You can use this client to create a bucket in any AWS Region in your create bucket request. If you do not specify a region, Amazon S3 creates the bucket in the US East (N. Virginia) Region. If you select an AWS Region, Amazon S3 creates the bucket in the specified region. If you create a client specific to any other AWS Region, it maps to the region-specific endpoint: `s3-.amazonaws.com`. For example, if you create a client and specify the `us-east-2` Region, it maps to the following region-specific endpoint: `s3-us-east-2.amazonaws.com`.

## Regions

You can choose the geographical region where Amazon S3 stores the buckets that you create. You might choose a region to optimize latency, minimize costs, or address regulatory requirements. Objects stored in a region never leave the region unless you explicitly transfer them to another region. For example, objects stored in the EU (Ireland) Region never leave it.

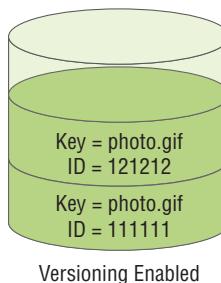
**Objects** *Objects* are the principal items stored in Amazon S3. Objects consist of object data and metadata. The data part is opaque to Amazon S3. The metadata is a set of name-value pairs that characterize the object. These include certain default metadata, such as the date last modified and standard HTTP metadata, such as Content-Type. It is also possible for you to configure custom metadata at the time of object creation.



A key (*name*) and a version ID uniquely identify an object within a bucket.

**Keys** A *key* is the unique identifier for an object within a bucket. Every object in a bucket has exactly one key. Because the combination of a bucket, key, and version ID uniquely identifies each object, Amazon S3 is like a basic data map between bucket + key + version and the object itself. Every object in Amazon S3 can be uniquely addressed through the combination of the web service endpoint, bucket name, key, and, optionally, a version. Figure 14.15 displays one object with a key and a version.

**FIGURE 14.15** Object with key and ID

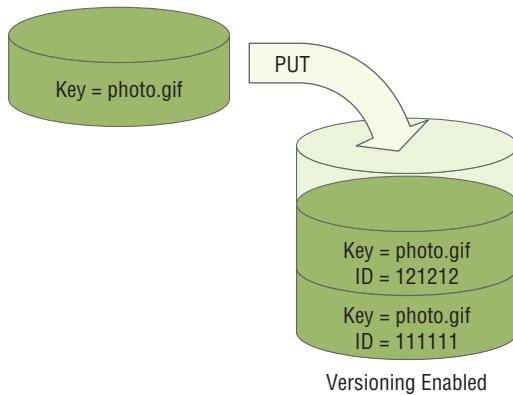


**Versioning** *Versioning* is a way to keep multiple variations of an object in the same bucket. You can use versioning to preserve, retrieve, and restore every version of every object stored in your Amazon S3 bucket. With versioning, you can recover from both unintended user actions and application failures.

In Figure 14.16, you can have two objects with the same key, but different version IDs, such as `photo.gif` (version 111111) and `photo.gif` (version 121212).

Versioning-enabled buckets allow you to recover objects from accidental deletion or overwrite. For instance:

- If you delete an object, instead of removing it permanently, Amazon S3 inserts a delete marker, which becomes the current object version. You can restore the previous version.
- You can delete object versions whenever you want.

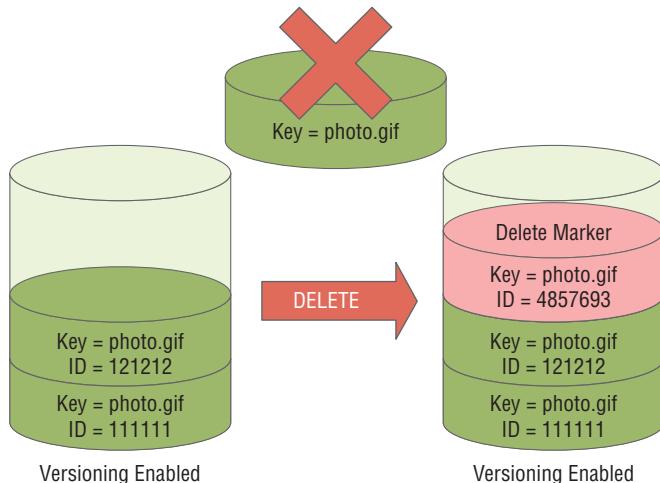
**FIGURE 14.16** Same key, different version

In addition, you can also define lifecycle configuration rules for objects that have a well-defined lifecycle to request Amazon S3 to expire current object versions or permanently remove noncurrent object versions. When your bucket is version-enabled or versioning is suspended, the lifecycle configuration actions work as follows:

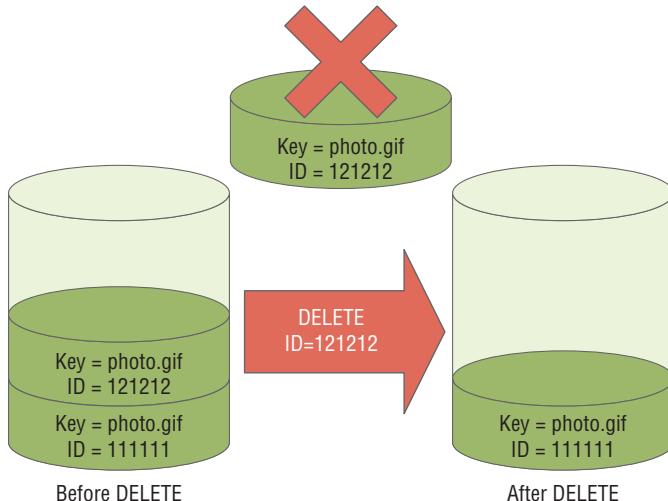
- The `Expiration` action applies to the current object version. Instead of deleting the current object version, Amazon S3 retains the current version as a noncurrent version by adding a delete marker, which then becomes the current version.
- The `NoncurrentVersionExpiration` action applies to noncurrent object versions, and Amazon S3 permanently removes these object versions. You cannot recover permanently removed objects.

A `Delete` request has the following use cases:

- When versioning is enabled, a simple `Delete` request cannot permanently delete an object. Instead, Amazon S3 inserts a delete marker in the bucket, and that marker becomes the current version of the object with a new ID.
- When you send a `Get` request for an object whose current version is a delete marker, Amazon S3 treats it as though the object has been deleted (even though it has not been erased) and returns a 404 error. Figure 14.17 shows that a simple `Delete` request does not actually remove the specified object. Instead, Amazon S3 inserts a delete marker.

**FIGURE 14.17** Delete marker

- To permanently delete versioned objects, you must use `DELETE Object versionId`. Figure 14.18 shows that deleting a specified object version permanently removes that object.

**FIGURE 14.18** Permanent delete

- If you overwrite an object, it results in a new object version in the bucket. You can restore the previous version.

## Accessing a Bucket

Use the Amazon S3 console to access your bucket. You can perform almost all bucket operations without having to write any code. If you access a bucket programmatically, Amazon S3 supports the RESTful architecture wherein your buckets and objects are resources, each with a resource Uniform Resource Identifier (URI) that uniquely identifies the resource.

Amazon S3 supports both path-style URLs and virtual hosted-style URLs to access a bucket:

- In a virtual hosted-style URL, the bucket name is part of the domain name in the URL. Here's an example:

`http://bucket.s3.amazonaws.com`

`http://bucket.s3-aws-region.amazonaws.com`

- In a path-style URL, the bucket name is not part of the domain (unless you use a region-specific endpoint). Here's an example:

US East (N. Virginia) Region endpoint: `http://s3.amazonaws.com/bucket`

Region-specific endpoint: `http://s3-aws-region.amazonaws.com/bucket`

In a path-style URL, the endpoint you use must match the AWS Region where the bucket resides. For example, if your bucket is in the South America (São Paulo) Region, you must use the `http://s3-saeast-1.amazonaws.com/bucket` endpoint.



Because you can access buckets by using either path-style or virtual hosted-style URLs, as a best practice, AWS recommends that you create buckets with DNS-compliant bucket names.

## Bucket Restrictions and Limitations

The account that created the bucket owns it. By default, you can create up to 100 buckets in each of your accounts. If you need additional buckets, increase your bucket limit by submitting a service limit increase.

- Bucket ownership is not transferable; however, if a bucket is empty, you can delete it.

After a bucket is deleted, the name becomes available for reuse, but the name may not be available for you to reuse for various reasons. For instance, another account could provision a bucket with the same name. Also, it might take some time before the name can be reused. Therefore, if you want to use the same bucket name after emptying the bucket, do not delete the bucket.

- There is no limit to the number of objects that you can store in a bucket and no difference in performance whether you use many buckets or only a few.

Moreover, you can store all of your objects in one bucket, or you can arrange all of your objects across multiple buckets.

- You cannot create a bucket within another bucket.

- The high-availability engineering of Amazon S3 is focused on GET, PUT, LIST, and DELETE operations.

Because bucket operations work against a centralized, global resource space, it is not appropriate to create or delete buckets on the high-availability code path of your application. It is better to create or delete buckets in a separate initialization or setup routine that you run less often.



If your solution is designed to create buckets automatically, develop a naming convention for your buckets that creates buckets that are globally unique. Also, make sure that your solution logic can create a different bucket name if a bucket name is already taken.

## Rules for Naming Buckets

After you create an S3 bucket, you cannot change the bucket name, so choose the name wisely.

The following are the rules for naming S3 buckets in all AWS Regions:

- Bucket names must be unique across all existing bucket names in Amazon S3.
- Bucket names must comply with DNS naming conventions.
- Bucket names must be between 3 and 63 characters long.
- Bucket names must not contain uppercase characters or underscores.
- Bucket names must start with a lowercase letter or number.
- Bucket names must be a series of one or more labels. Use a single period (.) to separate adjacent labels. Bucket names can contain lowercase letters, numbers, and hyphens. Each label must start and end with a lowercase letter or a number.
- Bucket names must not be formatted as an IP address (for example, 192.168.4.5).
- When you use virtual hosted-style buckets with SSL, the SSL wildcard certificate only matches buckets that do not contain periods. To work around this, use HTTP, or write your own certificate verification logic. AWS recommends that you do not use periods (“.”) in bucket names when using virtual hosted-style buckets.

## Working with Amazon S3 Buckets

Amazon S3 is cloud storage for the internet. To upload your data, such as images, videos, and documents, you must first create a bucket in one of the AWS Regions. You can then upload any number of objects to the bucket.

Amazon S3 is set up with buckets and objects as resources, and it includes APIs to interact with its resources. For instance, you can use the Amazon S3 API to create a bucket and then put objects in the bucket. Additionally, you can use the Amazon S3 web console to execute these actions. The console uses the Amazon S3 APIs to deliver requests to Amazon S3.

An Amazon S3 bucket name is globally unique regardless of the AWS Region in which you create the bucket. Configure the name at the time that you create the bucket. Amazon S3 creates buckets in a region that you choose. To minimize latency, reduce costs, or address regulatory requirements, choose any AWS Region that is geographically close to you.

The following are the most common operations executed through the API:

**Create a bucket** Create and name the bucket where you will store your objects.

**Write an object** Store data by creating or writing over an object. When you write an object, configure a unique key in the namespace of your bucket. Configure any access control that you want on the object.

**Read an object** Retrieve data. You can download data through HTTP or BitTorrent.

**Delete an object** Delete data.

**List keys** List the keys contained in one of your buckets. You can filter the key list based on a prefix.

**Make requests** Amazon S3 is a REST service. You can send requests to Amazon S3 using the REST API or the AWS SDK wrapper libraries. These libraries wrap the underlying Amazon S3 REST API, simplifying your programming tasks.

Every interaction with Amazon S3 is either authenticated or anonymous. Authentication is a process of verifying the identity of the requester trying to access an AWS offering. Authenticated requests must include a signature value that authenticates the request sender. The signature value is, in part, generated from the requester's AWS access keys (access key ID and secret access key).

If you are using the AWS SDK, the libraries compute the signature from the keys you provide. However, if you make direct REST API calls in your application, you must write the code to compute the signature and then add it to the request.

## Deleting or Emptying a Bucket

It is easy to delete an empty bucket. However, in certain situations, you may need to delete or empty a bucket that contains objects. In other situations, you may choose to empty a bucket instead of deleting it. This section explains various options that you can use to delete or empty a bucket that contains objects.

You can delete a bucket and its content programmatically using the AWS SDKs. You can also use lifecycle configuration on a bucket to empty its content and then delete the bucket. There are additional options, such as using Amazon S3 console and AWS CLI, but there are limitations on these methods based on the number of objects in your bucket and the bucket's versioning status.

### Deleting a Bucket Using Lifecycle Configuration

You can configure lifecycle on your bucket to expire objects. Amazon S3 then deletes expired objects. You can add lifecycle configuration rules to expire all or a subset of objects with a specific key name prefix. For example, to remove all objects in a bucket, set a life-cycle rule to expire objects one day after creation.

If your bucket has versioning enabled, you can also configure the rule to expire noncurrent objects. After Amazon S3 deletes all of the objects in your bucket, you can delete the bucket or keep it. If you want to only empty the bucket but not delete it, remove the life-cycle configuration rule you added to empty the bucket so that any new objects you create in the bucket remain in the bucket.

## Object Lifecycle Management

To manage your objects so that they are stored cost-effectively throughout their life, configure their lifecycle policy. A *lifecycle policy* is a set of rules that designates actions that Amazon S3 applies to a group of objects. There are two types of actions:

**Transition actions** Designate when objects transition from one storage class to another. For instance, you might decide to transition objects to the STANDARD\_IA storage class 45 days after you created them, or archive objects to the GLACIER storage class six months after you created them.

**Expiration actions** Designate when objects expire. Amazon S3 deletes expired objects on your behalf.

## When to Use Lifecycle Configuration

Set up lifecycle configuration rules for objects that have a clear-cut lifecycle. For example, set up configuration rules for these types of situations:

- If you upload logs periodically to a bucket, your solution may need access to them for a week or so. When you no longer need access to them, you may want to remove them.
- Some objects are frequently accessed for a specific period. When that period is over, they are sporadically accessed. You may not need real-time access to these objects, but your business or regulations might require that you archive them for a specific period. After that, you can delete them.
- You might upload certain types of data to Amazon S3 primarily for archival purposes. For instance, you might archive digital media, healthcare and financial data, database backups, and data that you must retain for regulatory compliance.

With lifecycle configuration policies, you can instruct Amazon S3 to transition objects to less expensive storage classes, archive them for later, or remove them altogether.

## Bucket Configuration Options

Amazon S3 supports various options to configure your bucket. For example, you can configure your bucket for website hosting, managing the lifecycle of objects in the bucket, and enabling all access to the bucket. Amazon S3 supports subresources for you to store, and it manages the bucket configuration information. Using the Amazon S3 API, you can create and manage these subresources. You can also use the Amazon S3 console or the AWS SDKs.

## Amazon S3 Consistency Model

Amazon S3 provides read-after-write consistency for PUT requests of new objects in your S3 bucket in all regions. However, if you make a HEAD or GET request to the key name (to determine whether the object exists) before creating the object, Amazon S3 provides eventual consistency for read-after-write.

Amazon S3 offers eventual consistency for overwriting PUT and DELETE requests in all regions.

Updates to a single key are atomic. For example, if you PUT to an existing key, a subsequent read might return the old data or the updated data, but it will not write corrupted or partial data.

Amazon S3 achieves high availability by replicating data across multiple servers within Amazon's data centers. If a PUT request is successful, your data is safely stored. However, information about the changes must replicate across Amazon S3, which can take time, so you might observe the following behaviors:

- A process writes a new object to Amazon S3 and immediately lists keys within its bucket. Until the change is fully propagated, the object might not appear in the list.
- A process replaces an existing object and immediately attempts to read it. Until the change is fully propagated, Amazon S3 might return the prior data.
- A process deletes an existing object and immediately attempts to read it. Until the deletion is fully propagated, Amazon S3 might return the deleted data.
- A process deletes an existing object and immediately lists keys within its bucket. Until the deletion is fully propagated, Amazon S3 might list the deleted object.

Amazon S3 does not currently support object locking. If two PUT requests are simultaneously made to the same key, the request with the latest timestamp takes effect. If this is an issue, build an object-locking mechanism into your application. Updates are key-based; there is no way to make atomic updates across keys. For example, you cannot make the update of one key dependent on the update of another key unless you design this functionality into your application.

Table 14.5 describes the characteristics of eventually consistent reads and consistent reads.

**TABLE 14.5** Amazon S3 Reads

Eventually Consistent Read	Consistent Read
▪ Stale reads possible	▪ No stale reads
▪ Lowest-read latency	▪ Potential higher-read latency
▪ Highest-read throughput	▪ Potential lower-read throughput

## Bucket Policies

*Bucket policies* are a centralized way to control access to buckets and objects based on numerous conditions, such as operations, requesters, resources, and aspects of the request. The policies are written using the IAM policy language and enable centralized management of permissions.

Individuals and companies can use bucket policies. When companies register with Amazon S3, they create an account. Thereafter, the company becomes synonymous with the account. Accounts are financially responsible for the Amazon resources they (and their employees) create. Accounts grant bucket policy permissions and assign employees

permissions based on a variety of conditions. For instance, an account could create a policy that grants a user read access in the following cases:

- From a particular S3 bucket
- From an account's corporate network
- During the weekend

An account can allow one user limited read and write access, while allowing other users to create and delete buckets. An account could allow several field offices to store their daily reports in a single bucket, allowing each office to write only to a certain set of names (e.g., Texas/\* or Alabama/\*) and only from the office's IP address range.

Unlike ACLs, which can add (grant) permissions only on individual objects, policies can grant or deny permissions across all (or a subset) of objects within a bucket. With one request, an account can set the permissions of any number of objects in a bucket. An account can use wildcards (similar to regular expression operators) on Amazon Resource Names (ARNs) and other values so that an account can control access to groups of objects that begin with a common prefix or end with a given extension, such as .doc.

Only the bucket owner is allowed to associate a policy with a bucket. Policies, written in the IAM policy language, allow or deny requests based on the following:

- Amazon S3 bucket operations (such as PUT Bucket acl) and object operations (such as PUT Object, or GET Object)
- Requester
- Conditions specified in the policy

An account can control access based on specific Amazon S3 operations, such as GetObject, GetObjectVersion, DeleteObject, or DeleteBucket.

The conditions can be such things as IP addresses, IP address ranges in Classless Inter-Domain Routing (CIDR) notation, dates, user agents, HTTP referrer, and transports, such as HTTP and HTTPS.

## Amazon S3 Storage Classes

Amazon S3 offers a variety of *storage classes* devised for different scenarios. Among these storage classes are *Amazon S3 STANDARD* for general-purpose storage of frequently accessed data; *Amazon S3 STANDARD\_IA* (Infrequent Access) for long-lived, but less frequently accessed data; and *GLACIER* for long-term archival purposes.

### Storage Classes for Frequently Accessed Objects

Amazon S3 provides storage classes for performance-sensitive use cases (millisecond access time) and frequently accessed data. Amazon S3 provides the following storage classes:

**STANDARD** *Standard* is the default storage class. If you do not specify the storage class when uploading an object, Amazon S3 assigns the STANDARD storage class.

**REDUCED\_REDUNDANCY** The *Reduced Redundancy Storage* (RRS) storage class is designed for noncritical, reproducible data that you can store with less redundancy than the STANDARD storage class.

Regarding durability, RRS objects have an average annual expected loss of 0.01 percent of objects. If an RRS object is lost, when requests are made to that object, Amazon S3 returns a 405 error.



The STANDARD storage class is more cost-effective than the REDUCED\_REDUNDANCY storage class; therefore, AWS recommends that you do not use the RRS storage class.

## Storage Classes for Infrequently Accessed Objects

The STANDARD\_IA and ONEZONE\_IA storage classes are designed for data that is long-lived and infrequently accessed. STANDARD\_IA and ONEZONE\_IA objects are available for millisecond access (similar to the STANDARD storage class). Amazon S3 charges a fee for retrieving these objects; thus, they are most appropriate for infrequently accessed data.

Possible use cases for STANDARD\_IA and ONEZONE\_IA are as follows:

- For storing backups
- For older data that is accessed infrequently but that still requires millisecond access

### **STANDARD\_IA and ONEZONE\_IA Storage Classes**

The STANDARD\_IA and ONEZONE\_IA storage classes are suitable for objects larger than 128 KB that you plan to store for at least 30 days. If an object is less than 128 KB, Amazon S3 charges you for 128 KB. If you delete an object before the 30-day minimum, you are charged for 30 days.

These storage classes differ as follows:

**STANDARD\_IA** Objects stored using this storage class are stored redundantly across multiple, geographically distinct Availability Zones (similar to the STANDARD storage class). STANDARD\_IA objects are resilient to data loss of an Availability Zone. This storage class provides more availability, durability, and resiliency than the ONEZONE\_IA class.

**ONEZONE\_IA** Amazon S3 stores the object data in only one Availability Zone, which makes it less expensive than STANDARD\_IA. However, the data is not resilient to the physical loss of the Availability Zone resulting from disasters, such as earthquakes and floods. The ONEZONE\_IA storage class is as durable as STANDARD\_IA, but it is less available and less resilient.

To determine when to use a particular storage class, follow these recommendations:

**STANDARD\_IA** Use for your primary copy (or only copy) of data that cannot be regenerated.

**ONEZONE\_IA** Use if you can regenerate the data if the Availability Zone fails.

## GLACIER Storage Class

You use the *GLACIER* storage class to archive data where access is infrequent. Objects that you archive are not available for real-time access. The *GLACIER* storage class offers the same durability and resiliency as the *STANDARD* storage class.

When you store objects in Amazon S3 with the *GLACIER* storage class, Amazon S3 uses the low-cost Amazon Simple Storage Service Glacier (Amazon S3 Glacier) service to store these objects. Though the objects are stored in Amazon S3 Glacier, these remain Amazon S3 objects that are managed in Amazon S3, and they cannot be accessed directly through Amazon S3 Glacier.

At the time that you create an object, it is not possible to specify *GLACIER* as the storage class. The way that *GLACIER* objects are created is by uploading objects first using *STANDARD* as the storage class. You can transition these objects to the *GLACIER* storage class by using lifecycle management.



You must restore the *GLACIER* objects to access them.

## Setting the Storage Class of an Object

Amazon S3 APIs offer support for setting or updating the storage class of objects. When you create a new object, configure its storage class. For example, when you create objects with the *PUT Object*, *POST Object*, and *Initiate Multipart Upload* APIs, add the *x-amz-storageclass* request header to configure a storage class. If you do not add this header, Amazon S3 uses *STANDARD*, the default storage class.

You can also change the storage class of an object that is already stored in Amazon S3 by making a copy of the object using the *PUT Object - Copy* API. Copy the object in the same bucket with the same key name, and configure request headers as follows:

- Set the *x-amz-metadata-directive* header to *COPY*.
- Set the *x-amz-storage-class* to the storage class that you want to use.

In a versioning-enabled bucket, you cannot change the storage class of a specific version of an object. When you copy it, Amazon S3 gives it a new version ID. You can direct Amazon S3 to change the storage class of objects by adding a lifecycle configuration to a bucket.

## Amazon S3 Default Encryption for S3 Buckets

Amazon S3 default encryption provides a way to set the default encryption behavior for an Amazon S3 bucket. You can set default encryption on a bucket so that all objects are encrypted when they are stored in the bucket. The objects are encrypted using server-side encryption with either Amazon S3 managed keys (SSE-S3) or AWS KMS managed keys (SSE-KMS).

When you use server-side encryption, Amazon S3 encrypts an object before saving it to disk in its data centers and then decrypts the object when you download it.

## Protecting Data Using Encryption

*Data protection* refers to protecting data while in transit (as it travels to and from Amazon S3), and at rest (while it is stored on disks in Amazon S3 data centers). You can protect data in transit by using SSL or by using client-side encryption with the following options of protecting data at rest in Amazon S3:

**Use server-side encryption** You request Amazon S3 to encrypt your object before saving it on disks in its data centers and then decrypt the object when you download it.

**Use client-side encryption** You can encrypt data on the client side and upload the encrypted data to Amazon S3 and then manage the encryption process, the encryption keys, and related tools.

## Protecting Data Using Server-Side Encryption

*Server-side encryption* is about data encryption at rest; that is, Amazon S3 encrypts your data at the object level as it writes it to disks in its data centers and decrypts it for you when you access it. As long as you authenticate your request and you have access permissions, there is no difference in the way that you access encrypted or unencrypted objects. For example, if you share your objects using a presigned URL, that URL works the same way for both encrypted and unencrypted objects.

Consider the following mutually exclusive options, depending on how you choose to manage the encryption keys:

**Use server-side encryption with Amazon S3 Managed Keys (SSE-S3)** Each object is encrypted with a unique key employing strong multifactor encryption. As an additional safeguard, it encrypts the key itself with a master key that it regularly rotates. Amazon S3 server-side encryption uses one of the strongest block ciphers available, 256-bit Advanced Encryption Standard (AES-256), to encrypt your data.

**Use server-side encryption with AWS KMS Managed Keys (SSE-KMS)** SSE-KMS is similar to SSE-S3, but with additional benefits and charges for using this service. There are separate permissions for the use of an envelope key (that is, a key that protects your data's encryption key) that provides added protection against unauthorized access of your objects in Amazon S3.

SSE-KMS also provides you with an audit trail of when your key was used and by whom. Additionally, you can create and manage encryption keys yourself or use a default key that is unique to you, the service you are using, and the region in which you are working.

**Use server-side encryption with customer-provided keys (SSE-C)** You manage the encryption keys, and Amazon S3 manages the encryption as it writes to disks. You also manage decryption when you access your objects.



When you list objects in your bucket, the list API will return a list of all objects, regardless of whether they are encrypted.

## Working with Amazon S3 Objects

Amazon S3 is a simple key-value store designed to store as many objects as you want. Store these objects in one or more buckets. An object consists of the following:

**Key** The key is the name that you assign to an object. The object key is used to retrieve the object.

**Version ID** Within a bucket, a key and version ID uniquely identify an object. The version ID is a string that Amazon S3 generates when you add an object to a bucket.

**Value** The information being stored. An object value can be any sequence of bytes. Objects can range in size from 0 to 5 terabytes (TB).

**Metadata** A set of key-value pairs with which you can store information about the object. You can assign metadata, referred to as *user-defined metadata*, to your objects in Amazon S3. Amazon S3 also assigns system metadata to these objects, which it uses for managing objects.

**Subresources** Amazon S3 uses the subresource mechanism to store object-specific additional information. Because subresources are subordinates to objects, they are always associated with an entity, such as an object or a bucket.

**Access control information** You can control access to the objects that you store in Amazon S3. Amazon S3 supports both the resource-based access control, such as an access control list (ACL) and bucket policies, and user-based access control.

## Object Keys and Metadata

Each Amazon S3 object is composed of several parts. These parts include the data, a key, and metadata. An *object key* (or key name) uniquely identifies the object in a bucket.

*Object metadata* is a set of name-value pairs. You can set the object metadata at the time that you upload an object. However, after you upload the object, you cannot modify object metadata. The only way to modify object metadata after it has been uploaded is to create a copy of the object.

When you upload an object, set the key name, which uniquely identifies the object in that bucket. As an example, in the Amazon S3 console, when you select a bucket, a list of objects that reside in your bucket is displayed. These names are the object keys. The name for a key is a sequence of Unicode characters whose UTF-8 encoding is limited to 1,024 bytes.



If you anticipate that your workload against Amazon S3 will exceed 100 requests per second, follow the Amazon S3 key naming guidelines for best performance.

## Object Key Naming Guidelines

Though you can use any UTF-8 characters in an object key name, the following best practices for key naming help ensure maximum compatibility with other applications. Each application might parse special characters differently. The following guidelines help you maximize compliance with DNS, web-safe characters, XML parsers, and other APIs.

The following character sets are generally safe for use in key names:

- a–z
- A–Z
- 0–9
- \_ (underscore)
- - (dash)
- . (period)
- ! (exclamation point)
- \* (asterisk)
- ' (apostrophe)
- , (comma)

The following are examples of valid object key names:

- 2your-customer
- your.great\_photos-2018/jane/yourvacation.jpg
- videos/2018/graduation/video1.wmv

The Amazon S3 data model is a flat structure: you create a bucket, and the bucket stores objects. There is no hierarchy of sub-buckets or subfolders; however, you can infer logical hierarchy by using key name prefixes and delimiters as the Amazon S3 console does. The Amazon S3 console supports the concept of folders.

## Object Metadata

There are two kinds of object metadata: system metadata and user-defined metadata.

### System-Defined Metadata

For each object stored in a bucket, Amazon S3 maintains a set of system metadata. Amazon S3 processes this system metadata as needed. For example, Amazon S3 maintains the object creation date and size metadata, and it uses this information as part of object management.

There are two categories of system metadata.

- Metadata, such as object creation date, is system controlled where only Amazon S3 can modify the value.
- Other system metadata are examples of system metadata whose values you control.

If your bucket is configured as a website, sometimes you might want to redirect a page request to another page or an external URL. In this case, a webpage is an object in your bucket. Amazon S3 stores the page redirect value as system metadata whose value you control.

When you create objects, you can configure values of these system metadata items or update the values when you need to.

Table 14.6 lists system-defined metadata and whether you can modify it.

**TABLE 14.6** System-Defined Metadata

Name	Description	Modifiable
Content-Length	Size of object in bytes.	No
Last-Modified	The object creation date or last modified date, whichever is the latest.	No
Content-MD5	The base64-encoded 128-bit MD5 digest of the object.	No
x-amz-server-side-encryption	Indicates whether server-side encryption is enabled for the object and whether that encryption is from the AWS Key Management Service (SSE-KMS) or from AWS managed encryption (SSE-S3).	Yes
x-amz-version-ID	Object version. When you enable versioning on a bucket, Amazon S3 assigns a version number to objects added to the bucket.	No
x-amz-delete-marker	In a bucket that has versioning enabled, this Boolean marker indicates whether the object is a delete marker.	No
x-amz-storage-class	Storage class used for storing the object.	Yes
x-amz-websiteredirect-location	Redirects requests for the associated object to another object in the same bucket, or to an external URL.	Yes
x-amz-server-sideencryption-aws-kmskey-ID	If x-amz-server-side-encryption is present and has the value of aws:kms, this indicates the ID of the AWS KMS master encryption key that was used for the object.	Yes
x-amz-server-sideencryption-customeralgorithm	Indicates whether server-side encryption with customer-provided encryption keys (SSE-C) is enabled.	Yes

## User-Defined Metadata

When uploading an object, you can also attach metadata to the object. Provide this optional information as a key-value pair when you deliver a PUT or POST request to create the object. When you upload objects using the REST API, the optional user-defined metadata names must begin with `x-amz-meta-` to distinguish them from other HTTP headers. When you retrieve the object using the REST API, this prefix is returned. User-defined metadata is a set of key-value pairs. Amazon S3 stores user-defined metadata keys in lowercase.

## Object Tagging

You can use object tagging as a way to categorize your objects. Each tag is a key-value pair. The following are tagging examples:

- Suppose that an object contains protected health information (PHI) data. You might tag the object by using the following key-value pair:

`PHI=True`

- Suppose that you store project files in your S3 bucket. You might tag these objects with a key called `Project`, and the following value:

`Project=Aqua`

- You can set up multiple tags to an object, as shown in the following:

`Project=Y`

`Classification=sensitive`

You can add tags to new objects and existing objects. Note the following:

- You can associate up to 10 tags with an object. Tags associated with an object must have unique tag keys.
- A tag key can be up to 128 Unicode characters in length, and tag values can be up to 256 Unicode characters in length.
- Key and values are case-sensitive.

Object key name prefixes also enable you to categorize storage. However, prefix-based categorization is one-dimensional. Consider the following object key names:

- `images/photo1.jpg`
- `projects/accountingproject/document.pdf`
- `project/financeproject/document2.pdf`

These key names have the prefixes `images/`, `projects/accountingproject/`, and `projects/financeproject/`. These prefixes enable one-dimensional categorization; that is, everything under a prefix is one category. For example, the prefix `projects/accountingproject` identifies all documents related to the project `accountingproject`.

Tagging creates another dimension. If you want photo1 in the project financeproject category, tag the object accordingly. In addition to data classification, tagging offers the following additional benefits:

- Object tags enable fine-grained access control of permissions. For instance, you could grant an IAM user permissions to read-only objects with certain tags.
- Object tags enable fine-grained object lifecycle policies where you can configure tag-based filters, in addition to key name prefixes, in a lifecycle policy.
- When using Amazon S3 analytics, you can configure filters to group objects together for analysis by object tags, by key name prefix, or by both prefix and tags.
- You can customize Amazon CloudWatch metrics to display information by specific tag filters.



Though it is acceptable to use tags to label objects that contain confidential information, the tags themselves should not contain any confidential data.

## Operations on Objects

Amazon S3 enables you to store (POST and PUT), retrieve (GET), and delete (DELETE) objects. You can retrieve an entire object or a portion of an object. If you have enabled versioning on the bucket, you can retrieve a specific version of the object. You can also retrieve a sub-resource associated with your object and update it where applicable. You can make a copy of your existing object.

Depending on the size of the object, consider the following when uploading or copying a file:

**Uploading objects** You can upload objects of up to 5 GB in size in a single operation. If your object is larger than 5 GB, use the multipart upload API. By using the multipart upload API, you can upload objects up to 5 TB each.

**Copying objects** The Copy operation creates a copy of an object that is already stored in Amazon S3. You can create a copy of your object up to 5 GB in size in a single atomic operation. However, for copying an object greater than 5 GB, use the multipart upload API.

## Getting Objects

You can retrieve objects directly from Amazon S3. The following options are available to you when retrieving objects:

**Retrieve an entire object** A single GET operation can return the entire object stored in Amazon S3.

**Retrieve object in parts** Use the Range HTTP header in a GET request to retrieve a specific range of bytes in an object stored in Amazon S3. Resume fetching other parts of the object when your application is ready. This ability to resume the download is useful when you need only portions of your object data. It is also useful where network connectivity is poor, and you must react to failures.

When you retrieve an object, its metadata is returned in the response headers. There are situations when you want to override certain response header values returned in a GET response. For example, you might override the Content-Disposition response header value in your GET request. To override these values, use the REST GET Object API to specify query string parameters in your GET request.

The AWS SDKs for Java, .NET, and PHP also provide necessary objects that you can use to specify values for these response headers in your GET request. When retrieving objects that are stored encrypted using server-side encryption, you must provide appropriate request headers.

### Sharing an Object with Others

All objects uploaded to Amazon S3 are private by default. Only the object owner has permission to access these objects. However, the object owner can choose to share objects with others by generating a presigned URL, using their own security credentials, to grant time-limited permission to interact with the objects.

When you create a presigned URL for your object, you must provide your security credentials, specify a bucket name and an object key, specify the HTTP method (GET to download the object), and specify both the expiration date and the time. The presigned URLs are valid only for the specified duration.

Anyone who receives the presigned URL can then access the object. For example, if you have an image in your bucket and both the bucket and object are private, you can share the image with others by generating a presigned URL. You can generate a presigned URL programmatically using the AWS SDK for Java and .NET.



Anyone with valid security credentials can create a presigned URL. However, to access an object successfully, only someone who has permission to perform the operation that the presigned URL is based upon can create the presigned URL.

### Uploading Objects

Depending on the size of the data that you are uploading, Amazon S3 provides the following options:

**Upload objects in a single operation** Use a single PUT operation to upload objects up to 5 GB in size. For objects that are up to 5 TB in size, use the multipart upload API.

**Upload objects in parts** The multipart upload API is designed to improve the upload experience for larger objects. Upload these object parts independently, in any order, and in parallel.

AWS recommends that you use multipart uploading in the following ways:

- If you are uploading large objects over a stable high-bandwidth network, use multipart uploading to maximize the use of your available bandwidth by uploading object parts in parallel for multithreaded performance.

- If you are uploading over a spotty network, use multipart uploading to increase resiliency to network errors by avoiding upload restarts. When using multipart uploading, you must retry uploading only parts that are interrupted during the upload. You do not need to restart uploading your object from the beginning.

When uploading an object, you can request that Amazon S3 encrypt it before saving it to disk and decrypt it when you download it.

## Uploading Objects Using Multipart Upload API

Use multipart upload to upload a single object as a set of parts. Each part is a contiguous portion of the object's data. You can upload these object parts independently and in any order. If transmission of any part fails, you can retransmit that part without affecting other parts. After all parts of your object are uploaded, Amazon S3 assembles these parts and then creates the object. When your object size reaches 100 MB, consider using multipart uploads instead of uploading the object in a single operation.

Using multipart upload provides the following advantages:

**Improved throughput** You can upload parts in parallel to improve throughput.

**Quick recovery from any network issues** Smaller part size minimizes the impact of restarting a failed upload resulting from a network error.

**Pause and resume object uploads** You can upload object parts over time. Once you initiate a multipart upload, there is no expiry; you must explicitly complete or abort the multipart upload.

**Begin an upload before you know the final object size** You can upload an object as you are creating it.

## Copying Objects

The Copy operation creates a copy of an object that is already stored in Amazon S3. You can create a copy of your object up to 5 GB in a single atomic operation. However, to copy an object that is greater than 5 GB, you must use the multipart upload API.

Using the Copy operation, you can do the following:

- Create additional copies of objects.
- Rename objects by copying them and then deleting the original ones.
- Move objects across Amazon S3 locations (for example, us-west-1 and EU).
- Change object metadata.

Each Amazon S3 object has metadata. It is a set of name-value pairs. You can set object metadata at the time you upload it. After you upload the object, you cannot modify object metadata. The only way to modify object metadata is to make a copy of the object and then set the metadata. In the Copy operation, you set the same object as the source and target.

Each object has metadata, which can be system metadata or user-defined. Users control some of the system metadata, such as storage class configuration to use for the object and configure server-side encryption. When you copy an object, both user-controlled

system metadata and user-defined metadata are also copied. Amazon S3 resets the system-controlled metadata. For example, when you copy an object, Amazon S3 resets the creation date of the copied object. You do not need to set any of these values in your copy request.

When you copy an object, you might decide to update several metadata values. For example, if your source object is configured to use standard storage, you might choose to use RRS for the object copy. You might also decide to alter user-defined metadata values present on the source object. If you choose to update any of the object's user-configurable metadata (system- or user-defined) during the copy, then you must explicitly specify all of the user-configurable metadata present on the source object in your request, even if you are changing only one of the metadata values.



If the source object is archived in Amazon S3 Glacier (the storage class of the object is GLACIER), you must first restore a temporary copy before you can copy the object to another bucket.

When you copy objects, you can request that Amazon S3 save the target object encrypted using an AWS KMS encryption key, an Amazon S3 managed encryption key, or a customer-provided encryption key. Accordingly, you must configure encryption information in your request. If the copy source is an object that is stored in Amazon S3 using server-side encryption with a customer-provided key, then provide encryption information in your request so that Amazon S3 can decrypt the object for copying.



Copying objects across locations incurs bandwidth charges.

## **Listing Object Keys**

You can list Amazon S3 object keys by prefix. By choosing a common prefix for the names of related keys and marking these keys with a special character that delimits hierarchy, you can use the list operation to select and browse keys in a hierarchical fashion. This can be likened to how files are stored in directories within a file system.

Amazon S3 exposes a list operation that enables you to enumerate the keys contained in a bucket. Keys are selected for listing by bucket and prefix. For instance, suppose that you have a bucket named dictionary that contains a key for every English word. You might make a call to list all of the keys in that bucket that start with the letter q. List results are returned in UTF-8 binary order. Whether you use the SOAP or REST list operations does not matter because they both return an XML document that has the names of matching keys and data about the object identified by each key.



SOAP support over HTTP is deprecated, but it is still available over HTTPS. New Amazon S3 features do not support SOAP. AWS recommends that you use either the REST API or the AWS SDKs.

Groups of keys that share a prefix terminated by a special delimiter can be rolled up by that common prefix for the purposes of listing. This enables applications to organize and browse their keys hierarchically, much like how you would organize your files into directories in a file system. For example, to extend the dictionary bucket to contain more than only English words, you might form keys by prefixing each word with its language and a delimiter, such as Spanish/logical. Using this naming scheme and the hierarchical listing feature, you could retrieve a list of only Spanish words. You could also browse the top-level list of available languages without having to cycle through all the lexicographically intervening keys.

### **Listing Keys Hierarchically Using a Prefix and Delimiter**

The prefix and delimiter parameters limit the kind of results the List operation returns. The prefix parameter limits results to only those keys that begin with the specified prefix, and the delimiter parameter causes the list to roll up all keys that share a common prefix into a single summary list result.

The purpose of the prefix and delimiter parameters is to help you organize and browse your keys hierarchically. To do this, first choose a delimiter for your bucket, such as slash (/), that does not occur in any of your anticipated key names. Next, construct your key names by concatenating all containing levels of the hierarchy, separating each level with the delimiter.

For example, if you were storing information about cities, you might naturally organize them by continent, then by country, and then by province or state. Because these names do not usually contain punctuation, you might select slash (/) as the delimiter. The following examples use a slash (/) delimiter:

- Europe/Spain/Madrid
- North America/Canada/Quebec/Bordeaux
- North America/USA/Texas/San Antonio
- North America/USA/Texas/Houston

If you stored data for every city in the world in this manner, it would become awkward to manage a flat key namespace. By using Prefix and Delimiter response elements with the list operation, you can use the hierarchy to list your data. For example, to list all of the states in the United States, set Delimiter='/' and Prefix='North America/USA/'. To list all of the provinces in Canada for which you have data, set Delimiter='/' and Prefix='North America/Canada/'.

### **Iterating Through Multipage Results**

Because buckets can contain a virtually unlimited number of objects and keys, the entire results of a list operation can be large. To manage large result sets, the Amazon S3 API supports pagination to break them into multiple responses. Each list keys response returns a page of up to 1,000 keys with an indicator that illustrates whether the response is truncated. Send a series of list keys requests until you have retrieved all of the keys.

## Deleting Objects

Amazon S3 provides several options for deleting objects from your bucket. You can delete one or more objects directly from your S3 bucket. If you want to delete a single object, you can use the Amazon S3 Delete API. If you want to delete multiple objects, you can use the Amazon S3 Multi-Object Delete API, which enables you to delete up to 1,000 objects with a single request. The Multi-Object Delete operation requires a single query string `Delete` parameter to distinguish it from other bucket POST operations.

When deleting objects from a bucket that is not version-enabled, provide only the object key name. However, when deleting objects from a version-enabled bucket, provide the version ID of the object to delete a specific version of the object.

### Deleting Objects from a Version-Enabled Bucket

If your bucket is version-enabled, then multiple versions of the same object can exist in the bucket. When working with version-enabled buckets, the DELETE API enables the following options:

**Specify a nonversioned delete request** Specify only the object's key, not the version ID. In this case, Amazon S3 creates a delete marker and returns its version ID in the response. This makes your object disappear from the bucket.

**Specify a versioned delete request** Specify both the key and a version ID. In this case, the following two outcomes are possible:

- If the version ID maps to a specific object version, then Amazon S3 deletes the specific version of the object.
- If the version ID maps to the delete marker of that object, Amazon S3 deletes the delete marker. This causes the object to reappear in your bucket.

## Performance Optimization

This section discusses Amazon S3 best practices for optimizing performance.

### Request Rate and Performance Considerations

Amazon S3 scales to support high request rates. If your request rate grows steadily, Amazon S3 automatically partitions your buckets as needed to support higher request rates. However, if you expect a rapid increase in the request rate for a bucket to more than 300 PUT/LIST/DELETE requests per second or more than 800 GET requests per second, AWS recommends that you open a support case to prepare for the workload and avoid any temporary limits on your request rate.

If your requests are typically a mix of GET, PUT, DELETE, or GET Bucket (List Objects), choose the appropriate key names for your objects to ensure better performance by providing low-latency access to the Amazon S3 index. It also ensures scalability regardless of the number of requests that you send per second. If the bulk of your workload consists of GET requests, AWS recommends using the Amazon CloudFront content delivery service.



The Amazon S3 best practice guidelines apply only if you are routinely processing 100 or more requests per second. If your typical workload involves only occasional bursts of 100 requests per second and fewer than 800 requests per second, you do not need to follow these recommendations.

## Workloads with Different Request Types

When you are uploading a large number of objects, you might use sequential numbers or date-and-time values as part of the key names. For instance, you might decide to use key names that include a combination of the date and time, as shown in the following example, where the prefix includes a timestamp:

```
demobucket/2018-31-05-16-00-00/order1234234/receipt1.jpg  
demobucket/2018-31-05-16-00-00/order3857422/receipt2.jpg  
demobucket/2018-31-05-16-00-00/order1248473/receipt2.jpg  
demobucket/2018-31-05-16-00-00/order8474937/receipt2.jpg  
demobucket/2018-31-05-16-00-00/order1248473/receipt3.jpg  
...  
demobucket/2018-31-05-16-00-00/order1248473/receipt4.jpg  
demobucket/2018-31-05-16-00-00/order1248473/receipt5.jpg  
demobucket/2018-31-05-16-00-00/order1248473/receipt6.jpg  
demobucket/2018-31-05-16-00-00/order1248473/receipt7.jpg
```

The way these keys are named presents a performance problem. To get a better understanding of this issue, consider the way that Amazon S3 stores key names. Amazon S3 maintains an index of object key names in each AWS Region. These keys are stored in UTF-8 binary ordering across multiple partitions in the index. The key name dictates where (or which partition) the key is stored in. In this case, where you use a sequential prefix such as a timestamp, or an alphabetical sequence, would increase the chances that Amazon S3 targets a specific partition for a large number of your keys, overwhelming the I/O capacity of the partition. However, if you introduce randomness in your key name prefixes, the key names, and therefore the I/O load, distribute across more than one partition.

If you anticipate that your workload will consistently exceed 100 requests per second, avoid sequential key names. If you must use sequential numbers or date-and-time patterns in key names, add a random prefix to the key name. The randomness of the prefix more evenly distributes key names across multiple index partitions.

The guidelines for the key name prefixes also apply to the bucket names. When Amazon S3 stores a key name in the index, it stores the bucket names as part of the key name (`demobucket/object.jpg`).

One way to introduce randomness to key names is to add a hash string as a prefix to the key name. For instance, you can compute an MD5 hash of the character sequence that you plan to assign as the key name. From the hash, select a specific number of characters and add them as the prefix to the key name.



A hashed prefix of three or four characters should be sufficient. AWS strongly recommends using a hexadecimal hash as the prefix.

## GET-Intensive Workloads

If your workload consists mostly of sending GET requests, then in addition to the preceding guidelines, consider using Amazon CloudFront for performance optimization.

Integrating CloudFront with Amazon S3 enables you to deliver content with low latency and a high data transfer rate. You will also send fewer direct requests to Amazon S3, which helps to lower your costs.

Suppose that you have a few objects that are popular. CloudFront fetches those objects from Amazon S3 and caches them. CloudFront can then serve future requests for the objects from its cache, reducing the number of GET requests it sends to Amazon S3.

## Storing Large Attribute Values in Amazon S3

Amazon DynamoDB currently limits the size of each item that you store in a table to 400 KB. If your application needs to store more data in an item than the DynamoDB size limit permits, store the large attributes as an object in Amazon S3. You can then store the Amazon S3 object identifier in your item.

You can also use the object metadata support in Amazon S3 to store the primary key value of the corresponding table item as Amazon S3 object metadata. Doing this provides a link back to the parent item in DynamoDB, which helps with maintenance of the Amazon S3 objects.

### Example 11: Store Large Attribute Values in Amazon S3

Suppose that you have a table that stores product data, such as item price, description, book authors, and dimensions for other products. If you want to store an *image* of each product that was too large to fit in an *item*, use Amazon S3 images as an item in DynamoDB.

When implementing this strategy, remember the following limitations and restrictions:

- DynamoDB does not support transactions that cross Amazon S3 and DynamoDB. Therefore, your application must deal with any failures, which could include cleaning up orphaned Amazon S3 objects.
- Amazon S3 limits the length of object identifiers. You must organize your data in a way that does not generate excessively long object identifiers or violate other Amazon S3 constraints.

# Amazon Elastic File System

*Amazon Elastic File System* (Amazon EFS) provides simple, scalable file storage for use with Amazon EC2. With Amazon EFS, storage capacity is elastic, growing and shrinking automatically as you add and remove files so your applications have the storage they need when they need it. The simple web services interface helps you create and configure file systems quickly and easily. The service manages all of the file storage infrastructure, meaning that you can avoid the complexity of deploying, patching, and maintaining complex file system configurations, such as situations where it must facilitate user uploads or interim results of batch processes. By placing those files in a shared storage layer, it helps you to avoid the introduction of stateful components.

Amazon EFS supports the Network File System versions 4.0 and 4.1 (NFSv4) protocol, so the applications and tools that you use today work seamlessly with Amazon EFS. Multiple Amazon EC2 instances can access an Amazon EFS file system at the same time, providing a common data source for workloads and applications running on more than one instance or server.

The service is highly scalable, highly available, and highly durable. Amazon EFS stores data and metadata across multiple Availability Zones in a region, and it can grow to petabyte scale, drive high levels of throughput, and allow massively parallel access from Amazon EC2 instances to your data.

Amazon EFS provides file system access semantics, such as strong data consistency and file locking. Amazon EFS also allows you to control access to your file systems through Portable Operating System Interface (POSIX) permissions.

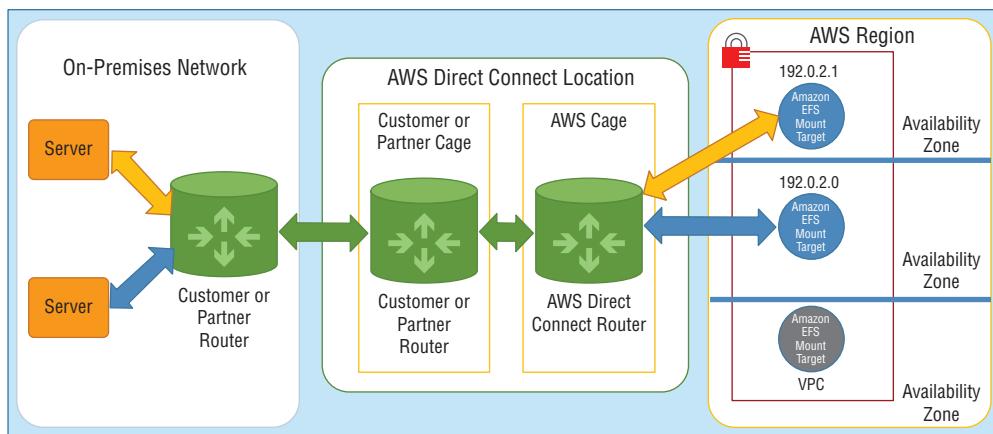
Amazon EFS supports two forms of encryption for file systems: encryption in transit and encryption at rest. You can enable encryption at rest when creating an Amazon EFS file system. If you do, all of your data and metadata is encrypted. You can enable encryption in transit when you mount the file system.

Amazon EFS is designed to provide the throughput, input/output operations per second (IOPS), and low latency needed for a broad range of workloads. With Amazon EFS, throughput and IOPS scale as a file system grows, and file operations are delivered with consistent, low latencies.

## How Amazon EFS Works

Figure 14.19 shows an example of VPC accessing an Amazon EFS file system. In this example, Amazon EC2 instances in the VPC have file systems mounted.

Amazon EFS provides file storage in the AWS Cloud. With Amazon EFS, you can create a file system, mount the file system on an Amazon EC2 instance, and then read and write data to and from your file system. You can mount an Amazon EFS file system in your VPC through the NFSv4 protocol.

**FIGURE 14.19** VPC accessing an Amazon EFS

You can access your Amazon EFS file system concurrently from Amazon EC2 instances in your Amazon VPC so that applications that scale beyond a single connection can access a file system. Amazon EC2 instances running in multiple Availability Zones within the same region can access the file system so that many users can access and share a common data source.

However, there are restrictions. You can mount an Amazon EFS file system on instances in only one VPC at a time. Both the file system and VPC must be in the same AWS Region.

To access your Amazon EFS file system in a VPC, create one or more mount targets in the VPC. A *mount target* provides an IP address for an NFSv4 endpoint at which you can mount an Amazon EFS file system. Mount your file system using its DNS name, which resolves to the IP address of the Amazon EFS mount target in the same Availability Zone as your EC2 instance. You can create one mount target in each Availability Zone in a region. If there are multiple subnets in an Availability Zone in your Amazon VPC, create a mount target in one of the subnets, and all EC2 instances in that Availability Zone share that mount target.

Mount targets themselves are designed to be highly available. When designing your application for both high availability and the ability to failover to other Availability Zones, keep in mind that the IP addresses and DNS for your mount targets in each Availability Zone are static. After mounting the file system via the mount target, use it like any other POSIX-compliant file system.

You can mount your Amazon EFS file systems on your on-premises data center servers when connected to your Amazon VPC with AWS Direct Connect (DX). You can mount your Amazon EFS file systems on on-premises servers to migrate datasets to Amazon EFS, enable cloud-bursting scenarios, or back up your on-premises data to Amazon EFS.

You can mount Amazon EFS file systems on Amazon EC2 instances or on-premises through a DX connection.

## How Amazon EFS Works with AWS Direct Connect

Using an Amazon EFS file system mounted on an on-premises server, you can migrate on-premises data into the AWS Cloud hosted in an Amazon EFS file system. You can also take advantage of bursting. This means that you can move data from your on-premises servers into Amazon EFS, analyze it on a fleet of Amazon EC2 instances in your Amazon VPC, and then store the results permanently in your file system or move the results back to your on-premises server.

Consider the following when using Amazon EFS with DX:

- Your on-premises server must have a Linux-based operating system. AWS recommends Linux kernel version 4.0 or later.
- For the sake of simplicity, AWS recommends mounting an Amazon EFS file system on an on-premises server using a mount target IP address instead of a DNS name.
- AWS Virtual Private Network (AWS VPN) is not supported for accessing an Amazon EFS file system from an on-premises server.

You can use any one of the mount targets in your Amazon VPC as long as the subnet of the mount target is reachable by using the DX connection between your on-premises server and Amazon VPC. To access Amazon EFS from an on-premises server, you must add a rule to your mount target security group to allow inbound traffic to the NFS port (2049) from your on-premises server.

In Amazon EFS, a file system is the primary resource. Each file system has properties such as ID, creation token, creation time, file system size in bytes, number of mount targets created for the file system, and the file system state.

Amazon EFS also supports other resources to configure the primary resource. These include mount targets and tags:

### Mount Target

- To access your file system, create mount targets in your Amazon VPC. Each mount target has the following properties:
  - Mount target ID
  - Subnet ID where it is created
  - File system ID for which it is created
  - IP address at which the file system may be mounted
  - Mount target state

You can use the IP address or the DNS name in your mount command.

### Tags

- To help organize your file systems, assign your own metadata to each of the file systems that you create. Each tag is a key-value pair.

Each file system has a DNS name of the following form:

*file-system-ID.efs.aws-region.amazonaws.com*

You can configure this DNS name in your `mount` command to mount the Amazon EFS file system.

Suppose that you create an `efs-mount-point` subdirectory in your home directory on your EC2 instance or on-premises server. Use the `mount` command to mount the file system. For example, on an Amazon Linux AMI, you can use following `mount` command:

```
$ sudo mount -t nfs -o nfsvers=4.1,rsize=1048576,wszie=1048576,hard,timeo=600, retrans=2,noresvport file-system-DNS-name:/ ~/efs-mount-point
```

You can think of mount targets and tags as *subresources* that do not exist without being associated with a file system.

Amazon EFS provides API operations for you to create and manage these resources. In addition to the create and delete operations for each resource, Amazon EFS also supports a describe operation that enables you to retrieve resource information. The following options are available for creating and managing these resources:

- Use the Amazon EFS console.
- Use the Amazon EFS command line interface (CLI).

You can also manage these resources programmatically as follows:

**Use the AWS SDKs** The AWS SDKs simplify your programming tasks by wrapping the underlying Amazon EFS API. The SDK clients also authenticate your requests by using access keys that you provide.

**Call the Amazon EFS API directly from your application** If you cannot use the SDKs, you can make the Amazon EFS API calls directly from your application. However, if you use this option, you must write the necessary code to authenticate your requests.

## Authentication and Access Control

You must have valid credentials to make Amazon EFS API requests, such as creating a file system. In addition, you must also have permissions to create or access resources. By default, when you use the account root user credentials, you can create and access resources owned by that account. However, AWS does not recommend using account root user credentials. In addition to creating or accessing resources, you must grant permissions to any AWS IAM users and roles you create in your account.

## Data Consistency in Amazon EFS

Amazon EFS provides the open-after-close consistency semantics that applications expect from NFS. In Amazon EFS, write operations are durably stored across Availability Zones when an application performs a synchronous write operation (for example, using the `open` Linux command with the `O_DIRECT` flag, or the `fsync` Linux command) and when an application closes a file.

Amazon EFS provides stronger consistency than open-after-close semantics, depending on the access pattern. Applications that perform synchronous data access and perform non-appending writes have read-after-write consistency for data access.

## Creating an IAM User

Services in AWS, such as Amazon EFS, require that you provide credentials when you access them so that the service can determine whether you have permissions to access its resources. AWS recommends that you do not use the AWS account credentials of your account to make requests. Instead, create an IAM user, and grant that user full access. AWS refers to these users as administrators. You can use the administrator credentials, instead of AWS account credentials, to interact with AWS and perform tasks, such as creating a bucket, creating users, and granting them permissions.

For all operations, such as creating a file system and creating tags, a user must have IAM permissions for the corresponding API action and resource. You can perform any Amazon EFS operations using the AWS account credentials of your account. However, using AWS account credentials is not considered a best practice. If you create IAM users in your account, you can give them permissions for Amazon EFS actions with user policies. Additionally, you can use roles to grant cross-account permissions.

## Creating Resources for Amazon EFS

Amazon EFS provides elastic, shared file storage that is POSIX-compliant. The file system that you create supports concurrent read and write access from multiple Amazon EC2 instances, and it is accessible from all of the Availability Zones in the AWS Region where it is created. You can mount an Amazon EFS file system on EC2 instances in your Amazon VPC through the Network File System versions 4.0 and 4.1 protocol (NFSv4).

As an example, suppose that you have one or more EC2 instances launched in your Amazon VPC. You want to create and use a file system on these instances. To use Amazon EFS file systems in the VPC, follow these steps:

1. **Create an Amazon EFS file system.** When creating a file system, AWS recommends that you consider using the Name tag, because its value appears in the console, making the file easier to identify. You can also add other optional tags to the file system.
2. **Create mount targets for the file system.** To access the file system in your Amazon VPC and mount the file system to your Amazon EC2 instance, you must create mount targets in the VPC subnets.
3. **Create security groups.** Both an Amazon EC2 instance and mount target must have associated security groups. These security groups act as a virtual firewall that controls the traffic between them. You can use the security group that you associated with the mount target to control inbound traffic to your file system by adding an inbound rule to the mount target security group that allows access from a specific EC2 instance. Then you can mount the file system only on that EC2 instance.

## Creating a File System

You can use the Amazon EFS console, or the AWS CLI, to create a file system. You can also use the AWS SDKs to create file systems programmatically.

## Using File Systems

Amazon EFS presents a standard file system interface that supports semantics for full file system access. Using NFSv4.1, you can mount your Amazon EFS file system on any Amazon EC2 Linux-based instance. Once mounted, you can work with the files and directories as you would with a local file system. You can also use AWS DataSync to copy files from any file system to Amazon EFS.

After you create the file system and mount it on your EC2 instance, you should be aware of several rules to use it effectively. For example, when you first create the file system, there is only *one root directory* at /. By default, only the *root user* (UID 0) has read-write-execute permissions. For other users to modify the file system, the root user must explicitly grant them access.

### NFS-Level Users, Groups, and Permissions

Amazon EFS file system objects have a Unix-style mode associated with them. This value defines the permissions for performing actions on that object, and users familiar with Unix-style systems can understand how Amazon EFS manages these permissions.

Further, on Unix-style systems, users and groups are mapped to numeric identifiers, which Amazon EFS uses to represent file ownership. A single owner and a single group own file system objects, such as files or directories on Amazon EFS. Amazon EFS uses these numeric IDs to check permissions when a user attempts to access a file system object.

### User and Group ID Permissions on Files and Directories within a File System

Files and directories in an Amazon EFS file system support standard Unix-style read/write/execute permissions based on the user ID and group ID asserted by the mounting NFSv4.1 client. When a user tries to access files and directories, Amazon EFS checks their user ID and group IDs to determine whether the user has permission to access the objects. Amazon EFS also uses these IDs as the owner and group owner for new files and directories that the user creates. Amazon EFS does not inspect user or group names—it uses only the numeric identifiers.

When you create a user on an EC2 instance, you can assign any numeric UID and GID to the user. The numeric user IDs are set in the /etc/passwd file on Linux systems. The numeric group IDs are in the /etc/group file. These files define the mappings between names and IDs. Outside of the EC2 instance, Amazon EFS does not perform any authentication of these IDs, including the root ID of 0.

If a user accesses an Amazon EFS file system from two different EC2 instances, depending on whether the UID for the user is the same or different on those instances, you may observe different behavior. If the user IDs are the same on both EC2 instances, Amazon EFS considers them the same user, regardless of the EC2 instance they use. The user experience when accessing the file system is the same from both EC2 instances. If the user IDs are not the same on both EC2 instances, Amazon EFS considers them to be different users, and the user experience will not be the same when accessing the Amazon EFS file system from

the two different EC2 instances. If two different users on different EC2 instances share an ID, Amazon EFS considers them the same user.

## Deleting an Amazon EFS File System

File system deletion is a permanent action that destroys the file system and any data in it. Any data that you delete from a file system is gone, and you cannot restore the data.



Always unmount a file system before you delete it.

## Managing Access to Encrypted File Systems

Using Amazon EFS, you can create encrypted file systems. Amazon EFS supports two forms of encryption for file systems: encryption in transit and encryption at rest. Any key management that you must perform is related only to encryption at rest. Amazon EFS automatically manages the keys for encryption in transit. If you create a file system that uses encryption at rest, data and metadata are encrypted at rest.

Amazon EFS uses AWS KMS for key management. When you create a file system using encryption at rest, specify a customer master key (CMK). The CMK can be `aws/elasticfilesystem` (the AWS managed CMK for Amazon EFS), or it can be a CMK that you manage. File data, the contents of your files, is encrypted at rest using the CMK that you specified when you created your file system.

The AWS managed CMK for your file system is used as the master key for the metadata in your file system; for instance, file names, directory names, and directory contents. You are responsible for the CMK used to encrypt your file data (the contents of your files) at rest. Moreover, you are responsible for the management of who has access to your CMKs and the contents of your encrypted file systems. IAM policies and AWS KMS control these permissions. IAM policies control a user's access to Amazon EFS API actions. AWS KMS key policies control a user's access to the CMK that you specified when the file system was created.

As a key administrator, you can both import external keys and modify keys by enabling, disabling, or deleting them. The state of the CMK that you specified (when you created the file system with encryption at rest) affects access to its contents. To provide users access to the contents of an encrypted at rest file system, the CMK must be in the enabled state.

## Amazon EFS Performance

Amazon EFS file systems are spread across an unconstrained number of storage servers, allowing file systems to expand elastically to petabyte scale. The distribution also allows them to support massively parallel access from Amazon EC2 instances to your data. Because of this distributed design, Amazon EFS avoids the bottlenecks and limitations inherent to conventional file servers.

This distributed data storage design means that multithreaded applications and applications that concurrently access data from multiple Amazon EC2 instances can drive substantial levels of aggregate throughput and IOPS. Analytics and big data workloads, media processing workflows, content management, and web serving are examples of these applications.

Additionally, Amazon EFS data is distributed across multiple Availability Zones, providing a high level of availability and durability.

## Performance Modes

To support a wide variety of cloud storage workloads, Amazon EFS offers two performance modes: General Purpose and Max I/O. At the time that you create your file system, you select a file system's performance mode. There are no additional charges associated with the two performance modes. Your Amazon EFS file system is billed and metered the same, irrespective of the performance mode chosen. You cannot change an Amazon EFS file system's performance mode after you have created the file system.

**General Purpose performance mode** AWS recommends the General Purpose performance mode for the majority of your Amazon EFS file systems. General Purpose is ideal for latency-sensitive use cases, such as web serving environments, content management systems, home directories, and general file serving. If you do not choose a performance mode when you create your file system, Amazon EFS selects the General Purpose mode for you by default.

**Max I/O performance mode** File systems in the Max I/O mode can scale to higher levels of aggregate throughput and operations per second with a trade-off of slightly higher latencies for file operations. Highly parallelized applications and workloads, such as big data analysis, media processing, and genomics analysis can benefit from this mode.

## Throughput Scaling in Amazon EFS

Throughput on Amazon EFS scales as a file system grows. Because file-based workloads are typically spiky, driving high levels of throughput for short periods of time and low levels of throughput the rest of the time, Amazon EFS is designed to burst to high throughput levels for periods of time.

All file systems, regardless of size, can burst to 100 MB/s of throughput, and those larger than 1 TB can burst to 100 MB/s per TB of data stored in the file system. For example, a 10-TB file system can burst to 1,000 MB/s of throughput ( $10\text{ TB} \times 100\text{ MB/s/TB}$ ). The portion of time a file system can burst is determined by its size, and the bursting model is designed so that typical file system workloads will be able to burst virtually any time they need to.

Amazon EFS uses a credit system to determine when file systems can burst. Each file system earns credits over time at a baseline rate that is determined by the size of the file system, and it uses credits whenever it reads or writes data. The baseline rate is 50 MB/s per TB of storage (equivalently, 50 KB/s per GB of storage).

Accumulated burst credits give the file system permission to drive throughput above its baseline rate. A file system can drive throughput continuously at its baseline rate. Whenever the file system is inactive or when it is driving throughput below its baseline rate, the file system accumulates burst credits.

## Summary

In this chapter, stateless applications are defined as those that do not require knowledge of previous individual interactions and do not store session information locally. Stateless application design is beneficial because it reduces the risk of loss of session information or critical data. It also improves user experience by reducing the chances that context-specific data is lost if a resource containing session information becomes unavailable. To accomplish this, AWS customers can use Amazon DynamoDB, Amazon ElastiCache, Amazon Simple Storage Service (Amazon S3), and Amazon Elastic File System (Amazon EFS).

DynamoDB is a fast and flexible NoSQL database service that is used by applications that require consistent, single-digit millisecond latency at any scale. In stateless application design, you can use DynamoDB to store and rapidly retrieve session information. This separates session information from application resources responsible for processing user interactions. For example, a web application can use DynamoDB to store user shopping carts. If an application server becomes unavailable, the users accessing the application do not experience a loss of service.

To further improve speed of access, DynamoDB supports global secondary indexes and local secondary indexes. A secondary index contains a subset of attributes from a table and uses an alternate key to support custom queries. A local secondary index has the same partition key as a table but uses a different sort key. A global secondary index has different partition and sort keys.

DynamoDB uses read and write capacity units to determine cost. A single read capacity unit represents one strongly consistent read per second (or two eventually consistent reads per second) for items up to 4 KB in size. A single write capacity unit represents one write per second for items up to 1 KB in size. Items larger than these values consume additional read or write capacity.

ElastiCache enables you to quickly deploy, manage, and scale distributed in-memory cache environments. With ElastiCache, you can store application state information in a shared location by using an in-memory key-value store. Caches can be created using either Memcached or Redis caching engines. Read and write operations to a backend database can be time-consuming. Thus, ElastiCache is especially effective as a caching layer for heavy-use applications that require rapid access to backend data. You can also use ElastiCache to store HTTP sessions, further improving the performance of your applications.

ElastiCache offers various scalability configurations that improve access times and availability. For example, read-heavy applications can use additional cache cluster nodes to respond to queries. Should there be an increase in demand, additional cluster nodes can be scaled out quickly.

There are some differences between the available caching engines. AWS recommends that you use Memcached for simple data models that may require scaling and partitioning/sharding. Redis is recommended for more complex data types, persistent key stores, read-replication, and publish/subscribe operations.

In certain situations, storing state information can involve larger file operations (such as file uploads and batch processes). Amazon S3 can support millions of operations per second on trillions of objects through a simple web service. Through simple integration, developers can take advantage of the massive scale of object storage.

Amazon S3 stores objects in buckets, which are addressable using unique URLs (such as <http://johnstiles.s3.amazonaws.com/>). Buckets enable you to group similar objects and configure access control policies for internal and external users. Buckets also serve as the unit of aggregation for usage reporting. There is no limit to the number of objects that can be stored in a bucket, and there is no performance difference between using one or multiple buckets for your web application. The decision to use one or more buckets is often a consideration of access control.

Amazon S3 buckets support versioning and lifecycle configurations to maintain the integrity of objects and reduce cost. Versioning ensures that any time an object is modified and uploaded to a bucket, it is saved as a new version. Authorized users can access previous versions of objects at any time. In versioned buckets, a delete operation places a marker on the object (without deleting prior versions). Conversely, you must use a separate operation to fully remove an object from a versioned bucket. Use lifecycle configurations to reduce cost by automatically moving infrequently accessed objects to lower-cost storage tiers.

Amazon EFS provides simple, scalable file storage for use with multiple concurrent Amazon EC2 instances or on-premises systems. In stateless design, having a shared block storage system removes the risk of loss of data in situations where one or more instances become unavailable.

## Exam Essentials

**Understand block storage vs. object storage.** The difference between block storage and object storage is the fundamental unit of storage. With block storage, each file saved to the drive is composed of blocks of a specific size. With object storage, each file is saved as a single object regardless of size.

**Understand when to use Amazon Simple Storage Service and when to use Amazon Elastic Block Storage or Amazon Elastic File System.** This is an architectural decision based on the type of data that you are storing and the rate at which you intend to update that data. Amazon Simple Storage Service (Amazon S3) can hold any type of data, but Amazon S3 would not be a good choice for a database or any rapidly changing data types.

**Understand Amazon S3 versioning.** Once Amazon S3 versioning is enabled, you cannot disable the feature—you can only suspend it. Also, when versioning is activated, items that are deleted are assigned a delete marker and are not accessible. The deleted objects are still in Amazon S3, and you will continue to incur charges for storing them.

**Know how to control access to Amazon S3 objects.** IAM policies specify which actions are allowed or denied on specific AWS resources. Amazon S3 bucket policies are attached only to Amazon S3 buckets. Amazon S3 bucket policies specify which actions are allowed or denied for principals on the bucket to which the bucket policy is attached.

**Know how to create or select a proper primary key for an Amazon DynamoDB table.** DynamoDB stores data as groups of attributes, known as *items*. Items are similar to rows or records in other database systems. DynamoDB stores and retrieves each item based on the primary key value, which must be unique. Items are distributed across 10 GB storage units, called *partitions* (physical storage internal to DynamoDB). Each table has one or more partitions. DynamoDB uses the partition key value as an input to an internal hash function. The output from the hash function determines the partition in which the item is stored. The hash value of its partition key determines the location of each item. All items with the same partition key are stored together. Composite partition keys are ordered by the sort key value. If the collection size grows bigger than 10 GB, DynamoDB splits partitions by sort key.

**Understand how to configure the read capacity units and write capacity units properly for your tables.** When you create a table or index in DynamoDB, you must specify your capacity requirements for read and write activity. By defining your throughput capacity in advance, DynamoDB can reserve the necessary resources to meet the read and write activity your application requires while ensuring consistent, low-latency performance.

One read capacity unit (RCU) represents one strongly consistent read per second, or two eventually consistent reads per second, for an item up to 4 KB in size. If you need to read an item that is larger than 4 KB, DynamoDB must consume additional RCUs. The total number of RCUs required depends on the item size and whether you want an eventually consistent or strongly consistent read.

One write capacity unit (WCU) represents one write per second for an item up to 1 KB in size. If you need to write an item that is larger than 1 KB, DynamoDB must consume additional WCUs. The total number of WCUs required depends on the item size.

**Understand the use cases for DynamoDB streams.** A DynamoDB stream is an ordered flow of information about changes to items in an DynamoDB table. When you enable a stream on a table, DynamoDB captures information about every modification to data items in the table. Whenever an application creates, updates, or deletes items in the table, DynamoDB Streams writes a stream record with the primary key attributes of the items that were modified. A stream record contains information about a data modification to a single item in a DynamoDB table. You can configure the stream so that the stream records capture additional information, such as the before and after images of modified items.

**Know what secondary indexes are and when to use a local secondary index versus a global secondary index and the differences between the two.** A global secondary index is an index with a partition key and a sort key that can be different from those on the base table.

A global secondary index is considered *global* because queries on the index can span all of the data in the base table, across all partitions. A local secondary index is an index that has the same partition key as the base table, but a different sort key. A local secondary index is *local* in the sense that every partition of a local secondary index is scoped to a base table partition that has the same partition key value.

**Know the operations that can be performed using the DynamoDB API.** Know the more common DynamoDB API operations: CreateTable, UpdateTable, Query, Scan, PutItem, GetItem, UpdateItem, DeleteItem, BatchGetItem, and BatchWriteItem. Understand the purpose of each operation and be familiar with some of the parameters and limitations for the batch operations.

**Be familiar with handling errors when using DynamoDB.** Understand the differences between 400 error codes and 500 error codes and how to handle both classes of errors. Also, understand which techniques to use to mitigate the different errors. In addition, you should understand what causes a ProvisionedThroughputExceededException error and what you can do to resolve the issue.

**Understand how to configure your Amazon S3 bucket to serve as a static website.** To host a static website, you configure an Amazon S3 bucket for website hosting and then upload your website content to the bucket. This bucket must have public read access. It is intentional that everyone has read access to this bucket. The website is then available at the AWS Region specific website endpoint of the bucket.

**Be familiar with the Amazon S3 API operations.** Be familiar with the API operations, such as PUT, GET, SELECT, and DELETE. Understand how having versioning enabled affects the behavior of the DELETE operation. You should also be familiar with the situations that require a multipart upload and how to use the associated API.

**Understand the differences among the different Amazon S3 storage classes.** The storage classes are Standard, Infrequent Access (IA), Glacier, and Reduced Redundancy. Understand the differences and why you might choose one storage class over the other and knowing the consequences of those choices.

**Know how to use Amazon ElastiCache.** Improve the performance of your application by deploying ElastiCache clusters as a part of your application and offloading read requests for frequently accessed data. Use the lazy loading caching strategy in your solution to first check the cache for your query results before checking the database.

**Understand when to choose one specific cache engine over another.** ElastiCache provides two open source caching engines. You are responsible for choosing the engine that meets your requirements. Use Redis when you must persist and restore your data, you need multiple replicas of your data, or you are seeking advanced features and functionality, such as sort and rank or leaderboards. Redis supports these features natively. Alternatively, you can use Memcached when you need a simpler, in-memory object store that can be easily partitioned and horizontally scaled.

# Resources to Review

AWS Database Blog:

<https://aws.amazon.com/blogs/database/>

Amazon DynamoDB Blog:

<https://aws.amazon.com/blogs/aws/tag/amazon-dynamo-db/>

Best Practices for Amazon DynamoDB:

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-practices.html>

Amazon DynamoDB Read Consistency:

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html>

Amazon ElastiCache for Redis User's Guide:

<https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/WhatIs.html>

Amazon ElastiCache Tutorials and Videos:

<https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/Tutorials.html>

Performance at Scale with Amazon ElastiCache (Whitepaper):

<https://d0.awsstatic.com/whitepapers/performance-at-scale-with-amazon-elasticsearch.pdf>

Amazon ElastiCache FAQs:

<https://aws.amazon.com/elasticache/faqs/>

Amazon VPCs and ElastiCache Security:

<https://docs.aws.amazon.com/AmazonElastiCache/latest/mem-ug/VPCs.html>

Amazon Simple Storage Service Developer Guide:

<https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html>

Getting Started with Amazon Simple Storage Service:

<https://docs.aws.amazon.com/AmazonS3/latest/gsg/GetStartedWithS3.html>

AWS Storage Services Overview (Whitepaper):

<https://d1.awsstatic.com/whitepapers/Storage-AWS%20Storage%20Services%20Whitepaper-v9.pdf>

Projects on AWS: Host a Static Website:

[https://aws.amazon.com/getting-started/projects/host-static-website/?trk=gs\\_card](https://aws.amazon.com/getting-started/projects/host-static-website/?trk=gs_card)

Amazon S3 Frequently Asked Questions:

<https://aws.amazon.com/s3/faqs/?nc=sn&loc=6>

Deep Dive on Amazon S3 & Amazon Glacier Storage Management (Video):

<https://www.youtube.com/watch?v=SUWqDOnXeDw>

What is Cloud Object Storage?

<https://aws.amazon.com/what-is-cloud-object-storage/>

When to Choose Amazon EFS:

<https://aws.amazon.com/efs/when-to-choose-efs/>

Amazon EFS FAQs:

<https://aws.amazon.com/efs/faq/>

Amazon Elastic File System: Choosing Between the Different Throughput and Performance Modes (Whitepaper):

[https://d1.awsstatic.com/whitepapers/Storage/amazon\\_efs\\_choosing\\_between\\_different\\_performance\\_and\\_throughput.pdf](https://d1.awsstatic.com/whitepapers/Storage/amazon_efs_choosing_between_different_performance_and_throughput.pdf)

Deep Dive on Amazon EFS (Video):

<https://www.youtube.com/watch?v=LWiAwIa2H7c&feature=youtu.be>

## Exercises

### EXERCISE 14.1

#### Create an Amazon ElastiCache Cluster Running Memcached

In this exercise, you will create an Amazon ElastiCache cluster using the Memcached engine.

1. Sign in to the AWS Management Console, and open the ElastiCache console at <https://console.aws.amazon.com/elasticache/>.
2. To create a new ElastiCache cluster, begin the launch and configuration process.
3. For **Cluster engine**, choose **Memcached** and configure the cluster name, number of nodes, and node type.
4. (Optional) Configure the security group and maintenance window as needed.
5. Review the cluster configuration, and begin provisioning the cluster. Connect to the cluster with any Memcached client by using the DNS name of the cluster.

You have now created your first ElastiCache cluster.

**EXERCISE 14.2****Expand the Size of a Memcached Cluster**

In this exercise, you will expand the size of an existing Amazon ElastiCache Memcached cluster.

1. Launch a Memcached cluster by following the steps in the previous exercise.
2. Navigate to the Amazon ElastiCache dashboard, and view the configuration of your existing cluster.
3. View the list of nodes that are currently being used, and then add one new node by increasing the number of nodes.
4. Apply the changes to the configuration, and wait for the new node to finish provisioning.
5. Confirm that the new node has been provisioned, and connect to the node using a Memcached client.

You have horizontally scaled an existing ElastiCache cluster by adding a new cache node.

---

**EXERCISE 14.3****Create and Attach an Amazon EFS Volume**

In this exercise, you will create a new Amazon EFS volume and attach it to a running instance.

1. While signed in to the AWS Management Console, open the Amazon EC2 console at <https://console.aws.amazon.com/ec2>.  
If you don't see a running Linux instance, launch a new instance.
2. Open the Amazon EFS service dashboard. Choose **Create File System**.
3. Select the Amazon VPC where your Linux instance is running.
4. Accept the default mount targets, and make a note of the security group ID assigned to the targets.
5. Choose any settings, and then create the file system.
6. Assign the same default security group used by the file system to your Linux instance.

---

*(continued)*

**EXERCISE 14.3 (*continued*)**

7. Log in to the console of the Linux instance, and install the NFS client on the Amazon EC2 instance. For Amazon Linux, use the following command:

```
sudo yum install -y nfs-utils
```

8. Create a new directory on your Amazon EC2 instances, such as awsdev: sudo mkdir awsdev.

9. Mount the file system using the DNS name:

```
sudo mount -t nfs4 -o nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=600, retrans=2 fs-12341234.efs.region-1.amazonaws.com:/ awsdev
```

You have mounted the Amazon EFS volume to the instance.

---

**EXERCISE 14.4****Create and Upload to an Amazon S3 Bucket**

In this exercise, you will create an Amazon S3 bucket and upload and publish files of the bucket.

1. While signed in to the AWS Management Console, open the Amazon S3 console at <https://console.aws.amazon.com/s3/>.
2. Choose **Create bucket**.
3. On the **Name and region** page, enter a globally unique name for your bucket and select the appropriate AWS Region. Accept all of the remaining default settings.
4. Choose **Create the bucket**.
5. Select your bucket.
6. Upload data files to the new Amazon S3 bucket:
  - a. Choose **Upload**.
  - b. In the **Upload - Select Files** wizard, choose **Add Files** and choose a file that you want to share publicly.
  - c. Choose **Next**.
7. To make the file available to the general public, on the **Set Permissions** page, under **Manage Public Permissions**, grant everyone read access to the object.
8. Review and upload the file.
9. Select the object name to go to the properties screen. Select and open the URL for the file in a new browser window.

You should see your file in the S3 bucket.

---

**EXERCISE 14.5****Create an Amazon DynamoDB Table**

In this exercise, you will create an DynamoDB table.

1. While signed in to the AWS Management Console, open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose **Create Table** and then do the following:
  - a. In **Table**, enter the table name.
  - b. For **Primary key**, in the **Partition** field, type **Id**.
  - c. Set the **data type** to **String**.
3. Retain all the remaining default settings and choose **Create**.

You have created a DynamoDB table.

---

**EXERCISE 14.6****Enable Amazon S3 Versioning**

In this exercise, you will enable Amazon S3 versioning, which prevents objects from being accidentally deleted or overwritten.

1. While signed in to the AWS Management Console, open the Amazon S3 console at <https://console.aws.amazon.com/s3/>.
2. In the **Bucket name** list, choose the name of the bucket for which you want to enable versioning.

If you don't have a bucket, follow the steps in Exercise 14.4 to create a new bucket.
3. Choose **Properties**.
4. Choose **Versioning**.
5. Choose **Enable versioning**, and then choose **Save**.

Your bucket is now versioning enabled.

---

**EXERCISE 14.7****Create an Amazon DynamoDB Global Table**

In this exercise, you will create a DynamoDB global table.

1. While signed in to the AWS Management Console, open the Amazon S3 console at <https://console.aws.amazon.com/dynamodb>.
2. Choose a region for the source table for your DynamoDB global table.
3. In the navigation pane on the left side of the console, choose **Create Table** and then do the following:
  - a. For **Table name**, type **Tables**.
  - b. For **Primary key**, choose an appropriate primary key. Choose **Add sort key**, and type an appropriate sort key. The data type of both the partition key and the sort key should be strings.
4. To create the table, choose **Create**.

This table will serve as the first replica table in a new global table, and it will be the prototype for other replica tables that you add later.

5. Select the **Global Tables** tab, and then choose **Enable streams**. Leave the **View type** at its default value (**New** and **old** images).
6. Choose **Add region**, and then choose another region where you want to deploy another replica table. In this case, choose **US West (Oregon)** and then choose **Continue**. This will start the table creation process in US West (Oregon).

The console will check to ensure that there is no table with the same name in the selected region. (If a table with the same name does exist, then you must delete the existing table before you can create a new replica table in that region.)

The **Global Table** tab for the selected table (and for any other replica tables) will show that the table is replicated in multiple regions.

7. Add another region so that your global table is replicated and synchronized across the United States and Europe. To do this, repeat step 6, but this time specify **EU (Frankfurt)** instead of US West (Oregon).

You have created a DynamoDB global table.

---

**EXERCISE 14.8****Enable Cross-Region Replication**

In this exercise, you will enable cross-region replication of the contents of the original bucket to a new bucket in a different region.

1. While signed into the AWS Management Console, open the Amazon S3 console at <https://console.aws.amazon.com/s3/>.
2. Create a new bucket in a different region from the bucket that you created in Exercise 14.4. Enable versioning on the new bucket (see Exercise 14.6).
3. Choose **Management**, choose **Replication**, and then choose **Add rule**.
4. In the **Replication rule** wizard, under **Set Source**, choose **Entire Bucket**.
5. Choose **Next**.
6. On the **Set destination** page, under **Destination bucket**, choose your newly-created bucket.
7. Choose the storage class for the target bucket. Under **Options**, select **Change the storage class for the replicated objects**. Select a storage class.
8. Choose **Next**.
9. For IAM, on the **Configure options** page, under **Select role**, choose **Create new role**.
10. Choose **Next**.
11. Choose **Save**.
12. Load a new object in the source bucket.

The object appears in the target bucket.

You have enabled cross-region replication, which can be used for compliance and disaster recovery.

**EXERCISE 14.9****Create an Amazon DynamoDB Backup Table**

In this exercise, you will create a DynamoDB table backup.

1. While signed into the AWS Management Console, open the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. Choose one of your existing tables. If there are no tables, follow the steps in Exercise 14.7 to create a new table.

---

*(continued)*

**EXERCISE 14.9 (*continued*)**

3. On the **Backups** tab, choose **Create Backup**.
4. Type a name for the backup name of the table you are backing. Then choose **Create** to create the backup.

While the backup is being created, the backup status is set to Creating. After the backup is finalized, the backup status changes to Available.

You have created a backup of a DynamoDB table.

---

**EXERCISE 14.10****Restoring an Amazon DynamoDB Table from a Backup**

In this exercise, you will restore a DynamoDB table by using the backup created in the previous exercise.

1. While signed in to the AWS Management Console, navigate to the DynamoDB console at <https://console.aws.amazon.com/dynamodb/>.
2. In the navigation pane on the left side of the console, choose **Backups**.
3. In the list of backups, choose the backup that you created in the previous step.
4. Choose **Restore Backup**.
5. Type a table name as the new table name. Confirm the backup name and other backup details. Then choose **Restore table** to start the restore process.

The table that is being restored is shown with the status Creating. After the restore process is finished, the status of your new table changes to Active.

You have performed the restoration of a DynamoDB table from a backup.

---

# Review Questions

1. Which of the following is the maximum Amazon DynamoDB item size limit?
  - A. 512 KB
  - B. 400 KB
  - C. 4 KB
  - D. 1,024 KB
2. Which of the following is true when using Amazon Simple Storage Service (Amazon S3)?
  - A. Versioning is enabled on a bucket by default.
  - B. The largest size of an object in an Amazon S3 bucket is 5 GB.
  - C. Bucket names must be globally unique.
  - D. Bucket names can be changed after they are created.
3. Which of the following is *not* a deciding factor when choosing an AWS Region for your bucket?
  - A. Latency
  - B. Storage class
  - C. Cost
  - D. Regulatory requirements
4. Which of the following features can you use to protect your data at rest within Amazon DynamoDB?
  - A. Fine-grained access controls
  - B. Transport Layer Security (TLS) connections
  - C. Server-side encryption provided by the DynamoDB service
  - D. Client-side encryption
5. You store your company's critical data in Amazon Simple Storage Service (Amazon S3). The data must be protected against accidental deletions or overwrites. How can this be achieved?
  - A. Use a lifecycle policy to move the data to Amazon S3 Glacier.
  - B. Enable MFA Delete on the bucket.
  - C. Use a path-style URL.
  - D. Enable versioning on the bucket.
6. How does Amazon Simple Storage Service (Amazon S3) object storage differ from block and file storage? (Select TWO.)
  - A. Amazon S3 stores data in fixed blocks.
  - B. Objects can be any size.
  - C. Objects are stored in buckets.
  - D. Objects contain both data and metadata.

7. What is the lifetime of data in an Amazon DynamoDB stream?
  - A. 14 days
  - B. 12 hours
  - C. 24 hours
  - D. 4 days
8. How many times does each stream record in Amazon DynamoDB Streams appear in the stream?
  - A. Twice
  - B. Once
  - C. Three times
  - D. This value can be configured.
9. Versioning is a means of keeping multiple variants of an object in the same bucket. You can use versioning to preserve, retrieve, and restore every version of every object stored in your Amazon S3 bucket. With versioning, you can easily recover from both unintended user actions and application failures. Which of the following is *not* a versioning state of a bucket?
  - A. Versioning paused
  - B. Versioning disabled
  - C. Versioning suspended
  - D. Versioning enabled
10. Your team has built an application as a document management system that maintains metadata on millions of documents in a DynamoDB table. When a document is retrieved, you want to display the metadata beside the document. Which DynamoDB operation can you use to retrieve metadata attributes from a table?
  - A. QueryTable
  - B. UpdateTable
  - C. Search
  - D. Scan
11. Which of the following objects are good candidates to store in a cache? (Select THREE.)
  - A. Session state
  - B. Shopping cart
  - C. Product catalog
  - D. Bank account balance
12. Which of the following cache engines does Amazon ElastiCache support? (Select TWO.)
  - A. Redis
  - B. MySQL
  - C. Couchbase
  - D. Memcached

- 13.** How many nodes can you add to an Amazon ElastiCache cluster that is running Redis?
- A.** 100
  - B.** 5
  - C.** 20
  - D.** 1
- 14.** What feature does Amazon ElastiCache provide?
- A.** A highly available and fast indexing service for querying
  - B.** An Amazon Elastic Compute Cloud (Amazon EC2) instance with a large amount of memory and CPU
  - C.** A managed in-memory caching service
  - D.** An Amazon EC2 instance with Redis and Memcached already installed
- 15.** When designing a highly available web solution using stateless web servers, which services are suitable for storing session-state data? (Select THREE.)
- A.** Amazon CloudFront
  - B.** Amazon DynamoDB
  - C.** Amazon CloudWatch
  - D.** Amazon Elastic File System (Amazon EFS)
  - E.** Amazon ElastiCache
  - F.** Amazon Simple Queue Service (Amazon SQS)
- 16.** Which AWS database service is best suited for nonrelational databases?
- A.** Amazon Simple Storage Service Glacier (Amazon S3 Glacier)
  - B.** Amazon Relational Database Service (Amazon RDS)
  - C.** Amazon DynamoDB
  - D.** Amazon Redshift
- 17.** Which of the following statements about Amazon DynamoDB table is true?
- A.** Only one local secondary index is allowed per table.
  - B.** You can create global secondary indexes only when you are creating the table.
  - C.** You can have only one global secondary index.
  - D.** You can create local secondary indexes only when you are creating the table.



# Chapter 15



AWS® Certified Developer Official Study Guide  
By Nick Alteen, Jennifer Fisher, Casey Gerena, Wes Gruver, Asim Jalis,  
Heiward Osman, Marife Pagan, Santosh Patlolla and Michael Roth  
Copyright © 2019 by Amazon Web Services, Inc.

# Monitoring and Troubleshooting

---

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

## Domain 5: Monitoring and Troubleshooting

### ✓ 5.1 Write code that can be monitored.

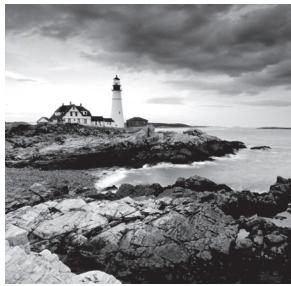
Content may include the following:

- Monitoring basics
- Using Amazon CloudWatch
- Using AWS CloudTrail

### ✓ 5.2 Perform root cause analysis on faults found in testing or production.

Content may include the following:

- Using AWS X-Ray to troubleshoot application issues



# Introduction to Monitoring and Troubleshooting

Monitoring the applications and services you build is vital to the success of any information technology (IT) organization. With the AWS Cloud, you can leverage monitoring resources to drive business decisions such as what resources to create, improve, optimize, and secure.

Traditional approaches to monitoring do not scale for cloud architectures. Large systems can be difficult to set up, configure, and scale. These efforts are compounded by the trend away from monolithic installations toward service-oriented architecture (SOA), microservices, and serverless architectures. Monitoring modern IT systems is proportionally difficult. When working on a monolithic application, you can add logging statements and troubleshoot with breakpoints. However, applications today are spread across multiple systems over large networks that make it difficult to track the health of systems and react to issues. For example, using logging statements to monitor execution time and error rates of AWS Lambda functions can become difficult as your infrastructure grows and spreads across multiple AWS Regions.

The AWS Cloud provides fully managed services to help you implement monitoring solutions that are reliable, scalable, and secure. AWS offers services to help you monitor, log, and analyze your applications and infrastructure. In this section, you explore Amazon CloudWatch, AWS CloudTrail, and AWS X-Ray. Figure 15.1 shows the AWS monitoring services available.

**FIGURE 15.1** Various monitoring services on AWS



## Monitoring Basics

Before you explore these services, consider why they are essential. As a developer, you are designing systems to provide IT or business solutions to a customer. Success is measured by the effective application of software to business objectives. What are some of the metrics that you must track over time to ensure that these objectives are being met?

### Choosing Metrics

AWS takes the approach of “working backward” from the customer. You can accomplish this by starting with the customer and tracing the underlying components that affect the customer’s experience. This provides a foundation for identifying which metrics to monitor, as they correlate directly to the customer experience. Frequently, the top characteristics that directly affect the customer experience are performance and cost. Changes to either have a direct impact on how customers perceive the software they use.

Deciding which metrics to monitor requires that you answer several crucial questions.

#### Performance and Cost

**Question:** Are my customers having a good experience with the services or systems that that I provide?

The phrase *good experience* can be broken down into measurable metrics, such as request latency, time to first byte, error rates, and more. Metrics, such as instance CPU utilization or network bytes in/out, however, may not be representative of the customer experience.

It is good practice to measure any metric that directly affects customers using your software or system. The second question to ask is: “What is the overall cost of my system?” Increases in performance often correlate directly to increases in cost. With unlimited money, it would be easy to design a system that scales infinitely in response to customer usage. However, this is never a reality. Instead, you need to measure the performance of your system to determine what is acceptable performance based on the usage at any point in time. This is the case when metrics that are not customer-facing often take precedence.

#### Trends

**Question:** How can I use monitoring to predict changes in customer demand?

With the agility and elasticity of the AWS Cloud, this can be especially useful. Monitoring and measuring customer demand over time allows you to scale your infrastructure predictively to meet changes in customer demand without having to purchase more resources than are necessary. For example, suppose that you have a web application that runs on three Amazon Elastic Compute Cloud (Amazon EC2) instances during the day. In the evenings, demand increases significantly for several hours before decreasing again late at night. On weekends, your application sees almost no traffic. With historical information obtained through monitoring, you can design your application to scale out across more Amazon EC2 instances during the evenings and scale in on the weekends when there is little demand. Predictive scaling occurs before customer demand changes, ensuring a smooth experience while new resources are created and brought online.

## Troubleshooting and Remediation

*Question:* Where do problems occur?

As Werner Vogels, VP and CTO of AWS, once said, “Everything fails all the time.” No system is impervious to failure. By gathering potentially relevant information ahead of time, it becomes easier to determine causes for failure. By collecting this information, you can reduce mean time between failure (MTBF), mean time to resolution (MTTR), and other key operational performance metrics.

## Learning and Improvement

*Question:* “Can you detect or prevent problems in the future?”

By evaluating operational metrics over time, you can reveal patterns and common issues in your systems.



When choosing metrics, align them closely to your business processes to provide a better customer experience. For example, suppose that you have an application running in AWS Elastic Beanstalk. Unknown to you, the application has a memory leak. Without tracking memory utilization over time, you will not have insight into why customers are experiencing degraded performance. If your Elastic Beanstalk environment is configured to scale out based on CPU utilization, it is possible that no new instances are launched to serve customer requests. In this case, the memory leak prevents new requests from being processed, causing a drop in CPU utilization. Without comprehensive tracking of system performance, issues such as this can go unnoticed until system-wide outages occur.

These factors impact what is referred to as the health of your systems. As a developer and contributor, you are not only responsible for the code that you develop but also for the operational health of these services. It is vital to align operational and health metrics properly with customer expectations and experiences.

# Amazon CloudWatch

*Amazon CloudWatch* is a monitoring and metrics service that provides you with a fully managed system to collect, store, and analyze your metrics and logs. By using CloudWatch, you can create notifications on changes in your environment.

Typical use cases include the following:

- Infrastructure monitoring and troubleshooting
- Resource optimization
- Application monitoring
- Logging analytics
- Error reporting and notification

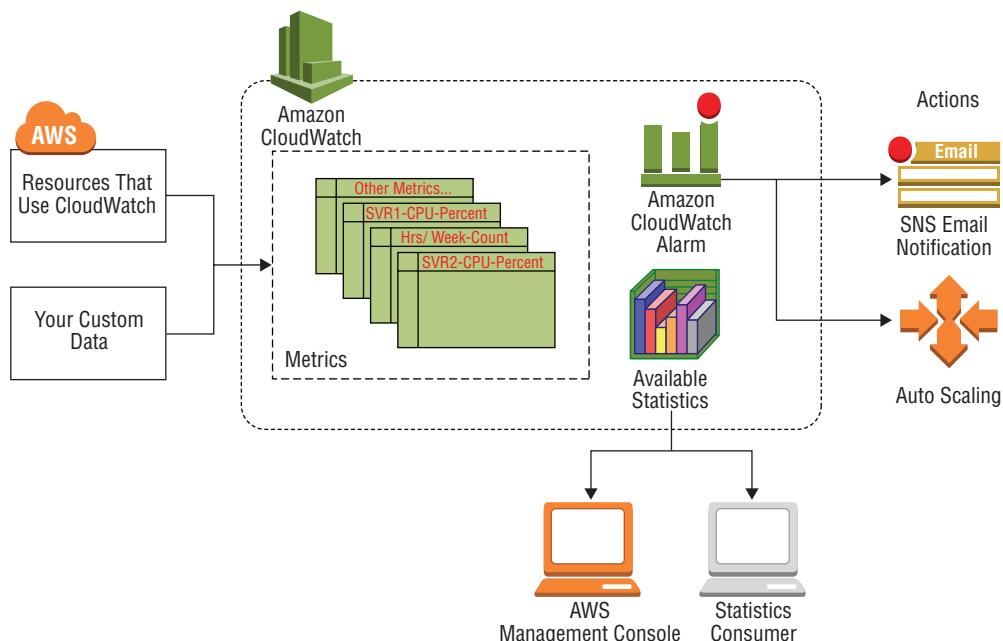
CloudWatch enables you to collect and store monitoring and operations data from logs, metrics, and events that run on AWS and on-premises resources. To ensure that your applications run smoothly, you can use CloudWatch to perform the following tasks:

- Set alarms
- Visualize logs and metrics
- Automate recovery from errors
- Troubleshoot issues
- Discover insights that enable you to optimize your resources

## How Amazon CloudWatch Works

CloudWatch acts as a *metrics repository*, storing metrics and logs from various sources. These metrics can come from AWS resources using built-in or custom metrics. Figure 15.2 illustrates the role of CloudWatch in operational health.

**FIGURE 15.2** Diagram of Amazon CloudWatch



CloudWatch can process these metrics into statistics that are made available through the CloudWatch console, AWS APIs, the AWS Command Line Interface (AWS CLI), and AWS software development kits (AWS SDKs). Using CloudWatch, you can display graphs, create alarms, or integrate with third-party solutions.

## Amazon CloudWatch Metrics

To understand CloudWatch better, especially how data is collected and organized, review the following terms.

### Built-In Metrics

A *metric* is a set of time-series data points that you publish to CloudWatch. For example, a commonly monitored metric for Amazon EC2 instances is CPU utilization. Data points can come from multiple systems, both AWS and on-premises. You can also define custom metrics based on data specific to your system. A metric is identified uniquely by a namespace, a name, and zero or more dimensions.

### Namespace

A *namespace* is a collection of metrics or a container of related metrics; for example, namespaces used by AWS offerings or services that all start with AWS. Amazon EC2 uses the AWS/EC2 namespace. As a developer, you can create namespaces for different components of your applications, such as front-end, backend, and database components.

### Name

A *name* for a given metric defines the attribute or property that you are monitoring; for example, CPU Utilization in the AWS/EC2 namespace. The AWS/EC2 namespace contains various metrics that are important to monitoring the health of Amazon EC2 resources, such as CPU Utilization, Disk I/O, Network I/O, or Status Check. You can also create custom metrics for attributes, such as request latency, HTTP 400/500 response codes, and throttling.

### Dimension

A *dimension* is a name/value pair used to define a metric uniquely. For example, for the namespace AWS/EC2 and name/metric CPUUtilization, the dimension might be InstanceId. For a fleet of Amazon EC2 instances, you can measure CPUUtilization as one metric for multiple dimensions (one for each instance). You can use the dimensions to structure and organize the data points you gather.



When you’re creating metrics, consider defining namespaces that align with your different services and assigning dimensions as important metrics that describe the health of that service. For example, if you have a front-end web fleet running NGINX servers, then dimensions, such as requests-per-second, response time, active connections, and response codes, could help you determine what configuration changes would optimize system performance.

### Data Points

When data is published to CloudWatch, it is pushed in sets of data points. Each data point contains information such as the timestamp, value, and unit of measurement.

## Timestamp

*Timestamps* are `dateTime` objects with the complete date and time; for example, `2016-10-31T23:59:59Z`. Although not required, AWS recommends formatting times as Coordinated Universal Time (UTC).

## Value

The *value* is the measurement for the data point.

## Unit

A *unit* of measurement is used to label your data. This offers a better understanding of what the value represents. Example units include Bytes, Seconds, Count, and Percent. If you do not specify a unit in CloudWatch, your data point units are designated as None.

CloudWatch stores this data based on the *retention period*, which is the length of time to keep data points available. Data points are stored in CloudWatch based on how often the data points are published.

- Data points with a published frequency less than 60 seconds are available for 3 hours. These data points are high-resolution custom metrics.
- Data points with a published frequency of 60 seconds (1 minute) are available for 15 days.
- Data points with a published frequency of 300 seconds (5 minutes) are available for 63 days.
- Data points with a published frequency of 3,600 seconds (1 hour) are available for 455 days (15 months).

From these data points, CloudWatch can calculate statistics to provide you with insight into your application, service, or environment. In the next section, you will discover how CloudWatch calculates and organizes these statistics.

## Statistics

CloudWatch provides statistics based on metric data provided to the service. *Statistics* are aggregations of data points over specified periods of time for specified metrics. A *period* is the length of time, in seconds. Periods can be defined in values of 1, 5, 10, 30, or any multiple of 60 seconds (up to 86,400 seconds, or 1 day). The available statistics in CloudWatch include the following:

- `Minimum` (`Min`), the lowest value recorded over the specified period
- `Maximum` (`Max`), the highest value recorded over the specified period
- `Sum`, the total value of the samples added together over the specified period
- `Average` (`Avg`), the `Sum` divided by the `SampleCount` over the specified period
- `SampleCount`, the number of data points used in the calculation over the specified period
- `pNN`, percentile statistics for tracking metric outliers

Statistics can be used to gain insight into the health of your application and to help you determine the correct settings for various configurations. For example, you may want to implement automatic scaling on your fleet of Amazon EC2 instances in order to avoid having to launch and terminate instances manually. To do so, you must configure an Auto Scaling group. Configuration settings for an Auto Scaling group include the minimum, desired, and maximum number of instances to run in your account. By monitoring statistics over time, you can determine the minimum and maximum number of instances needed to support the average, minimum, and maximum workload.

CloudWatch statistics provide a powerful way to process large amounts of metrics at scale and present insightful data that is easy to consume. Now that you understand how CloudWatch metrics work and are organized, explore the metrics available.

## Aggregations

CloudWatch aggregates metrics according to the period of time you specify when retrieving statistics. When you request this statistic, you also can have CloudWatch filter the data points based on the dimensions of the metrics. For example, in Amazon DynamoDB, metrics are fetched across all DynamoDB operations. You can specify a filter on the dimension operations to exclude specific operations, such as `GetItem` requests. CloudWatch does not aggregate data across regions.

## Available Metrics

Table 15.1 describes the available metrics for Elastic Load Balancing resources. To discover all of the available metrics, refer to the AWS documentation.

**TABLE 15.1** Elastic Load Balancing Metrics

Namespace	AWS/ELB AWS/ApplicationELB AWS/NetworkELB
Dimensions	LoadBalancerName: name of the load balancer
Key metrics	<ul style="list-style-type: none"><li>▪ <code>HealthyHostCount</code>: number of responding backend servers</li><li>▪ <code>RequestCount</code>: number of IPv4 and IPv6 requests</li><li>▪ <code>ActiveConnectionCount</code>: total number of concurrent active connections from clients</li></ul>

Table 15.2 describes the available Amazon EC2 metrics.

**TABLE 15.2** Amazon EC2 Metrics

---

Namespace	AWS/EC2
Dimensions	<ul style="list-style-type: none"> <li>▪ InstanceId: identifier of a particular Amazon EC2 instance</li> <li>▪ InstanceType: type of Amazon EC2 instance, such as t2.micro, m4.large</li> </ul>
Key metrics	<ul style="list-style-type: none"> <li>▪ CPUUtilization: percentage of vCPU utilization on the instance</li> <li>▪ DiskReadOps, DiskWriteOps: number of operations per second on attached disk</li> <li>▪ DiskReadBytes, DiskWriteBytes: volume of bytes to transfer on attached disk</li> <li>▪ NetworkIn, NetworkOut: number of bytes sent or received by network interfaces</li> <li>▪ NetworkPacketsIn, NetworkPacketsOut: number of packets sent or received by network interfaces</li> </ul>

---



Amazon EC2 does not report memory utilization to CloudWatch. This is because memory is allocated in full to an instance by the underlying host. Memory consumption is visible only to the guest operating system (OS) of the instance. However, you can report memory utilization to CloudWatch using the CloudWatch agent.

Table 15.3 describes the AWS Auto Scaling group metrics.

**TABLE 15.3** AWS Auto Scaling Groups

---

Namespace	AWS/AutoScaling
Dimensions	AutoScalingGroupName: name of the Auto Scaling group
Key metrics	<ul style="list-style-type: none"> <li>▪ GroupMinSize, GroupMaxSize, GroupDesiredCapacity: minimum, maximum, and desired size of the Auto Scaling group</li> <li>▪ GroupInServiceInstances: number of instances up and running in the Auto Scaling group</li> <li>▪ GroupTotalInstances: total number of instances in the Auto Scaling group, regardless of state</li> </ul>

---

Table 15.4 describes the Amazon Simple Storage Service (Amazon S3) metrics.

**TABLE 15.4** Amazon S3 Metrics

---

Namespace	AWS/S3
Dimensions	<ul style="list-style-type: none"> <li>▪ BucketName: name of a specific Amazon S3 bucket</li> <li>▪ StorageType: the Amazon S3 storage class (STANDARD, STANDARD_IA, and GLACIER storage classes) of the bucket</li> </ul>
Key metrics	<ul style="list-style-type: none"> <li>▪ BucketSizeBytes: total size, in bytes, of data stored in an Amazon S3 bucket</li> <li>▪ NumberOfObjects: total number of objects stored in an Amazon S3 bucket</li> <li>▪ AllRequests: total number of requests made to an Amazon S3 bucket</li> </ul>

---

Table 15.5 describes the Amazon DynamoDB metrics.

**TABLE 15.5** Amazon DynamoDB Metrics

---

Namespace	AWS/DynamoDB
Dimensions	<ul style="list-style-type: none"> <li>▪ TableName: name of Amazon DynamoDB table</li> <li>▪ Operation: limits metrics to either a particular operation (PutItem, GetItem, UpdateItem, DeleteItem, Query, Scan, BatchGetItem) or BatchWriteItem</li> </ul>
Key metrics	<ul style="list-style-type: none"> <li>▪ ConsumedReadCapacityUnits, ConsumedWriteCapacityUnits: total number of read and write capacity units consumed</li> <li>▪ ThrottledRequests: requests to DynamoDB that exceed the provisioned throughput limits on a resource (such as a table or an index)</li> <li>▪ ReadThrottleEvents: requests to DynamoDB that exceed the provisioned read capacity units for a table or a global secondary index</li> <li>▪ WriteThrottleEvents: requests to DynamoDB that exceed the provisioned write capacity units for a table or a global secondary index</li> <li>▪ ReturnedBytes: size of response returned in request</li> <li>▪ ReturnedItemCount: number of items returned in request</li> </ul>

---

Table 15.6 describes the Amazon API Gateway metrics.

**TABLE 15.6** Amazon API Gateway Metrics

Namespace	AWS/ApiGateway
Dimensions	<ul style="list-style-type: none"><li>▪ ApiName: filters out metrics for a particular API</li><li>▪ ApiName, Method, Resource, Stage: filters out metrics for a particular API, method, resource, and stage</li><li>▪ ApiName, Stage: filters out metrics for a particular deployed stage of an API</li></ul>
Key metrics	<ul style="list-style-type: none"><li>▪ 4XXError: number of HTTP 400 errors</li><li>▪ 5XXError: number of HTTP 500 errors</li><li>▪ Latency: time between when Amazon API Gateway receives a request and when it responds to the client</li></ul>

Table 15.7 describes the AWS Lambda metrics.

**TABLE 15.7** AWS Lambda Metrics

Namespace	AWS/Lambda
Dimensions	FunctionName: name of your AWS Lambda function
Key metrics	<ul style="list-style-type: none"><li>▪ Invocations: number of executions of your AWS Lambda function</li><li>▪ Errors: number of executions in which your AWS Lambda function failed</li><li>▪ Duration: total time for each execution of your AWS Lambda function</li></ul>

Table 15.8 describes the Amazon Simple Queue Service (Amazon SQS) metrics.

**TABLE 15.8** Amazon SQS Metrics

Namespace	AWS/SQS
Dimensions	QueueName: name of the Amazon SQS queue
Key metrics	<ul style="list-style-type: none"> <li>▪ ApproximateNumberOfMessagesVisible: number of messages currently available for retrieval</li> <li>▪ ApproximateNumberOfMessagesNotVisible: number of messages currently being processed, or messages that are inflight (Visibility Timeout is still active)</li> <li>▪ NumberOfMessagesDeleted: number of messages that have been deleted</li> </ul>



Amazon SQS does not report the total number of messages in the queue. You can find this value by adding ApproximateNumberOfMessagesVisible and ApproximateNumberOfMessagesNotVisible.

Table 15.9 describes the Amazon Simple Notification Service (Amazon SNS) metrics.

**TABLE 15.9** Amazon SNS Metrics

Namespace	AWS/SNS
Dimensions	TopicName: name of the Amazon SNS topic
Key metrics	<ul style="list-style-type: none"> <li>▪ NumberOfMessagesPublished: number of messages sent to an SNS topic</li> <li>▪ NumberOfNotificationsDelivered: number of messages that were successfully delivered to subscribers</li> <li>▪ NumberOfNotificationsFailed: number of messages that were unsuccessfully delivered to subscribers</li> </ul>

## Custom Metrics

In addition to the built-in metrics that AWS provides, CloudWatch also supports custom metrics that you can publish from your systems. This section includes some commands that you can use to publish metrics to CloudWatch.

## High-Resolution Metrics

With custom metrics, you have two options for resolution (the time interval between data points) for your metrics. You can use *standard resolution* for data points that have a granularity of one minute or *high resolution* for data points that have a granularity of less than one second. By default, most metrics delivered by AWS services have standard resolution.

## Publishing Metrics

CloudWatch supports multiple options when you publish metrics. You can publish them as single data points, statistics sets, or zero values. Single data points are optimal for most telemetry. However, statistics sets are recommended for values with high-resolution data points in which you are sampling multiple times per minute. *Statistics sets* are sets of calculated values, such as minimum, maximum, average, sum, and sample count, as opposed to individual data points. The value 0 is for applications that have periods of inactivity, where no data is sent. The following are some sample scripts using the AWS CLI to publish data points.

### USING THE AWS CLI TO PUBLISH SINGLE DATA POINTS

The following commands each publish a single data point under the Metric Name PageViewCount to the Namespace MyService with respective values and timestamps. You are not required to create a metric name or namespace. CloudWatch is aware of the data points to a metric or creates a new metric if it does not exist.

```
aws cloudwatch put-metric-data \
    --metric-name PageViewCount \
    --namespace MyService \
    --value 2 \
    --timestamp 2018-10-20T12:00:00.000Z

aws cloudwatch put-metric-data \
    --metric-name PageViewCount \
    --namespace MyService \
    --value 4 \
    --timestamp 2018-10-20T12:00:01.000Z

aws cloudwatch put-metric-data \
    --metric-name PageViewCount \
    --namespace MyService \
    --value 5 \
    --timestamp 2018-10-20T12:00:02.000Z
```

### USING THE AWS CLI TO PUBLISH STATISTICS SETS

The following command publishes a statistic set to the metric-name PageViewCount to the namespace MyService, with values for various statistics (Sum 11, Minimum 2, Maximum 5), and SampleCount 3 with the corresponding timestamp:

```
aws cloudwatch put-metric-data \
    --metric-name PageViewCount \
    --namespace MyService \
    --statistic-values Sum=11,Minimum=2,Maximum=5,SampleCount=3 \
    --timestamp 2018-10-14T12:00:00.000Z
```

### USING THE AWS CLI TO PUBLISH THE VALUE ZERO

The following command publishes a single data point with the value 0 to the metric-name PageViewCount to the namespace MyService with the corresponding timestamp:

```
aws cloudwatch put-metric-data \
    --metric-name PageViewCount \
    --namespace MyService \
    --value 0 \
    --timestamp 2018-10-14T12:00:00.000Z
```

### Retrieving Statistics for a Metric

After you publish data to CloudWatch, you may want to retrieve statistics for a specified metric of a given resource.

### USING THE AWS CLI TO RETRIEVE STATISTICS FOR A METRIC

This command retrieves the Sum, Max, Min, Average, and SampleCount statistics for the metric-name PageViewCount to the namespace MyService with a period interval of 60 seconds between the start-time and end-time. This means that CloudWatch will aggregate data points in one-minute intervals to calculate statistics.

```
aws cloudwatch get-metric-statistics \
    --namespace MyService \
    --metric-name PageViewCount \
    --statistics "Sum" "Maximum" "Minimum" "Average" "SampleCount" \
    --start-time 2018-10-20T12:00:00.000Z \
    --end-time 2018-10-20T12:05:00.000Z \
    --period 60
```

Example output from this command displays a single data point for the Metric PageViewCount.

```
{  
    "Datapoints": [  
        {  
            "SampleCount": 3.0,  
            "Timestamp": "2016-10-20T12:00:00Z",  
            "Average": 3.6666666666666665,  
            "Maximum": 5.0,  
            "Minimum": 2.0,  
            "Sum": 11.0,  
            "Unit": "None"  
        }  
    ],  
    "Label": "PageViewCount"  
}
```

## Amazon CloudWatch Logs

Though most commercial standard applications already produce some form of logging, most modern applications are deployed in distributed or service-oriented architectures. Collecting and processing these logs can be a challenge as a system grows and expands across multiple regions. Centralized logging using CloudWatch Logs can overcome this challenge. With CloudWatch Logs, you can set up a central log storage location to ingest and process logs at scale.

### Log Aggregation

Setting up centralized logging with CloudWatch Logs is a straightforward process. The first step is to install and configure the CloudWatch agent, which is used to collect custom logs and metrics from Amazon EC2 instances or on-premises servers. You can choose which log files you want to ingest by pointing to the locations using a JavaScript Object Notation (JSON) configuration file. The second step is to configure AWS Identity and Access Management (IAM) roles or users to grant permission for the agent to publish logs into CloudWatch. In addition to the CloudWatch agent, you can also send metrics to CloudWatch using the AWS CLI, AWS SDK, or AWS API.

Because you are collecting logs from multiple sources, CloudWatch organizes your logs into three conceptual levels: groups, streams, and events.

#### Log Groups

A *log group* is collection of log streams. For example, if you have a service that consists of a cluster of multiple machines, a log group would be a container for the logs from each of the individual instances.

## Log Streams

A *log stream* is a sequence of log events such as a single log file from one of your instances.

## Log Events

A *log event* is a record of some activity from an application, process, or service. This is analogous to a single line in a log file.

CloudWatch stores log events based on your retention settings, which are assigned at the log group. The default configuration is to store log data in Amazon CloudWatch Logs indefinitely. You are charged for any data stored in CloudWatch Logs in addition to data transferred out of the service. You can export CloudWatch Logs to Amazon S3 for long-term storage, which is valuable when regulations require long-term log retention. Long-term retention can be combined with Amazon S3 lifecycle policies to archive data to Amazon S3 Glacier for additional cost savings.

## Log Searches

With centralized logging on CloudWatch Logs, you do not need to search through hundreds of individual servers to find a problem. After logs are ingested into CloudWatch Logs, you can search for logs through a central location using metric filters.

## Metric Filters

A *metric filter* is a text pattern used to parse log data for specific events. As an example, consider the log in Table 15.10.

**TABLE 15.10** Example Log

Line	Log Event
1	[ERROR] Caught IllegalArgumentException
2	[ERROR] Unhandled Exception
3	Another message
4	Exiting with ERRORCODE: -1
5	[WARN] Some message
6	[ERROR][WARN] Some other message

To look for occurrences of the ERROR event, you use ERROR as your metric filter, as illustrated in Table 15.11. CloudWatch will search for that term across the logs.

**TABLE 15.11** Example Metric Filters

Metric Filter	Description
""	Matches all log events.
"ERROR"	Matches log events containing the term "ERROR." Based on the events in the example log in Table 15.10, this metric filter would find lines 1–3 and 6.
"ERROR" - "EXITING"	Matches log events containing the term "ERROR" except "EXITING." Based on the events in the example log in Table 15.10, this metric filter would find lines 1, 2, and 6.
"ERROR Exceptions"	Matches log events containing both terms "ERROR" and "Exceptions." This filter is an AND function. Based on the events in the example log in Table 15.10, this metric filter would find lines 1 and 2.
"?ERROR ?WARN"	Matches log events containing either the term "ERROR" or "WARN." This filter is an OR function. Based on the event in the example log in Table 15.10, this metric filter would find lines 1, 2, 4, and 6.

If your logs are structured in JSON format, CloudWatch can also filter object properties. Consider the following example JSON log.

### Example AWS CloudTrail JSON Log Event

```
{
  "user": {
    "id": 1,
    "email": "Admin@example.com"
  },
  "users": [
    {
      "id": 2,
      "email": "John.Doe@example.com"
    },
    {
      "id": 3,
      "email": "Jane.Doe@example.com"
    }
  ]
}
```

```

        }
    ],
    "actions": [
        "GET",
        "PUT",
        "DELETE"
    ],
    "coordinates": [
        [0, 1, 2],
        [4, 5, 6],
        [7, 8, 9]
    ]
}

```

You can create a metric filter that selects and compares certain properties of this event, as shown in Table 15.12.

**TABLE 15.12** Example JSON Metric Filters

JSON Metric Filter	Description
{ (\$.user.id = 1) && (\$.users[0].email = "John.Doe@example.com") }	Check that the property <code>user.id</code> equals 1 and the first user's email is <code>John.Doe@example.com</code> . The preceding log event would be returned.
{ (\$.user.id = 2 && \$.users[0].email = "John.Doe@example.com")    \$.actions[2] = "GET" }	Check that the property <code>user.id</code> equals 2 and the first user's email is <code>John.Doe@example.com</code> or the second action is <code>GET</code> . The preceding example would not be returned, because the second action is <code>PUT</code> , not <code>GET</code> .

## Log Processing

Instead of having to write additional code to add monitoring to your application, CloudWatch can process logs that you already generate and provide valuable metrics. Using the example from the previous section, the same metric filter can be used to generate metrics corresponding to the number of occurrences of the term `ERROR` in your logs.

## Amazon CloudWatch Alarms

After data points are established in CloudWatch, either as metrics or as logs (from which you generate metrics), you can set *alarms* to monitor your metrics and trigger actions in

response to changes in state. CloudWatch alarms have three possible states: OK, ALARM, and INSUFFICIENT\_DATA. Table 15.13 defines each alarm state.

**TABLE 15.13** Alarm States

State	Description
OK	The metric or expression is within the defined threshold.
ALARM	The metric or expression is outside of the defined threshold.
INSUFFICIENT_DATA	The alarm has just started, the metric is not available, or not enough data is available for the metric to determine the alarm state.

An ALARM state may not indicate a problem. It means that the given metric is outside the defined threshold. For example, you have two alarms for Auto Scaling groups: one for high CPU utilization and one for low CPU utilization. During normal use, both alarms should be OK, indicating that you have adequate capacity to handle the current workload. If your workload changes, the high CPU utilization metric threshold may be breached, sending the corresponding alarm into ALARM state. With an Auto Scaling group, the alarm's state change triggers a scale-out event, adding capacity to your infrastructure.

## Using Amazon CloudWatch Alarms

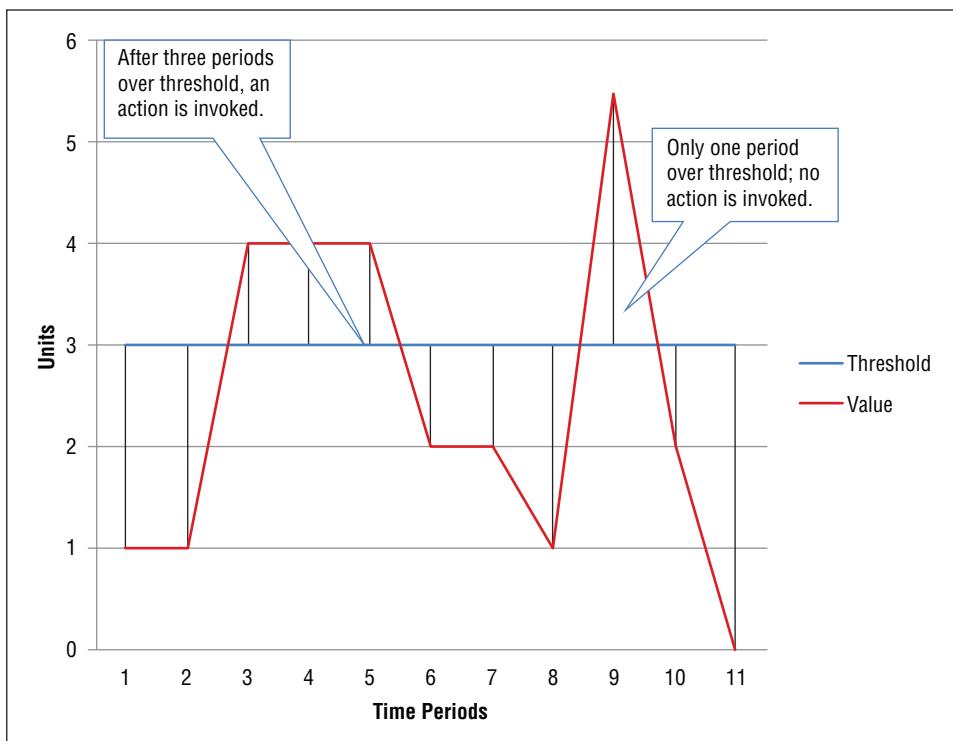
When you create an alarm, specify three settings that determine when the alarm should change states: the threshold, period, and data points on which you want to notify, as described in Table 15.14.

**TABLE 15.14** Alarm Settings

Setting	Description
Period	The length of time (in seconds) to evaluate the metric or expression to create each individual data point for an alarm. If you choose one minute as the period, there is one data point every minute.
Evaluation Period	The number of the most recent periods, or data points, to evaluate when determining alarm state.
Data Points to Alarm	The number of data points within the evaluation period that must breach the specified threshold to cause the alarm to go to the ALARM state. These data points do not have to be consecutive.

Figure 15.3 illustrates how an alarm works based on configuration settings.

**FIGURE 15.3** Alarm evaluation



The figure illustrates a threshold configured to the value 3 (in blue), a period set to 3, and data points in red. Notice how the settings drive the alarm occurrence. Even though the data points breach the threshold after the third period, it is not sustained for the required three periods to be in an ALARM state. Only after the fifth period would the alarm change to an ALARM state (the upper threshold is breached for three periods). Between the fifth and sixth period, the data points drop below the threshold. However, because the state has not dropped below the threshold for three periods, it does not change to an OK state until the eighth period. It remains in the OK state past the ninth period because three consecutive periods exceeding the threshold are necessary for the alarm state to change.

Alarms can trigger Amazon EC2 actions and EC2 Auto Scaling actions. CloudWatch can leverage Amazon SNS or Amazon SQS for alarm state notifications, both of which provide numerous integrations with other AWS services.

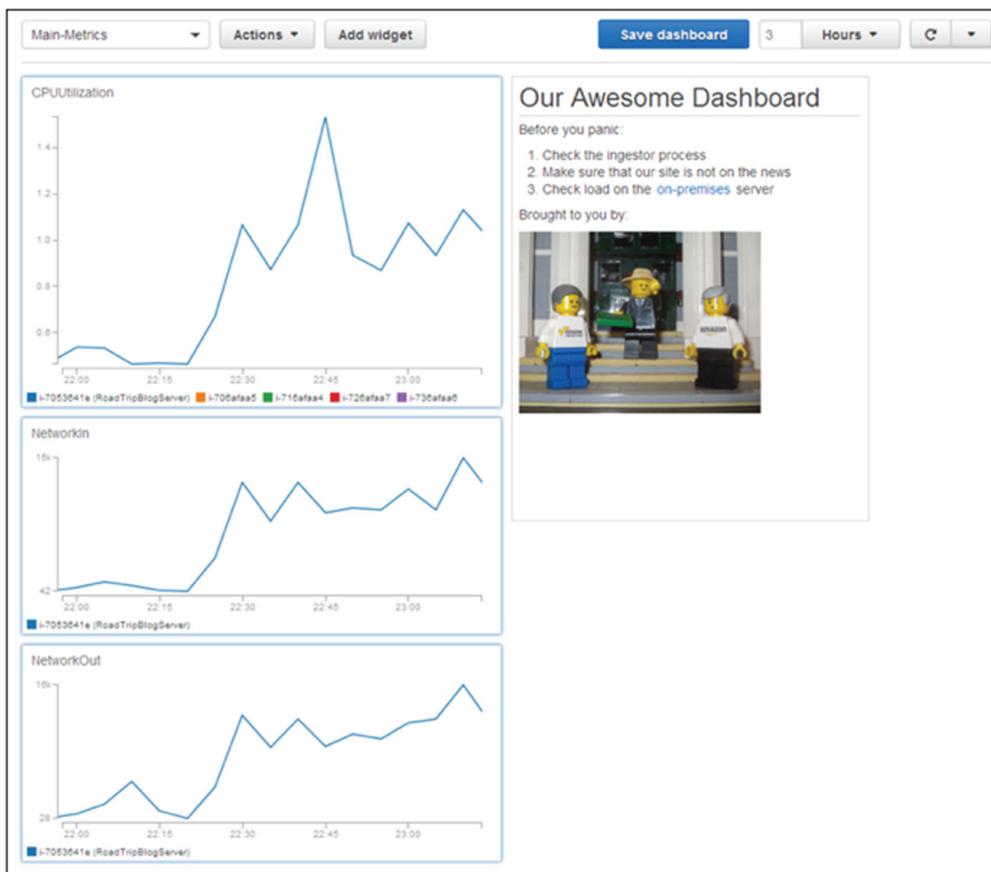


Exercise caution when creating email notifications for alarms in your environment. This can lead to many unnecessary emails to you or your team. Ultimately, these notifications get filtered as spam or result in “notification fatigue.” Evaluate your alarms and the metrics you are monitoring to determine whether notifications are necessary. If they are only status updates, set notifications sparingly.

## Amazon CloudWatch Dashboards

CloudWatch offers a convenient way to observe operational metrics for all of your applications. *CloudWatch dashboards* are customizable pages in the CloudWatch console that you can use to monitor resources in a single view (see Figure 15.4).

**FIGURE 15.4** Amazon CloudWatch dashboard



CloudWatch dashboards provide customizable status pages in the CloudWatch console. These status pages can be used to monitor resources across multiple regions and on-premises in a consolidated view using widgets. Each widget can be customized to present information in CloudWatch in a user-friendly way so that educated decisions can be made based on the current status of your system.

## AWS CloudTrail

All actions in your AWS account are composed of API calls, regardless of the origin (the AWS Management Console or programmatic/scripted actions). As you create resources in your account, API calls are being made to AWS services in different regions around the world. *AWS CloudTrail* is a fully managed service that continuously monitors and records API calls and stores them in Amazon S3. You can use these logs to troubleshoot and resolve operational issues, meet and verify regulatory compliance, and monitor or alarm on specific events in your account. CloudTrail supports most AWS services, making it easy for IT and security administrators to analyze activity in accounts. IT auditors can also use log files as compliance aids.

CloudTrail helps answer the following five key questions about monitoring access:

- Who made the API call?
- When was the API call made?
- What was the API call?
- Which resources were acted upon in the API call?
- Where was the origin of the API call?

### AWS CloudTrail Events

A *CloudTrail event* is any single API activity in an AWS account. This activity can be an action triggered by any of the following:

- AWS IAM user
- AWS IAM role
- AWS service

CloudTrail tracks two types of events: management events and data events. Events are recorded in the region where the action occurred, except for global service events.

#### Management Events

*Management events* give insight into operations performed on AWS resources, such as the following examples:

**Configuring security:** An example is attaching a policy to an IAM role.

**Configuring routing rules:** An example is adding inbound security group rules.

## Data Events

*Data events* give insight into operations that store data in (or extract data from) AWS resources, such as the following examples:

**Amazon S3 object activity:** Examples are GetObject and PutObject operations.

**AWS Lambda function executions:** These use the InvokeFunction operation.

By default, CloudTrail tracks the last 90 days of API history for management events. The following is example output for a CloudTrail event:

```
{  
    "Records": [  
        {  
            "eventVersion": "1.01",  
            "userIdentity": {  
                "type": "IAMUser",  
                "principalId": "AIDAJDPLRKLG7UEXAMPLE",  
                "arn": "arn:aws:iam::123456789012:user/Alice",  
                "accountId": "123456789012",  
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",  
                "userName": "Alice",  
                "sessionContext": {  
                    "attributes": {  
                        "mfaAuthenticated": "false",  
                        "creationDate": "2014-03-18T14:29:23Z"  
                    }  
                }  
            },  
            "eventTime": "2014-03-18T14:30:07Z",  
            "eventSource": "cloudtrail.amazonaws.com",  
            "eventName": "StartLogging",  
            "awsRegion": "us-west-2",  
            "sourceIPAddress": "198.162.198.64",  
            "userAgent": "signin.amazonaws.com",  
            "requestParameters": {  
                "name": "Default"  
            },  
            "responseElements": null,  
            "requestID": "cdc73f9d-aea9-11e3-9d5a-835b769c0d9c",  
            "eventID": "3074414d-c626-42aa-984b-68ff152d6ab7"  
        },  
        ... additional entries ...  
    ]
```

This event provides the following information:

- The user who made the request from the `userIdentityField`. In this example, it is the IAM user Alice.
- When the request was made (the `eventTime`). In this case, it is `2014-03-18T14:30:07Z`.
- Where the request was made (the `sourceIPAddress`). In this case, it is `198.162.198.64`.
- The action the request is trying to perform (the `eventName`). In this case, it is the `StartLogging` operation.

As a security precaution, you can use events such as this example to configure alerts when an IAM user attempts to sign in to the AWS Management Console too many times.

### Global Service Events

Some AWS services allow you to create, modify, and delete resources from any region. These are referred to as *global services*. Examples of global services include the following:

- IAM
- AWS Security Token Service (AWS STS)
- Amazon CloudFront
- Amazon Route 53

Global services are logged as occurring in US East (N. Virginia) Region. Any trails created in the CloudTrail console log global services by default, which are delivered to the Amazon S3 bucket for the trail.

### Trails

If you need long-term storage of events (for example, for compliance purposes), you can configure a trail of events as log files in CloudTrail. A trail is a configuration that enables delivery of CloudTrail events to an Amazon S3 bucket, Amazon CloudWatch Logs, and Amazon CloudWatch Events. When you configure a trail, you can filter the events that you want to be delivered.

## AWS X-Ray

The services covered so far are centered on the concept of using logs as monitoring and troubleshooting tools. Developers often write code, test the code, and inspect the logs. If there are errors, they may add breakpoints, run the test again, and add log statements. This works well in small cases, but it becomes cumbersome as teams, software, and infrastructure grow. Traditional troubleshooting and debugging processes do not work well at scaling across multiple services. Troubleshooting cross-service and cross-region interactions can be especially difficult when different systems use varying log formats.

AWS X-Ray is a service that collects data about requests served by your application. It provides tools you can use to view, filter, and gain insights into that data to identify issues and opportunities for optimization.

## AWS X-Ray Use Cases

X-Ray helps developers build, monitor, and improve applications. Use cases include the following:

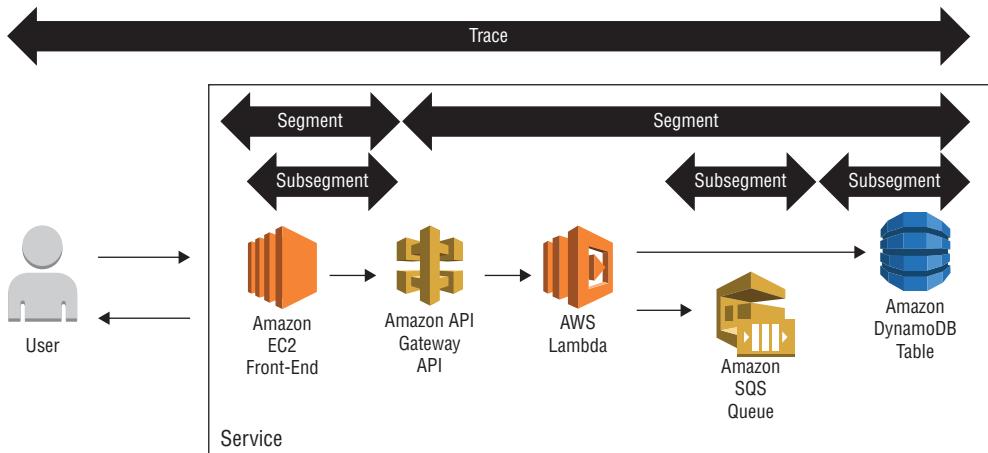
- Identifying performance bottlenecks
- Pinpointing specific service issues
- Identifying errors
- Identifying impact to users

X-Ray integrates with the AWS SDK, adding traces to track your application requests as they are generated and received from various services.

## Tracking Application Requests

To understand better how X-Ray works, consider the example service shown in Figure 15.5. In this service, the front-end fleet relies on a backend API, which is built using API Gateway, which acts as proxy to Lambda. Lambda then uses Amazon DynamoDB to store data.

**FIGURE 15.5** Microservice example



X-Ray can track a user request using a trace, segment, and subsegment.

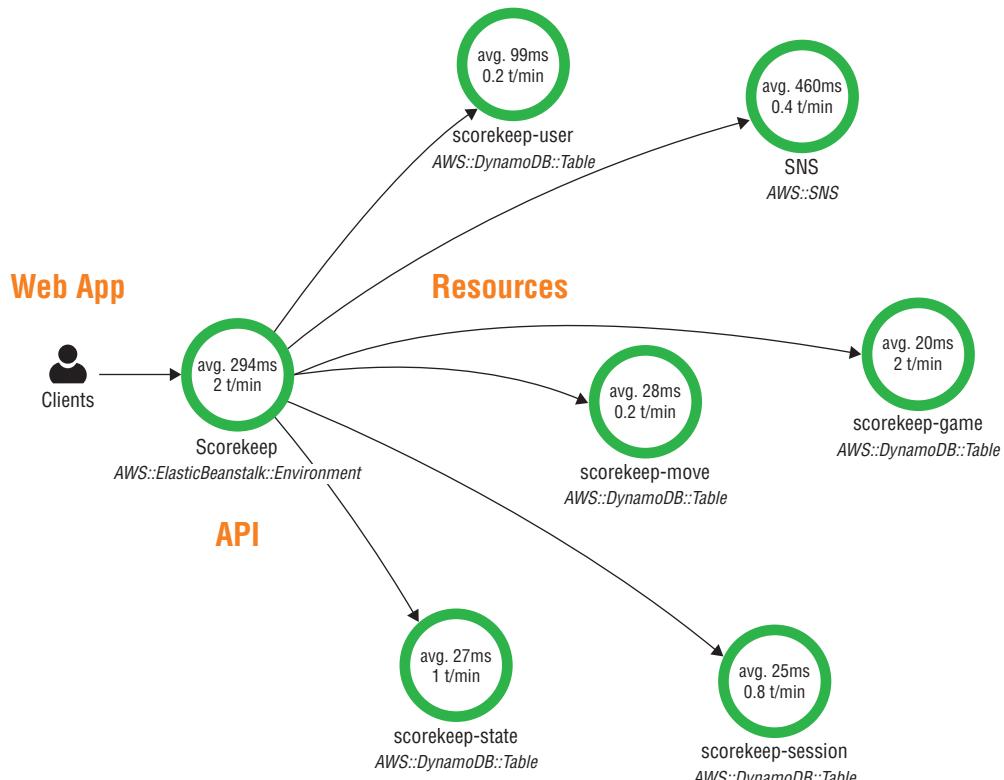
**Trace** A *trace* is the path of a request through your application. This is the end-to-end request from the client—from its entry into your environment to the backend and back to the user. A trace ID is passed through the AWS services with the request so that X-Ray can collate related segments.

**Segment** A *segment* is data from a particular service. When a segment is reported to X-Ray, a trace ID is reported. Segments are analogous to links in a chain whereby the chain is the request generated by the user. In the example microservice, two segments correspond to two services: the front-end service and the backend API.

**Subsegment** A *subsegment* identifies the underlying API calls made from a particular service. Subsegments are collated into segments. In this scenario, the backend API sends requests to Amazon DynamoDB and Amazon SQS.

From these components of a request, X-Ray compiles the traces into a service graph that describes the components and their interactions needed to complete a request. A *service graph* is a visual representation of the services and resources that make up your application. Figure 15.6 shows an example of a service graph.

**FIGURE 15.6** Example service graph for an application



The service graph provides an overview of the health of various aspects of your system, such as average latencies and request rates between your services and dependent resources. The colored circles also show the ratio of different response codes, as listed in Table 15.15.

**TABLE 15.15** AWS X-Ray Service Graph Status Codes

Color	Status Code
Purple	Throttling or HTTP 5XX codes
Orange	Client-side or HTTP 4XX codes
Red	Fault application failure
Green	OK or HTTP 2XX codes

X-Ray provides a convenient way for you to view system performance and to identify problems or bottlenecks in your applications. However, it does not provide auditing capabilities or the tracking of all requests to a system. X-Ray collects a statistically significant number of requests to a system so that meaningful insights can be provided. These insights enable you to focus on troubleshooting a particular service or improvements to a specific component of your application.

## Summary

AWS provides multiple options for monitoring and troubleshooting your applications. As you have discovered, AWS services help you manage logs from various systems, either running on the cloud or on-premises, create triggers that notify you about application health and issues in your infrastructure, and build applications with modern debugging tools for distributed applications. These services overcome the difficulties of creating a centralized logging solution.

## Exam Essentials

**Know what Amazon CloudWatch is and why it is used.** CloudWatch is the service used to aggregate, analyze, and alert on metrics generated by other AWS services. It is used to monitor the resources you create in AWS and the on-premises infrastructure. You can use CloudWatch to store logs from your applications and trigger actions in response to events.

**Know what common metrics are available for Amazon Elastic Compute Cloud (Amazon EC2) in Amazon CloudWatch.** Amazon EC2 metrics in CloudWatch include the following:

- CPUUtilization
- DiskReadOps
- DiskReadBytes
- DiskWriteOps
- DiskWriteBytes
- NetworkIn
- NetworkOut
- StatusCheckFailed

Amazon EC2 does not report OS-level metrics such as memory utilization.

**Understand the difference between high-resolution and standard-resolution metrics.** High-resolution metrics are delivered in a period of less than one minute. Standard-resolution metrics are delivered in a period greater than or equal to one minute.

**Know what AWS CloudTrail is and why it is used.** CloudTrail is used to monitor API calls made to the AWS Cloud for various services. CloudTrail helps IT administrators, IT security administrators, DevOps engineers, and auditors to enable compliance and the monitoring of access to AWS resources within an account.

**Know what AWS CloudTrail tracks automatically.** By default, CloudTrail tracks the last 90 days of activity. These events are limited to management events with create, modify, and delete API calls.

**Understand the difference between AWS CloudTrail management and data events.**

Management events are operations performed on resources in your AWS account. Data events are operations performed on data stored in AWS resources. Examples are creating or deleting objects in Amazon S3 and inserting or updating items in an Amazon DynamoDB table.

**Know what AWS X-Ray is and why it is used.** X-Ray is a service that collects data about your application requests, including the various subservices or systems that perform tasks to complete a request. X-Ray is commonly used to help developers find bottlenecks in distributed applications and monitor the health of various components in their services.

**Know the basics of AWS X-Ray and how it helps troubleshoot applications.** X-Ray records requests by initiating a trace ID with the origin of the request. This trace ID is added as a header to the request that propagates to various services. If you enable the X-Ray SDK in your applications, X-Ray submits telemetry and the request as segments for each service and subsegments for downstream services upon which you depend. Using these traces, X-Ray collates the data to view request performance metrics, such as latency and error rates. The data can then be used to create a graph of your application and its dependencies and the health of any requests your application might make.

# Resources to Review

Launch your application with the AWS Startup Kit:

<https://aws.amazon.com/blogs/startups/launch-your-app-with-the-aws-startup-kit/>

AWS re:Invent 2018: Monitor All Your Things: Amazon CloudWatch in Action with BBC (DEV302):

<https://www.youtube.com/watch?v=uuBuc60AcVY>

Create a CloudWatch Dashboard:

[https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/create\\_dashboard.html](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/create_dashboard.html)

What is Amazon CloudWatch?

<https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html>

Using Amazon CloudWatch Alarms:

<https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/AlarmThatSendsEmail.html>

Using Amazon CloudWatch Metrics:

[https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/working\\_with\\_metrics.html](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/working_with_metrics.html)

AWS re:Invent 2018: Augmenting Security Posture and Improving Operational Health with AWS CloudTrail (SEC323):

<https://www.youtube.com/watch?v=YWzmoDzzg4U>

What is Amazon CloudWatch Logs?

<https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html>

SID 341: Using AWS CloudTrail Logs for Scalable, Automated Anomaly Detection:

<https://github.com/aws-samples/aws-cloudtrail-analyzer-workshop>

What Is AWS CloudTrail?

<https://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudtrail-user-guide.html>

CloudTrail Concepts:

<https://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudtrail-concepts.html>

AWS X-Ray Sample Application:

<https://docs.aws.amazon.com/xray/latest/devguide/xray-scorekeep.html>

AWS re:Invent 2017: Monitoring Modern Applications: Introduction to AWS X-Ray (DEV204):

<https://www.youtube.com/watch?v=kFsIZsaqpzE>

AWS X-Ray Service Graph:

<https://docs.aws.amazon.com/xray/latest/devguide/xray-concepts.html#xray-concepts-servicegraph>

AWS CloudTrail Event History Now Available to All Customers:

<https://aws.amazon.com/about-aws/whats-new/2017/08/aws-cloudtrail-event-history-now-available-to-all-customers/>

## Exercises

### EXERCISE 15.1

#### Create an Amazon CloudWatch Alarm on an Amazon S3 Bucket

It is common to monitor the storage usage of your Amazon S3 buckets and trigger notifications when there is a large increase in storage used. In this exercise, you will use the AWS CLI to configure an Amazon CloudWatch alarm to trigger a notification when more than 1 KB of data is uploaded to an Amazon S3 bucket.

If you need directions while completing this exercise, see “Using Amazon CloudWatch Alarms” here:

<https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/AlarmThatSendsEmail.html>

1. Create an Amazon S3 bucket in your AWS account. For instructions, see this page:

<https://docs.aws.amazon.com/AmazonS3/latest/user-guide/create-bucket.html>

2. Open the Amazon CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.

3. Select Alarms > Create Alarm.

4. Choose Select Metric.

- Select the All Metrics tab.
- Expand AWS Namespaces.
- Select S3.
- Select Storage Metrics.
- Select a metric where BucketName matches the name of the Amazon S3 bucket that you created and where Metric Name is BucketSizeBytes.

5. Choose Select Metric.
6. Under Alarm Details:
  - a. For Name, enter **S3 Storage Alarm**.
  - b. For the comparator, select  $\geq$  (greater than or equal to).
  - c. Set the value to 1000 for 1 KB.
7. Under Actions:
  - a. For Whenever This Alarm, select State Is ALARM.
  - b. For Send Notification To, select New List.
  - c. For Name, enter **My S3 Alarm List**.
  - d. For Email List, enter your email address.
8. Choose Create Alarm.

The alarm is created in your account. If you already have data in your Amazon S3 bucket, it is switched from Insufficient Data to Alarm state. Otherwise, try uploading several files to your bucket to monitor changes in alarm state.

To delete the alarm, follow these steps:

1. Open the Amazon CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
2. Select Alarms.
3. Select the alarm you want to delete.
4. For Actions, select Delete.

In this exercise, you created an Amazon CloudWatch alarm to notify administrators when large files are uploaded to Amazon S3 buckets in your account.

---

## EXERCISE 15.2

### Enable an AWS CloudTrail Trail on an Amazon S3 Bucket

1. In this exercise, you will set up access logs to an Amazon S3 bucket in your account to monitor activity. Create an Amazon S3 bucket in your AWS account.

For instructions on how to do so, see the following:

<https://docs.aws.amazon.com/AmazonS3/latest/user-guide/create-bucket.html>

2. Open the AWS CloudTrail console at <https://console.aws.amazon.com/cloudtrail/>.

---

(continued)

**EXERCISE 15.2 (*continued*)**

3. Select Create Trail.
4. Set Trail name to **s3\_logs**.
5. Under Management Events, select None.
6. Under Data Events, select Add S3 Bucket.
7. For S3 bucket, enter your Amazon S3 bucket name.
8. Under Storage Location, for Create A New S3 bucket, select Yes.
9. For Name, enter a name for your Amazon S3 bucket.
- 10. Choose Create.**

In this exercise, you enabled AWS CloudTrail to record data events and store corresponding logs to an Amazon S3 bucket.

---

**EXERCISE 15.3****Create an Amazon CloudWatch Dashboard**

In this exercise, you will create an Amazon CloudWatch dashboard to see graphed metric data.

1. Open the Amazon CloudWatch console at <https://console.aws.amazon.com/cloudwatch/>.
2. In the navigation pane, select Dashboards.
3. Choose Create Dashboard.
4. For Dashboard Name, enter a name for your dashboard.
5. Select Create Dashboard.
6. In the modal window, select the Line graph.
7. Choose Configure.
8. From the available metrics, select one or more metrics that you want to monitor.
9. Choose Create Widget.
10. To add more widgets, choose Add Widget and repeat steps 6 through 9 for other widget types.
- 11. Choose Save Dashboard.**

In this exercise, you created an Amazon CloudWatch dashboard to create graphs of important metric data for resources in your account.

---

# Review Questions

1. You are required to set up dynamic scaling using Amazon CloudWatch alarms.  
Which of the following metrics could you monitor to trigger Auto Scaling events to scale out and scale in your instances?
  - A. High CPU utilization to trigger scale-in action, and low CPU utilization to trigger scale-out action
  - B. High CPU utilization to trigger scale-out action, and low CPU utilization to trigger scale-in action
  - C. High latency to trigger a scale-in action, and low latency to trigger a scale-out action
  - D. None of the above
2. What is the length of time that metrics are stored for a data point with a period of 300 seconds (5 minutes) in Amazon CloudWatch?
  - A. The data point is stored for 3 hours.
  - B. The data point is stored for 15 days.
  - C. The data point is stored for 30 days.
  - D. The data point is stored for 63 days.
  - E. The data point is stored for 455 days (15 months).
3. Which of the following does an AWS CloudTrail event *not* provide?
  - A. Who made the request
  - B. When the request was made
  - C. What request is being made
  - D. Why the request was made
  - E. Which resource was acted on
4. You must set up centralized logging for an application and create a cost-effective way to archive logs for compliance purposes.  
Which is the best solution?
  - A. Install the Amazon CloudWatch agent on your servers to ingest the logs and store them indefinitely.
  - B. Configure Amazon CloudWatch to ingest logs from your application servers.
  - C. Install the Amazon CloudWatch agent on your servers to ingest the logs and set a new retention period for logs with regular exports to Amazon S3 for archival.
  - D. None of the above.

5. Which of the following options allow logs and metrics to be ingested into Amazon CloudWatch? (Select THREE.)

- A. Install the Amazon CloudWatch agent and configure it to ingest logs.
- B. Execute API operations to push metrics to Amazon CloudWatch.
- C. Configure Amazon CloudWatch to pull logs from servers.
- D. Use the AWS CLI to push metrics to Amazon CloudWatch.

6. The following are Apache HTTP access logs.

Which filter pattern would select events matching 404 errors?

```
127.0.0.1 - - [24/Sep/2013:11:49:52 -0700] "GET /index.html HTTP/1.1" 404 287
127.0.0.1 - - [24/Sep/2013:11:49:52 -0700] "GET /index.html HTTP/1.1" 404 287
127.0.0.1 - - [24/Sep/2013:11:50:51 -0700] "GET ~/test/ HTTP/1.1" 200 3
127.0.0.1 - - [24/Sep/2013:11:50:51 -0700] "GET /favicon.ico HTTP/1.1" 404 308
127.0.0.1 - - [24/Sep/2013:11:50:51 -0700] "GET /favicon.ico HTTP/1.1" 404 308
127.0.0.1 - - [24/Sep/2013:11:51:34 -0700] "GET ~/test/index.html HTTP/1.1" 200 3
```

- A. 4xx
- B. 400
- C. 404
- D. None of the above

7. You build an application and enable AWS X-Ray tracing. You analyze the service graph and determine that the application requests to Amazon DynamoDB are not performing well and a majority of the issues are purple.

What kind of problem is your application experiencing?

- A. Throttling
- B. Error
- C. Faults
- D. OK

8. Which AWS service enables you to monitor resources and gather statistics, such as CPU utilization, from a single “pane of glass” interface?

- A. AWS CloudTrail logs
- B. Amazon CloudWatch alarms
- C. Amazon CloudWatch dashboards
- D. Amazon CloudWatch Logs

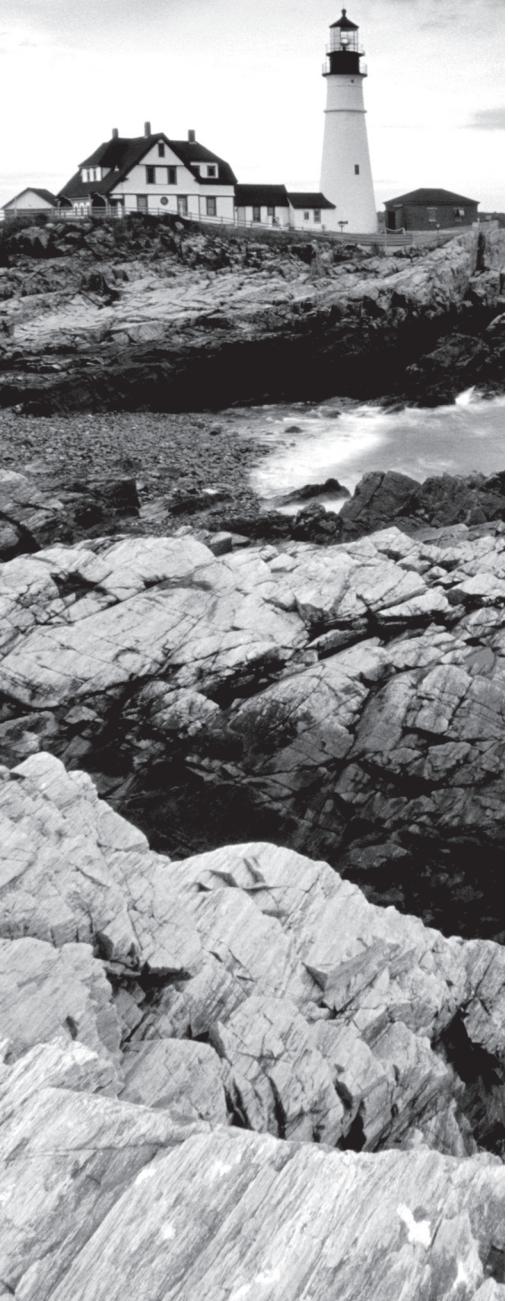
9. By default, what is the number of days of AWS account activity that you can view, search, and download from the AWS CloudTrail event history?

- A. 30 days
- B. 60 days
- C. 75 days
- D. 90 days

- 10.** Which of the following is not able to access AWS CloudTrail data?
- A.** AWS CLI
  - B.** AWS Management Console
  - C.** AWS CloudTrail API
  - D.** None of the above
- 11.** In AWS CloudTrail, which of the following are management events? (Select TWO.)
- A.** Adding a row to an Amazon DynamoDB table
  - B.** Modifying an Amazon S3 bucket policy
  - C.** Uploading an object to an Amazon S3 bucket
  - D.** Creating an Amazon Relational Database Service (Amazon RDS) database instance
  - E.** Sending a notification to Amazon Simple Notification Service (Amazon SNS)
- 12.** Suppose that you have a custom web application running on an Amazon Elastic Compute Cloud (Amazon EC2) instance.  
What steps are needed to configure this instance to send custom application logs to Amazon CloudWatch Logs? (Select THREE.)
- A.** Install the Amazon CloudWatch Logs agent.
  - B.** Attach an Elastic IP address to your Amazon EC2 instance.
  - C.** Configure the agent to send specific logs.
  - D.** Start the agent.
  - E.** Install the AWS Systems Manager agent.
- 13.** Which of the following are not supported Amazon CloudWatch alarm actions?
- A.** AWS Lambda functions
  - B.** Amazon Simple Notification Service (Amazon SNS) topics
  - C.** Amazon Elastic Compute Cloud (Amazon EC2) actions
  - D.** EC2 Auto Scaling actions
- 14.** Which of the following Amazon Elastic Compute Cloud (Amazon EC2) metrics is not directly available through Amazon CloudWatch metrics?
- A.** CPU utilization
  - B.** Network traffic in/out
  - C.** Disk I/O
  - D.** Memory (RAM) utilization
- 15.** Which of the following is the correct Amazon CloudWatch metric namespace for Amazon Elastic Compute Cloud (Amazon EC2) instances?
- A.** AWS/EC2
  - B.** Amazon/EC2
  - C.** AWS/EC2Instance
  - D.** Amazon/EC2Instance



# Chapter 16



# Optimization

---

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

## Domain 3: Development with AWS Services

- ✓ **3.4 Write code that interacts with AWS services by using APIs, SDKs, and AWS CLI.**

Content may include the following:

- Programming AWS APIs

## Domain 4: Refactoring

- ✓ **4.1 Optimize application to best use AWS services and features.**

Content may include the following:

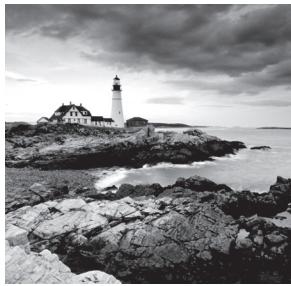
- Cost optimization
- Performance optimization
- Best practices for achieving optimization

## Domain 5: Monitoring and Troubleshooting

- ✓ **5.1 Write code that can be monitored.**

Content may include the following:

- Tools for cost monitoring
- Tools for performance monitoring



## Introduction to Optimization

Creating a software system is a lot like constructing a building. If the foundation is not solid, structural problems can undermine the integrity and function of the building. The AWS Well-Architected Tool helps you understand the pros and cons of decisions that you make while building systems on AWS. By using the tool, you will learn architectural best practices for designing and operating reliable, secure, efficient, and cost-effective systems in the AWS Cloud. When architecting technology solutions, if you neglect the five pillars of operational excellence, security, reliability, performance efficiency, and cost optimization, it can become challenging to build a system that delivers on your expectations and requirements. Incorporating these pillars into your architecture helps you to produce stable and efficient systems.

This chapter covers some of the best practices and considerations in designing systems with the most effective use of services and resources to achieve business outcomes at a minimal cost and maintain the optimal performance efficiency.

## Cost Optimization: Everyone's Responsibility

All teams help manage cloud costs, and cost optimization is everyone's responsibility. Make sure that costs are known from beginning to end, at every level, and from executives to engineers. Ensure that project owners and budget holders know what their upfront and ongoing costs are. Business decision makers must track costs against budgets and understand return on investment (ROI).

Encourage everyone to track their cost optimization daily so that they can establish a habit of efficiency and see the daily impact of their cost savings over time.

Developers' and engineers' contributions are a significant part of the organization's success. Every engineer can be a cost engineer. Engineers should design the code to consume resources only when needed, control the utilization, build sizing into architecture, and tag the resources to optimize usage.

## Tagging

By tagging your AWS resources, you can assign custom metadata to instances, images, and other resources. For example, you can categorize resources by owner, purpose, or environment, which helps you organize them and assign cost accountability. When you apply tags to your AWS resources and activate the tags, AWS adds this information to the Cost and Usage reports.

### Follow Mandatory Cost Tagging

An effective tagging strategy gives you improved visibility and monitoring, helps you create accurate chargeback and showback models, and extract more granular and precise insights into usage and spending by applications and teams. The following tag categories help you achieve these goals:

**Environment** Distinguishes among development, test, and production infrastructure. Specifying an environment tag reduces analysis time, post-processing, and the need to maintain a separate mapping file of production versus nonproduction accounts.

**Application ID** Identifies resources that are related to a specific application for easy tracking of spending changes and that turn off at the end of projects.

**Automation Opt-In/Opt-Out** Indicates whether a resource should be included in an automated activity such as starting, stopping, or resizing instances.

**Cost Center/Business Unit** Identifies the cost center or business unit associated with a resource, typically for cost allocation and tracking.

**Owner** Used to identify who is responsible for the resource. This is typically the technical owner. If needed, you can add a separate business owner tag. You can specify the owner as an email address. Using email addresses supports automated notifications to both the technical and business owners as required (for example, if the resource is a candidate for elasticity or right sizing).

### Tag on Creation

You can make tagging a part of your build process and automate it with AWS management tools, such as AWS Elastic Beanstalk and AWS OpsWorks.

The following AWS CLI sample adds two tags, `CostCenter` and `environment`, for an Amazon Machine Image (AMI) and an instance:

```
aws ec2 create-tags --resources ami-1a2b3c4d i-1234567890abcdef0 --tags  
Key=CostCenter,Value=123 Key=environment,Value=Production
```

You can execute management tasks at scale by listing resources with specific tags and then executing the appropriate actions. For example, you can list all the resources with the tag and value of `environment:test`; then, for each of the resources, delete or terminate the resource. This is useful for automating shutdown or removal of a test environment at the end of the working day. Running reports on tagged and, more importantly, untagged resources enables greater compliance with internal cost management policies.

## Enforce Tag Use

Using AWS Identity and Access Management (IAM) policies, you can enforce tag use to gain precise control over access to resources, ownership, and accurate cost allocation.

The following example policy allows a user to create an Amazon Elastic Block Store (Amazon EBS) volume only if the user applies the tags (Costcenter and environment) that are defined in the policy using the qualifier `ForAllValues`. If the user applies any tag that is not included in the policy, the action is denied. To enforce case sensitivity, use the condition `aws:TagKeys` as follows:

```
Effect: Allow
Action: 'ec2:CreateVolume'
Resource: 'arn:aws:ec2:us-east-1:123456789012:volume/*'
Condition:
  StringEquals:
    'aws:RequestTag/costcenter': '115'
    'aws:RequestTag/environment': prod
  'ForAllValues:StringEquals':
    'aws:TagKeys':
      - Costcenter
      - environment
```

## Tagging Tools

The following tools help you manage your tags:

**AWS Tag Editor**—Finds resources with search criteria (including missing and misspelled tags) and enables you to edit tags from the AWS Management Console

**AWS Config**—Identifies resources that do not comply with tagging policies

**Capital One's Cloud Custodian** (open source)—Ensures tagging compliance and remediation

## Reduce AWS Usage

Set a continuous practice to review your consumption of AWS resources, and understand the factors that contribute to cost. Use various AWS monitoring tools to provide visibility, control, and cost optimization. Implement the best practice of oversight to make sure that you are not overspending. Following the DevOps phase, use dashboards to view the estimated costs of your AWS usage, top services that you use most, and the proportion of your costs to which each service contributed. If your monthly bill increases, make sure that it is for the right reason (business growth) and not the wrong reason (waste).

## Delete Unnecessary EBS Volumes

Stopping an Amazon Elastic Compute Cloud (Amazon EC2) instance leaves any attached Amazon Elastic Block Store (Amazon EBS) volumes operational. You continue to incur charges for these volumes until you delete them.

## Stop Unused Instances

Stop instances used in development and production during hours when these instances are not in use and then start them again when their capacity is needed. Assuming a 50-hour workweek, you can save 70 percent of costs by automatically stopping dev/test/production instances during nonbusiness hours.

## Delete Idle Resources

Consider the following best practices to reduce costs associated with AWS idle resources, such as unattached Amazon EBS volumes and unused Elastic IP addresses:

- The easiest way to reduce operational costs is to turn off instances that are no longer being used. If you find instances that have been idle for more than two weeks, it's safe to stop or even terminate them.
- Terminating an instance, however, automatically deletes attached EBS volumes and requires effort to re-provision if the instance is needed again. If you decide to delete an EBS volume, consider storing a snapshot of the volume so that it can be restored later if needed.
- Spin up instances to test new ideas. If the ideas work, keep the instance for further refinement. If not, spin it down.
- An Elastic IP address does not incur charges as long as it is associated with an Amazon EC2 instance. If an Elastic IP address is not used, you can avoid charges by releasing the IP address. After you release an IP address, you cannot provision that same Elastic IP address again.

## Update Outdated Resources

As AWS releases new services and features, it is a best practice to review your existing architectural decisions to ensure that they remain cost effective and stay evergreen. As your requirements change, be aggressive in decommissioning resources, components, and workloads that you no longer require.

## Delete Unused Keys

Each customer master key (CMK) that you create in AWS Key Management Service (AWS KMS), regardless of whether you use it with KMS-generated key material or key material imported by you, incurs a cost you until you delete it. Before deleting a CMK, you might want to know how many ciphertexts were encrypted under that key. Knowing how a CMK was used in the past might help you decide whether you will need it in the future by using AWS CloudTrail usage logs. After you are sure that you want to delete a CMK in AWS KMS, schedule the key deletion.

## Delete Old Snapshots

If your architecture suggests a backup policy that takes EBS volume snapshots daily or weekly, then you will quickly accumulate snapshots. To reduce storage costs, check for

“stale” snapshots—ones that are more than 30 days old—and delete them. Deleting a snapshot has no effect on the volume. You can use the AWS Management Console or AWS Command Line Interface (AWS CLI) for this purpose.

## Right Sizing

*Right sizing* is the process of matching instance types and sizes to performance and capacity requirements at the lowest possible cost. To achieve cost optimization, right sizing must become an ongoing process within your organization. Even if you right size workloads initially, performance and capacity requirements can change over time, which can result in underused or idle resources. New projects and workloads require additional cloud resources. Therefore, if there is no periodic check on right sizing, overprovisioning is the likely outcome.

AWS provides APIs, SDKs, and features that allow resources to be modified as demands change.

The following are examples of how you can change the instance type to match performance and capacity requirements:

- On Amazon Elastic Compute Cloud (Amazon EC2), you can perform a stop-and-start to allow a change of instance size or instance type.
- On Amazon EBS, you can increase volume size or adjust performance while volumes are still in use to improve performance through increased input/output operations per second (IOPS) or throughput or to reduce cost by changing the type of volume.

## Select the Right Use Case

As you monitor current performance, identify the following usage needs and patterns so that you can take advantage of potential right-sizing options:

**Steady state** The load remains constant over time, making forecasting simple. Consider using Reserved Instances to gain significant savings.

**Variable, but predictable** The load changes on a predictable schedule. Consider using AWS Auto Scaling.

**Dev, test, production** Development, testing, and production environments can usually be turned off outside of work hours.

**Temporary** Temporary workloads that have flexible start times and can be interrupted are good candidates for Spot Instances instead of On-Demand Instances.

## Select the Right Instance Family

When you launch an instance, the instance type that you specify determines the hardware of the host computer used for your instance. Each instance type offers different compute,

memory, and storage capabilities, and they are grouped in instance families based on these capabilities. Depending on the AWS offering, you can determine the right instance family for your infrastructure.

## Amazon Elastic Cloud Compute

Amazon Elastic Cloud Compute (Amazon EC2) provides a wide selection of instances, which gives you flexibility to right size CPU and memory needs for your compute resources to match capacity needs at the lowest cost. Following are the different options for CPU, memory, and network resources:

**General purpose (includes A1, T2, M3, and M4 instance types)** A1 instances deliver significant cost savings and are ideally suited for scale-out workloads, such as web servers, containerized microservices, caching fleets, and distributed data stores. T2 instances are a low-cost option that provides a small amount of CPU resources that can be increased in short bursts when additional cycles are available. They are well suited for lower throughput applications, such as administrative applications or low-traffic websites. M3 and M4 instances provide a balance of CPU, memory, and network resources, and they are ideal for running small and midsize databases, more memory-intensive data processing tasks, caching fleets, and backend servers.

**Compute optimized (includes the C3 and C4 instance types)** This family has a higher ratio of virtual CPUs to memory than the other families and the lowest cost per virtual CPU of all of the Amazon EC2 instance types. Consider compute-optimized instances first if you are running CPU-bound, scale-out applications, such as front-end fleets for high-traffic websites, on-demand batch processing, distributed analytics, web servers, video encoding, and high-performance science and engineering applications.

**Memory optimized (includes the X1, R3, and R4 instance types)** Designed for memory-intensive applications, these instances have the lowest cost per GiB of RAM of all Amazon EC2 instance types. Use these instances if your application is memory-bound.

**Storage optimized (includes the I3 and D2 instance types)** Optimized to deliver tens of thousands of low-latency, random input/output operations per second (IOPS) to applications. Storage-optimized instances are best for large deployments of NoSQL databases. I3 instances are designed for I/O-intensive workloads and equipped with super-efficient NVMe SSD storage. These instances can deliver up to 3.3 million IOPS in 4-KB blocks and up to 16 GB per second of sequential disk throughput. D2 or dense storage instances are designed for workloads that require high sequential read and write access to large datasets such as Hadoop, distributed computing, massively parallel processing data warehousing, and log-processing applications.

**Accelerated computing (includes the P2, G3, and F1 instance types)** Provides access to hardware-based compute accelerators, such as graphics processing units (GPUs) or field programmable gate arrays (FPGAs). Accelerated-computing instances enable more parallelism for higher throughput on compute-intensive workloads.

## Amazon Relational Database Service

Similar to Amazon EC2 instances, Amazon Relational Database Service (Amazon RDS) provides options to choose from database instances that are optimized for memory, performance, and I/O.

**Standard performance (includes the M3 and M4 instance types)** Designed for general-purpose database workloads that do not run many in-memory functions. This family has the most options for provisioning increased IOPS.

**Burstable performance (includes T2 instance types)** For workloads that require burstable performance capacity.

**Memory optimized (includes the R3 and R4 instance types)** Optimized for in-memory functions and big data analysis.

## Select the Right Instance Compatibility

You can right size an instance by migrating to a different model within the same instance family or by migrating to another instance family. When you're migrating within the same instance family, consider vCPU, memory, network throughput, and ephemeral storage.

**Virtualization type** The instances must have the same Linux Amazon Machine Image (AMI) virtualization type (PV AMI versus HVM) and platform (Amazon EC2-Classic versus Amazon EC2-VPC).

**Network** Instances unsupported in Amazon EC2-Classic must be launched in a virtual private cloud (VPC).

**Platform** If the current instance type supports 32-bit AMIs, make sure to select a new instance type that also supports 32-bit AMIs (not all Amazon EC2 instance types do).

## Using Instance Reservations

Amazon EC2 provides several purchasing options to enable you to optimize your costs based on your needs.

### AWS Pricing for Reserved Instances

Amazon EC2 Reserved Instances allow you to commit to usage parameters. To unlock an hourly rate that is up to 75 percent lower than On-Demand pricing, you can commit to a one-year or three-year duration at the time of purchase.

There are three payment options for Reserved Instances:

**No Upfront** No upfront payment is required, and Reserved Instances are billed monthly. This requires a good payment history with AWS.

**Partial Upfront** A portion of the cost is paid upfront, and the remaining hours in the term are billed at a discounted hourly rate, regardless of whether the RI is being used.

**All Upfront** Full payment is made at the start of the term, with no other costs or additional hourly charges incurred for the remainder of the term, regardless of hours used.

## Amazon EC2 Reservations

Amazon EC2 Reserved Instances provide a reservation of resources and capacity when used in a specific Availability Zone within an AWS Region:

- With Reserved Instances, you commit to a period of usage (one or three years) and save up to 75 percent over equivalent On-Demand hourly rates.
- For applications that have steady state or predictable usage, Reserved Instances can provide significant savings compared to using On-Demand Instances, without requiring a change to your workload.

## Convertible Reserved Instances

Convertible Reserved Instances are provided for a one-year or three-year term, and they enable conversion to different families, new pricing, different instance sizes, different platforms, or tenancy during the period. Use Convertible Reserved Instances when you are uncertain about instance needs in the future, but you are still committed to using Amazon EC2 instances for a three-year term in exchange for a significant discount.

Suppose that you own an Amazon EC2 Reserved Instance for a c4.8xlarge for three years. This Reserved Instance applies to any usage of a Linux/Unix c4 instance with shared tenancy in the same region as the Reserved Instance, such as 1 c4.8xlarge instance, 2 c4.4xlarge instances, or 16 c4.large instances, during this term. This adds flexibility to match the new needs of your workloads:

- There are no limits to how many times you perform an exchange, as long as the target Convertible Reserved Instance is of an equal or higher value than the Convertible Reserved Instances that you are exchanging.
- Exchanging Convertible Reserved Instances is free of charge, but you might need to pay a true-up cost if the value is lower than the value of the Reserved Instances for which you're exchanging. For example, you can convert C3 Reserved Instances to C4 Reserved Instances to take advantage of a newer instance type, or you can convert C4 Reserved Instances to M4 Reserved Instances if your application requires more memory. You can also use Convertible Reserved Instances to take advantage of Amazon EC2 price reductions over time.

## Reserved Instance Marketplace

Use the *Reserved Instance Marketplace* to sell your unused Reserved Instances and buy Reserved Instances from other AWS customers. As your needs change throughout the

course of your term, the AWS Marketplace provides an option to buy Reserved Instances for shorter terms and with a wider selection of prices.

## Amazon Relational Database Service Reservations

Reserved DB instances are not physical instances; they are a billing discount applied to the use of certain on-demand DB instances in your account. Discounts for reserved DB instances are tied to instance type and AWS Region.

All Reserved Instance types are available for Amazon Aurora, MySQL, MariaDB, PostgreSQL, Oracle, and SQL Server database engines.

- Reserved Instances can also provide significant cost savings for mission-critical applications that run on Multi-AZ database deployments for higher availability and data durability. Reserved Instances can minimize your costs up to 69 percent over On-Demand rates when used in steady state.
- Most production applications require database servers to be available 24/7. Consider using Reserved Instances to gain substantial savings if you are currently using On-Demand Instances.
- Any usage of running DB instances that exceeds the number of applicable Reserved Instances you have purchased are charged the On-Demand rate. For example, if you own three Reserved Instances with the same database engine and instance type (or instance family, if size flexibility applies) in a given region, the billing system checks each hour to determine how many total instances you have running that match those parameters. If it is three or fewer, you are charged the Reserved Instance rate for each instance running that hour. If more than three are running, you are charged the On-Demand rate for the additional instances.
- With size flexibility, your Reserved Instance's discounted rate is automatically applied to usage of any size in the instance family (using the same database engine) for the MySQL, MariaDB, PostgreSQL, and Amazon Aurora database engines and the "Bring your own license" (BYOL) edition of the Oracle database engine. For example, suppose that you purchased a db.m4.2xlarge MySQL Reserved Instance in US East (N. Virginia). The discounted rate of this Reserved Instance can automatically apply to two db.m4.xlarge MySQL instances without you needing to do anything.
- The Reserved Instance discounted rate also applies to usage of both Single-AZ and Multi-AZ configurations for the same database engine and instance family.

Suppose that you purchased a db.r3.large PostgreSQL Single-AZ Reserved Instance in EU (Frankfurt). The discounted rate of this Reserved Instance can automatically apply to 50 percent of the usage of a db.r3.large PostgreSQL Multi-AZ instance in the same region.

# Using Spot Instances

Amazon EC2 Spot Instances offer spare compute capacity in the AWS Cloud at steep discounts compared to On-Demand Instances.

You can use Spot Instances to save up to 90 percent on stateless web applications, big data, containers, continuous integration/continuous delivery (CI/CD), high performance computing (HPC), and other fault-tolerant workloads. Or, scale your workload throughput by up to 10 times and stay within the existing budget.

## Spot Fleets

Use *Spot Fleets* to request and manage multiple Spot Instances automatically, which provides the lowest price per unit of capacity for your cluster or application, such as a batch-processing job, a Hadoop workflow, or an HPC grid computing job. You can include the instance types that your application can use. You define a target capacity based on your application needs (in units, including instances, vCPUs, memory, storage, or network throughput) and update the target capacity after the fleet is launched. Spot Fleets enable you to launch and maintain the target capacity and to request resources automatically to replace any that are disrupted or manually terminated.

To ensure that you have instance capacity, you can include a request for On-Demand capacity in your Spot Fleet request. If there is capacity, the On-Demand request is fulfilled. If there is capacity and availability, the balance of the target capacity is fulfilled as Spot.

The following example specifies the desired target capacity as 10, of which 5 must be On-Demand capacity. Spot capacity is not specified; it is implied in the balance of the target capacity minus the On-Demand capacity. If there is available Amazon EC2 capacity and availability, Amazon EC2 launches 5 capacity units as On-Demand and 5 capacity units ( $10-5=5$ ) as Spot.

```
{  
    "IamFleetRole": "arn:aws:iam::1234567890:role/aws-ec2-spot-fleet-tagging-role",  
    "AllocationStrategy": "lowestPrice",  
    "TargetCapacity": 10,  
    "SpotPrice": null,  
    "ValidFrom": "2018-04-04T15:58:13Z",  
    "ValidUntil": "2019-04-04T15:58:13Z",  
    "TerminateInstancesWithExpiration": true,  
    "LaunchSpecifications": [],  
    "Type": "maintain",  
    "OnDemandTargetCapacity": 5,  
    "LaunchTemplateConfigs": [  
        {  
            "LaunchTemplateSpecification": {  
                "LaunchTemplateId": "lt-0dbb04d4a6abcabcabc",  
                "Version": "1"  
            }  
        }  
    ]  
}
```

```
"Version": "2"
},
"Overrides": [
{
"InstanceType": "t2.medium",
"WeightedCapacity": 1,
"SubnetId": "subnet-d0dc51fb"
}
]
}
```

## Amazon EC2 Fleets

With a single API call, *Amazon EC2 Fleet* enables you to provision compute capacity across different instance types, Availability Zones, and across On-Demand, Reserved Instances, and Spot Instances purchase models to help optimize scale, performance, and cost.

By default, Amazon EC2 Fleet launches the On-Demand option that is at the lowest price. For Spot Instances, Amazon EC2 Fleet provides two allocation strategies: lowest price and diversified. The lowest-price strategy allows you to provision Spot Instances in pools that provide the lowest price per unit of capacity at the time of the request. The diversified strategy allows you to provision Spot Instances across multiple Spot pools, and you can maintain your fleet's target capacity to increase application.

## Design for Continuity

With Spot Instances, you avoid paying more than the maximum price you specified. If the Spot price exceeds your maximum willingness to pay for a given instance or when capacity is no longer available, your instance is terminated automatically (or stopped or hibernated, if you opt for this behavior on a persistent request).

Spot offers features, such as termination notices, persistent requests, and spot block duration, to help you better track and control when Spot Instances can run and terminate (or stop or hibernate).

## Using Termination Notices

If you need to save state, upload final log files, or remove Spot Instances from an Elastic Load Balancing load balancer before interruption, you can use termination notices, which are issued 2 minutes before interruption.

If your instance is marked for termination, the termination notice is stored in the instance's metadata 2 minutes before its termination time. The notice is accessible at <http://169.254.169.254/latest/meta-data/spot/termination-time>. The notice includes the time when the shutdown signal will be sent to the instance's operating system.

Relevant applications on Spot Instances should poll for the termination notice at 5-second intervals, which gives the application almost the entire 2 minutes to complete any needed processing before the instance is terminated and taken back by AWS.

## Using Persistent Requests

You can set your request to remain open so that a new instance is launched in its place when the instance is interrupted. You can also have your Amazon EBS-backed instance stopped upon interruption and restarted when Spot has capacity at your preferred price.

## Using Block Durations

You can also launch Spot Instances with a fixed duration (Spot blocks, 1–6 hours), which are not interrupted as the result of changes in the Spot price. Spot blocks can provide savings of up to 50 percent.

You submit a Spot Instance request and use the new `BlockDuration` parameter to specify the number of hours that you want your instances to run, along with the maximum price that you are willing to pay.

You can submit a request of this type by running the following command:

```
$ aws ec2 request-spot-instances \
    block-duration-minutes 360 \
    instance-count 2 \
    spot-price "0.25"
:
```

Alternatively, you can call the `RequestSpotInstances` function.

## Minimizing the Impact of Interruptions

Because the Spot service can terminate Spot Instances without warning, it is important to build your applications in a way that allows you to make progress even if your application is interrupted. There are many ways to accomplish this, including the following:

**Adding checkpoints** Add checkpoints that save your work periodically, for example, to an Amazon EBS volume. Another approach is to launch your instances from Amazon EBS-backed AMI.

**Splitting up the work** By using Amazon Simple Queue Service (Amazon SQS), you can queue up work increments and track work that has already been done.

# Using AWS Auto Scaling

Using AWS Auto Scaling, you can scale workloads in your architecture. It automatically increases the number of resources during the demand spikes to maintain performance and decreases capacity when demand lulls to reduce cost. AWS Auto Scaling is well-suited for

applications that have stable demand patterns and for ones that experience hourly, daily, or weekly variability in usage. AWS Auto Scaling is useful for applications that show steady demand patterns and that experience frequent variations in usage.

## Amazon EC2 Auto Scaling

*Amazon EC2 Auto Scaling* helps you scale your Amazon EC2 instances and Spot Fleet capacity up or down automatically according to conditions that you define. AWS Auto Scaling is generally used with Elastic Load Balancing to distribute incoming application traffic across multiple Amazon EC2 instances in an AWS Auto Scaling group. AWS Auto Scaling is triggered using scaling plans that include policies that define how to scale (manual, schedule, and demand spikes) and the metrics and alarms to monitor in Amazon CloudWatch.

CloudWatch metrics are used to trigger the scaling event. These metrics can be standard Amazon EC2 metrics, such as CPU utilization, network throughput, Elastic Load Balancing observed request and response latency, and even custom metrics that might originate from application code on your Amazon EC2 instances.

You can use Amazon EC2 Auto Scaling to increase the number of Amazon EC2 instances automatically during demand spikes to maintain performance and decrease capacity during lulls to reduce costs.

## Dynamic Scaling

The *dynamic scaling* capabilities of Amazon EC2 Auto Scaling refers to the functionality that automatically increases or decreases capacity based on load or other metrics. For example, if your CPU spikes above 80 percent (and you have an alarm set up), Amazon EC2 Auto Scaling can add a new instance dynamically, reducing the need to provision Amazon EC2 capacity manually in advance. Alternatively, you could set a target value by using the new Request Count Per Target metric from Application Load Balancer, a load balancing option for the Elastic Load Balancing service. Amazon EC2 Auto Scaling will then automatically adjust the number of Amazon EC2 instances as needed to maintain your target.

## Scheduled Scaling

Scaling based on a schedule allows you to scale your application ahead of known load changes, such as the start of business hours, thus ensuring that resources are available when users arrive, or in typical development or test environments that run only during defined business hours or periods of time.

You can use APIs to scale the size of resources within an environment (vertical scaling). For example, you could scale up a production system by changing the instance size or class. This can be achieved by stopping and starting the instance and selecting the different instance size or class. You can also apply this technique to other resources, such as EBS volumes, which can be modified to increase size, adjust performance (IOPS), or change the volume type while in use.

## Fleet Management

*Fleet management* refers to the functionality that automatically replaces unhealthy instances in your application, maintains your fleet at the desired capacity, and balances instances across Availability Zones. Amazon EC2 Auto Scaling fleet management ensures that your application is able to receive traffic and that the instances themselves are working properly. When AWS Auto Scaling detects a failed health check, it can replace the instance automatically.

## Instances Purchasing Options

With Amazon EC2 Auto Scaling, you can provision and automatically scale instances across purchase options, Availability Zones, and instance families in a single application to optimize scale, performance, and cost. You can include Spot Instances with On-Demand and Reserved Instances in a single AWS Auto Scaling group to save up to 90 percent on compute. You have the option to define the desired split between On-Demand and Spot capacity, select which instance types work for your application, and specify preferences for how Amazon EC2 Auto Scaling should distribute the AWS Auto Scaling group capacity within each purchasing model.

## Golden Images

A *golden image* is a snapshot of a particular state of a resource, such as an Amazon EC2 instance, Amazon EBS volumes, and an Amazon RDS DB instance. You can customize an Amazon EC2 instance and then save its configuration by creating an Amazon Machine Image (AMI). You can launch as many instances from the AMI as you need, and they will all include those customizations. A golden image results in faster start times and removes dependencies to configuration services or third-party repositories. This is important in auto-scaled environments in which you want to be able to launch additional resources in response to changes in demand quickly and reliably.

## AWS Auto Scaling

*AWS Auto Scaling* monitors your applications and automatically adjusts capacity of all scalable resources to maintain steady, predictable performance at the lowest possible cost. Using AWS Auto Scaling, you can set up application scaling for multiple resources across multiple services in minutes.

AWS Auto Scaling automatically scales resources for other AWS services, including Amazon ECS, Amazon DynamoDB, Amazon Aurora, Amazon EC2 Spot Fleet requests, and Amazon EC2 Scaling groups.

If you have an application that uses one or more scalable resources and experiences variable load, use AWS Auto Scaling. A good example would be an ecommerce web application that receives variable traffic throughout the day. It follows a standard three-tier architecture with Elastic Load Balancing for distributing incoming traffic, Amazon EC2 for the compute layer, and Amazon DynamoDB for the data layer. In this case, AWS Auto Scaling scales one or more Amazon EC2 Auto Scaling groups and DynamoDB tables that are powering the application in response to the demand curve.

AWS Auto Scaling continually monitors your applications to make sure that they are operating at your desired performance levels. When demand spikes, AWS Auto Scaling automatically increases the capacity of constrained resources so that you maintain a high quality of service.

AWS Auto Scaling bases its scaling recommendations on the most popular scaling metrics and thresholds used for AWS Auto Scaling. It also recommends safe guardrails for scaling by providing recommendations for the minimum and maximum sizes of the resources. This way, you can get started quickly and then fine-tune your scaling strategy over time, allowing you to optimize performance, costs, or balance between them.

The *predictive scaling* feature uses machine learning algorithms to detect changes in daily and weekly patterns, automatically adjusting their forecasts. This removes the need for the manual adjustment of AWS Auto Scaling parameters as cyclical changes over time, making AWS Auto Scaling simpler to configure, and provides more accurate capacity provisioning. Predictive scaling results in lower cost and more responsive applications.

## DynamoDB Auto Scaling

*DynamoDB automatic scaling* uses the AWS Auto Scaling service to adjust provisioned throughput capacity dynamically on your behalf in response to actual traffic patterns. This enables a table or a global secondary index to increase its provisioned read and write capacity to handle sudden increases in traffic without throttling. When the workload decreases, AWS Auto Scaling decreases the throughput so that you don't pay for unused provisioned capacity.

## Amazon Aurora Auto Scaling

*Amazon Aurora automatic scaling* dynamically adjusts the number of Aurora Replicas provisioned for an Aurora DB cluster. Aurora automatic scaling is available for both Aurora MySQL and Aurora PostgreSQL. Aurora automatic scaling enables your Aurora DB cluster to handle sudden increases in connectivity or workload. When the connectivity or workload decreases, Aurora automatic scaling removes unnecessary Aurora Replicas so that you don't pay for unused provisioned DB instances.

*Amazon Aurora Serverless* is an on-demand, automatic scaling configuration for the MySQL-compatible edition of Amazon Aurora. An Aurora Serverless DB cluster automatically starts up, shuts down, and scales capacity up or down based on your application's needs. Aurora Serverless provides a relatively simple, cost-effective option for infrequent, intermittent, or unpredictable workloads.

## Accessing AWS Auto Scaling

There are several ways to get started with AWS Auto Scaling. You can set up AWS Auto Scaling through the AWS Management Console, with the AWS CLI, or with AWS SDKs.

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

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

# Using Containers

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

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

## Containerize Everything

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

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

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

## Containers without Servers

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

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

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

## Using Serverless Approaches

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

Benefits gained by using AWS Serverless services include the following:

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

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

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

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

Here are a few examples of application use cases:

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

- IT automation
- AWS IoT Core backends
- Web hooked systems
- Chatbots
- Clickstream and other near real-time streaming data processes

## Optimize Lambda Usage

*AWS Lambda* provides the cloud-logic layer, and with Lambda you can run code for virtually any type of application or backend service, all with zero administration. A variety of events can trigger Lambda functions, enabling developers to build reactive, event-driven systems without managing infrastructure. When there are multiple, simultaneous events, Lambda scales by running more copies of the function in parallel, responding to each individual trigger. As a result, there is no possibility of an idle server or container. The problem of wasted infrastructure expenditures is eliminated by design in architectures that use Lambda functions.

Serverless applications are typically composed of one or more Lambda functions; therefore, monitor the execution duration and configuration of your functions closely.

Consider the following recommendations for optimizing Lambda functions:

**Optimal memory size** The memory usage for your function is determined per invocation and can be viewed in CloudWatch Logs. By analyzing the Max Memory Used: field in the Invocation report, you can determine whether your function needs more memory or whether you over-provisioned your function's memory size.

**Language runtime performance** If your application use case is both latency-sensitive and susceptible to incurring the initial invocation cost frequently (spiky traffic or infrequent use), then recommend one of the interpreted languages, such as Node.js or Python.

**Optimizing code** Much of the application performance depends on your logic and dependencies. Pay attention to reusing the objects and using global/static variables. Keep live or reuse HTTP/session connections, and use default network environments as much as possible.

# Optimizing Storage

AWS storage services are optimized for different storage scenarios—there is no single data storage option that is ideal for all workloads. When evaluating your storage requirements, consider data storage options for each workload separately.

To optimize the storage, you must first understand the performance levels of your workloads. Conduct a performance analysis to measure input/output operations per second, throughput, quick access to your data, durability, sensitivity, size, and budget.

Amazon offers three broad categories of storage services: object, block, and file storage. Each offering is designed to meet a different storage requirement, which gives you flexibility to find the solution that works best for your storage scenarios.

## Object Storage

*Amazon Simple Storage Service (Amazon S3)* is highly durable, general-purpose object storage that works well for unstructured datasets such as media content.

There are multiple tiers of storage: hot, warm, or cold data. In terms of pricing, the colder the data, the cheaper it is to store, and the costlier it is to access when needed.

**Standard (STANDARD)** This is the best storage option for data that you frequently access. Amazon S3 delivers low latency and high throughput, and it is ideal for use cases such as cloud applications, dynamic websites, content distribution, gaming, and data analytics.

**Amazon S3 Standard – Infrequent Access (STANDARD\_IA)** Use this storage option for data that you access less frequently, such as long-term backups and disaster recovery. It offers cheaper storage over time, but higher charges to retrieve or transfer data.

**Amazon S3 Intelligent-Tiering (INTELLIGENT\_TIERING)** This storage class is designed to optimize the cost by moving data to the most cost-effective access tier automatically without degrading the performance of the application. If an object in the infrequent access tier is accessed, it is automatically moved back to the frequent access tier.

**Amazon S3 One Zone-Infrequent Access (ONEZONE\_IA)** This storage class provides a lower-cost option for infrequently accessed data that requires rapid access. The data is stored in only one Availability Zone (AZ), and it saves up to 20 percent of storage costs as compared to STANDARD\_IA. Use this option for storing secondary backups of on-premises data or data that can be easily recreated.

**Amazon S3 Glacier (GLACIER)** This option is designed for long-term storage of infrequently accessed data, such as end-of-lifecycle, compliance, or regulatory backups. Different methods of data retrieval are available at various speeds and cost. Retrieval can take from a few minutes to several hours.

**Amazon S3 Glacier Deep Archive (DEEP\_ARCHIVE)** This is the lowest-cost class designed for long-term retention of rarely accessed data. Data will be retained for 7–10 years and may be accessed about once or twice a year. When you need the data, you can retrieve it within 12 hours. This storage is ideal for maintaining backups of historical regulatory or compliance data and disaster recovery backups.

## Block Storage

*Amazon Elastic Block Store (Amazon EBS)* volumes provide a durable block-storage option for use with Amazon EC2 instances. Use Amazon EBS for data that requires

long-term persistence and quick access at assured levels of performance. There are two types of block storage: solid-state drive (SSD) storage and hard disk drive (HDD) storage.

*SSD storage* is optimized for transactional workloads wherein performance is closely tied to IOPS. Choose from two SSD volume options:

**General Purpose SSD (gp2)** Designed for general use and offers a balance between cost and performance.

**Provisioned IOPS SSD (io1)** Best for latency-sensitive workloads that require specific minimum-guaranteed IOPS. With io1 volumes, you pay separately for Provisioned IOPS, so unless you need high levels of Provisioned IOPS, gp2 volumes are a better match at lower cost.

*HDD storage* is designed for throughput-intensive workloads, such as data warehouses and log processing. There are two types of HDD volumes:

**Throughput Optimized HDD (st1)** Best for frequently accessed, throughput-intensive workloads.

**Cold HDD (sc1)** Designed for less frequently accessed, throughput-intensive workloads.

## File Storage

*Amazon Elastic File System (Amazon EFS)* provides simple, scalable file storage for use with Amazon EC2 instances. Amazon EFS supports any number of instances at the same time. Amazon EFS is designed for workloads and applications such as big data, media-processing workflows, content management, and web serving.

Amazon S3 and Amazon EFS allocate storage based on your usage, and you pay for what you use. However, for EBS volumes, you are charged for provisioned (allocated) storage regardless of whether you use it or not. The key to keeping storage costs low without sacrificing required functionality is to maximize the use of Amazon S3 when possible and use more expensive EBS volumes with provisioned I/O only when application requirements demand it.

## Optimize Amazon S3

Perform analysis on data access patterns, create inventory lists, and configure lifecycle policies. Identifying the right storage class and moving less frequently accessed Amazon S3 data to cheaper storage tiers yields considerable savings. For example, by moving data from the STANDARD to STANDARD\_IA storage class, you can save up to 60 percent (on a per-gigabyte basis) of Amazon S3 pricing. By moving data that is at the end of its lifecycle and accessed on rare occasions from Amazon S3 Glacier, you can save up to 80 percent of Amazon S3 pricing.

## Storage Management Tools/Features

The following sections detail some of the tools that help to determine when to transition data to another storage class.

## Cost Allocation S3 Bucket Tags

To track the storage cost or other criteria for individual projects or groups of projects, label your Amazon S3 buckets using cost allocation tags. A *cost allocation tag* is a key-value pair that you associate with an S3 bucket. To manage storage data most effectively, you can use these tags to categorize your S3 objects and filter on these tags in your data lifecycle policies.

## Amazon S3 Analytics: Storage Class Analysis

Use this feature to analyze storage access patterns to help you decide when to transition the right data to the right storage class. This feature observes data access patterns to help you determine when to transition less frequently accessed STANDARD storage to the STANDARD\_IA storage class.

After storage class analysis observes the infrequent access patterns of a filtered set of data over a period of time, you can use the analysis results to help you improve your lifecycle policies. You can configure storage class analysis to analyze all the objects in a bucket. Alternatively, you can configure filters to group objects together for analysis by common prefix (that is, objects that have names that begin with a common string), by object tags, or by both prefix and tags. You'll most likely find that filtering by object groups is the best way to benefit from storage class analysis.

You can use the Amazon S3 console, the `s3:PutAnalyticsConfiguration` REST API, or the equivalent from the AWS CLI or AWS SDKs to configure storage class analysis.

## Amazon S3 Inventory

This tool audits and reports on the replication and encryption status of your S3 objects on a weekly or monthly basis. This feature provides CSV output files that list objects and their corresponding metadata, and it lets you configure multiple inventory lists for a single bucket, organized by different Amazon S3 metadata tags. You can also query Amazon S3 inventory through standard SQL by using Amazon Athena, Amazon Redshift Spectrum, and other tools, such as Presto, Apache Hive, and Apache Spark.

## Amazon CloudWatch

Amazon S3 can also publish storage, request, and data transfer metrics to Amazon CloudWatch. Storage metrics are reported daily, are available at one-minute intervals for granular visibility, and can be collected and reported for an entire bucket or a subset of objects (selected via prefix or tags).

## Use Amazon S3 Select

*Amazon S3 Select* enables applications to retrieve only a subset of data from an object by using simple SQL expressions. By using Amazon S3 Select to retrieve only the data needed by your application, you can achieve drastic performance increases—in many cases, you can get as much as a 400 percent improvement.

Following is a Python sample code snippet that shows how to retrieve columns from an object containing data in CSV format. This code snippet retrieves the city and airport code,

where country name is similar to “United States.” If you have column headers and you set the `FileHeaderInfo` to `Use`, you can identify columns by name in the SQL expression.

```
result = s3.select_object_content(  
    Bucket='example-bucket-us-west-2',  
    Key='sample-data/airportCodes.csv',  
    ExpressionType='SQL',  
    Expression="select s.city, s.code from s3object s where" \  
        "s.\\"Country (Name)\" like '%United States%'",  
    InputSerialization = {'CSV': {"FileHeaderInfo": "Use"}},  
    OutputSerialization = {'CSV': {}},  
    :  
:
```

## Use Amazon Glacier Select

*Amazon Glacier Select* unlocks an opportunity to query your archived data easily. With Glacier Select, you can filter directly against an Amazon S3 Glacier object by using standard SQL statements.

It works like any other retrieval job, except for having an additional set of parameters (`SelectParameters`) that you can pass in an initiate job request.

The following is an example of a Python code snippet that shows how to pass an SQL expression under `SelectParameters`:

```
jobParameters = {  
    "Type": "select", "ArchiveId": "ID",  
    "Tier": "Expedited",  
    "SelectParameters": {  
        "InputSerialization": {"csv": {}},  
        "ExpressionType": "SQL",  
        "Expression": "SELECT * FROM archive WHERE _5='498960'",  
        "OutputSerialization": {  
            "csv": {}  
        }  
    }  
}
```

With both Amazon S3 Select and Glacier Select, you can lower your costs and uncover more insights from your data, regardless of which storage tier it is in.

## Optimize Amazon EBS

With Amazon EBS, you are paying for provisioned capacity and performance—even if the volume is unattached or has low write activity. To optimize storage performance and costs for Amazon EBS, monitor volumes periodically to identify ones that are unattached or appear to be underutilized or overutilized, and adjust provisioning to match actual usage.

## Check Configuration

Follow these configuration guidelines

- To achieve the best performance consistently, launch instances as EBS optimized. For instances that are not EBS-optimized by default, you can enable EBS optimization when you launch the instances or enable EBS optimization after the instances are running.  
To enable this feature, you can use either the Amazon EC2 console or AWS Tools. For AWS CLI, use `ebs-optimized` with the command `run-instances` to enable EBS optimization when launching and with the command `modify-instance-attribute` to enable EBS optimization for a running instance.
- Choose an EBS-optimized instance that provides more dedicated EBS throughput than your application needs; otherwise, the Amazon EBS to Amazon EC2 connection becomes a performance bottleneck.
- New EBS volumes receive their maximum performance the moment that they are available and do not require initialization. However, storage blocks on volumes that were restored from snapshots must be initialized before you can access the block. This preliminary action takes time and can cause a significant increase in the latency of an I/O operation the first time each block is accessed.
- To achieve a higher level of performance for a file system than you can provision on a single volume, create a RAID 0 (zero) array. Consider using RAID 0 when I/O performance is more important than fault tolerance. For example, you could use it with a heavily used database where data replication is already set up separately.

## Use Monitoring Tools

AWS offers tools that help you optimize block storage.

### Amazon CloudWatch

Amazon CloudWatch automatically collects a range of data points for EBS volumes, and you can then set alarms on volume behavior.

Consider the following important metrics:

**BurstBalance** When your burst bucket is depleted, volume I/O credits (for gp2 volumes) or volume throughput credits (for st1 and sc1 volumes) are throttled to the baseline. Check the `BurstBalance` value to determine whether your volume is being throttled for this reason.

**VolumeQueueLength** If your I/O latency is higher than you require, check `VolumeQueueLength` to make sure that your application is not trying to drive more IOPS than you have provisioned. If your application requires a greater number of IOPS than your volume can provide, consider using a larger gp2 volume with a higher base performance level or an io1 volume with more Provisioned IOPS to achieve faster latencies.

**VolumeReadBytes, VolumeWriteBytes, VolumeReadOps, VolumeWriteOps** HDD-backed st1 and sc1 volumes are designed to perform best with workloads that take advantage of

the 1,024 KiB maximum I/O size. To determine your volume's average I/O size, divide `VolumeWriteBytes` by `VolumeWriteOps`. The same calculation applies to read operations. If the average I/O size is below 64 KiB, increasing the size of the I/O operations sent to an `st1` or `sc1` volume should improve performance.

### AWS Trusted Advisor

*AWS Trusted Advisor* is another way for you to analyze your infrastructure to identify unattached, underutilized, and overutilized EBS volumes.

## Delete Unattached Amazon EBS Volumes

To find unattached EBS volumes, look for volumes that are *available*, which indicates that they are not attached to an Amazon EC2 instance. You can also look at network throughput and IOPS to determine whether there has been any volume activity over the previous two weeks, or you can look up the last time the EBS volume was attached. If the volume is in a nonproduction environment, hasn't been used in weeks, or hasn't been attached in a month, there is a good chance that you can delete it.

Before deleting a volume, store an Amazon EBS snapshot (a backup copy of an EBS volume) so that the volume can be quickly restored later if needed.

## Resize or Change the EBS Volume Type

Identify volumes that are underutilized and downsize them or change the volume type. Monitor the read/write access of EBS volumes to determine whether throughput is low. If you have a current-generation EBS volume attached to a current-generation Amazon EC2 instance type, you can use the elastic volumes feature to change the size or volume type or (for an SSD `io1` volume) adjust IOPS performance without detaching the volume.

Follow these tips:

- For General Purpose SSD `gp2` volumes, optimize for capacity so that you're paying only for what you use.
- With Provisioned IOPS SSD `io1` volumes, pay close attention to IOPS utilization rather than throughput, since you pay for IOPS directly. Provision 10–20 percent above maximum IOPS utilization.
- You can save by reducing Provisioned IOPS or by switching from a Provisioned IOPS SSD `io1` volume type to a General Purpose SSD `gp2` volume type.
- If the volume is 500 GB or larger, consider converting to a Cold HDD `sc1` volume to save on your storage rate.

## Delete Stale Amazon EBS Snapshots

If you have a backup policy that takes EBS volume snapshots daily or weekly, you will quickly accumulate snapshots. Check for stale snapshots that are more than 30 days old and delete them to reduce storage costs. Deleting a snapshot has no effect on the volume.

# Optimizing Data Transfer

Optimizing data transfer ensures that you minimize data transfer costs. Review your user presence if global or local and how the data gets located in order to reduce the latency issues.

- Use *Amazon CloudFront*, a global content delivery network (CDN), to locate data closer to users. It caches data at edge locations across the world, which reduces the load on your resources. By using CloudFront, you can reduce the administrative effort in delivering content automatically to large numbers of users globally, with minimum latency. Depending on your application types, distribute your entire website, including dynamic, static, streaming, and interactive content through CloudFront instead of scaling out your infrastructure.
- *Amazon S3 transfer acceleration* enables fast transfer of files over long distances between your client and your S3 bucket. Transfer acceleration leverages Amazon CloudFront globally distributed edge locations to route data over an optimized network path. For a workload in an S3 bucket that has intensive GET requests, you should use Amazon S3 with CloudFront.
- When uploading large files, use *multipart uploads* with multiple parts uploading at once to help maximize network throughput. Multipart uploads provide the following advantages:
  - *Improved throughput*—You can upload parts in parallel to improve throughput.
  - *Quick recovery from any network issues*—Smaller part size minimizes the impact of restarting a failed upload due to a network error.
  - *Pause and resume object uploads*—You can upload object parts over time. After you initiate a multipart upload, there is no expiry; you must explicitly complete or abort the multipart upload.
  - *Begin an upload before you know the final object size*—You can upload an object as you are creating it.
- Using *Amazon Route 53*, you can reduce latency for your users by serving their requests from the AWS Region for which network latency is lowest. Amazon Route 53 latency-based routing lets you use Domain Name System (DNS) to route user requests to the AWS Region that will give your users the fastest response.

## Caching

Caching improves application performance by storing frequently accessed data items in memory so that they can be retrieved without accessing the primary data store. Cached information might include the results of I/O-intensive database queries or the outcome of computationally intensive processing.

When the result set is not found in the cache, the application can calculate it or retrieve it from a database of expensive, slowly mutating third-party content and store it in the cache for subsequent requests.

## Amazon ElastiCache

Amazon ElastiCache is a web service that makes it easy to deploy, operate, and scale an in-memory cache in the cloud. It supports two open-source, in-memory caching engines: Memcached and Redis.

- The *Memcached* caching engine is popular for database query results caching, session caching, webpage caching, API caching, and caching of objects such as images, files, and metadata. Memcached is also a great choice to store and manage session data for internet-scale applications in cases wherein persistence is not critical.
- *Redis* caching engine is a great choice for implementing a highly available in-memory cache to decrease data access latency, increase throughput, and ease the load off your relational or NoSQL database and application. Redis has disk persistence built in, and you can use it for long-lived data.

*Lazy loading* is a good caching strategy whereby you populate the cache only when an object is requested by the application, keeping the cache size manageable. Apply a lazy caching strategy anywhere in your application where you have data that is going to be read often but written infrequently. In a typical web or mobile app, for example, a user's profile rarely changes but is accessed throughout the application.

## Amazon DynamoDB Accelerator (DAX)

*Amazon DynamoDB Accelerator (DAX)* is a fully managed, highly available, in-memory cache for Amazon DynamoDB. This feature delivers performance improvements from milliseconds to microseconds, for high throughput. DAX adds in-memory acceleration to your DynamoDB tables without requiring you to manage cache invalidation, data population, or clusters.

DAX is ideal for applications that require the fastest possible response time read operations but that are also cost-sensitive and require repeated reads against a large set of data. For example, consider an ecommerce system that has a one-day sale on a popular product that would sharply increase the demand or a long-running analysis of regional weather data that could temporarily consume all of the read capacity in a DynamoDB table. Naturally, these would negatively impact other applications that must access the same data.

# Relational Databases and Amazon DynamoDB

Traditional *relational database management system (RDBMS)* platforms store data in a normalized relational structure that reduces hierarchical data structures to a set of common elements that are stored across multiple tables.

RDBMS platforms use an ad hoc query language (generally a flavor of SQL) to generate or materialize views of the normalized data to support application-layer access patterns.

A relational database system does not scale well for the following reasons:

- It normalizes data and stores it on multiple tables that require multiple queries to write to disk.
- It generally incurs the performance costs of an Atomicity, Consistency, Isolation, Durability (ACID)-compliant transaction system.
- It uses expensive joins to reassemble required views of query results.

For this reason, when your business requires a low-latency response to high-traffic queries, taking advantage of a NoSQL system generally makes technical and economic sense. Amazon DynamoDB helps solve the problems that limit relational system scalability by avoiding them.

DynamoDB scales well for these reasons:

- Schema flexibility lets Amazon DynamoDB store complex hierarchical data within a single item.
- Composite key design lets it store related items close together on the same table.

The following are some recommendations for maximizing performance and minimizing throughput costs when working with Amazon DynamoDB.

## Apply NoSQL Design

NoSQL design requires a different mind-set than RDBMS design. For an RDBMS, you can create a normalized data model without thinking about access patterns. You can then extend it later when new questions and query requirements arise.

NoSQL design is different:

- For DynamoDB, by contrast, design your schema after you know the questions it needs to answer. Understanding the business problems and the application use cases up front is essential.
- Maintain as few tables as possible in an Amazon DynamoDB application. Most well-designed applications require only one table.

## Keep Related Data Together

Keeping related data in proximity has a major impact on cost and performance. Instead of distributing related data items across multiple tables, keep related items in your NoSQL system as close together as possible.

## Keep Fewer Tables

In general, maintain as few tables as possible in an Amazon DynamoDB application. Most well-designed applications require only one table, unless there is a specific reason for using multiple tables.

## Distribute Workloads Evenly

The optimal usage of a table's provisioned throughput depends on the workload patterns of individual items and the partition key design.

### Designing Partition Keys

The more distinct partition key values that your workload accesses, the more those requests are spread across the partitioned space. In general, you use your provisioned throughput more efficiently as the ratio of partition key values accessed to the total number of partition key values increases.

Table 16.1 provides a comparison of the provisioned throughput efficiency of some common partition key schemas.

**TABLE 16.1** Samples of Partition Key Distributions

Partition Key Value	Uniformity
<i>User ID</i> , where the application has many users	Good
<i>Status code</i> , where there are only a few possible status codes	Bad
<i>Item creation date</i> , rounded to the nearest time period (for example, day, hour, or minute)	Bad
<i>Device ID</i> , where each device accesses data at relatively similar intervals	Good
<i>Device ID</i> , where even if there are many devices being tracked, one is by far more popular than all the others	Bad

If a single table has only a small number of partition key values, consider distributing your write operations across more distinct partition key values. Structure the primary key elements to avoid one “hot” (heavily requested) partition key value that slows the overall performance.

For example, consider a table with a composite primary key. The partition key represents the item's creation date, rounded to the nearest day. The sort key is an item identifier. On a given day, say 2014-07-09, all of the new items are written to that single partition key value (and corresponding physical partition).

If the table fits entirely into a single partition (considering growth of your data over time) and if your application's read and write throughput requirements don't exceed the read and write capabilities of a single partition, your application won't encounter any unexpected throttling because of partitioning.

### Implementing Write Sharding

One better way to distribute writes across a partition key space in DynamoDB is to expand the space. One strategy for distributing loads more evenly across a partition key space is

to add a random number or a calculated hash suffix to the end of the partition key values. Then you can randomize the writes across the larger space. A randomizing strategy can greatly improve write throughput.

For example, in the case of a partition key that represents today's date in the Order table, suppose that each item has an accessible OrderId attribute and that you most often need to find items by OrderId in addition to date. Before your application writes the item to the table, it could calculate a hash suffix based on the OrderId (similar to OrderId, modulo 200, + 1) and append it to the partition key date. The calculation might generate a number between 1 and 200 that is fairly evenly distributed, similar to what the random strategy produces.

With this strategy, the writes are spread evenly across the partition key values and thus across the physical partitions. You can easily perform a GetItem operation for a particular item and date because you can calculate the partition key value for a specific OrderId value.

## Upload Data Efficiently

Typically, when you load data from other data sources, Amazon DynamoDB partitions your table data on multiple servers. You get better performance if you upload data to all the allocated servers simultaneously.

For example, suppose that you want to upload user messages to a DynamoDB table that uses a composite primary key with UserID as the partition key and MessageID as the sort key.

You can distribute your upload work by using the sort key to load one item from each partition key value, then another item from each partition key value, and so on.

Every upload in this sequence uses a different partition key value, keeping more DynamoDB servers busy simultaneously and improving your throughput performance.

## Use Sort Keys for Version Control

Many applications need to maintain a history of item-level revisions for audit or compliance purposes and to be able to retrieve the most recent version easily.

For each new item, create two copies of the item. One copy should have a version-number prefix of zero (for example, v0) at the beginning of the sort key, and one should have a version-number prefix of one (for example, v001\_).

Every time the item is updated, use the next higher version prefix in the sort key of the updated version and copy the updated contents into the item with the version prefix of zero. This means that the latest version of any item can be located easily by using the zero prefix.

## Keep the Number of Indexes to a Minimum

Create secondary indexes on attributes that are queried often. Indexes that are seldom used contribute to increased storage and I/O costs without improving application performance.

## Choose Projections Carefully

Because secondary indexes consume storage and provisioned throughput, keep the size of the index as small as possible. Also, the smaller the index, the greater the performance advantage compared to querying the full table. Project only the attributes that you regularly request. Every time you update an attribute that is projected in an index, you incur the extra cost of updating the index as well.

## Optimize Frequent Queries to Avoid Fetches

To get the fastest queries with the lowest possible latency, project all of the attributes that you expect those queries to return. In particular, if you query a local secondary index for attributes that are not projected, Amazon DynamoDB automatically fetches those attributes from the table, which requires reading the entire item from the table. This introduces latency and additional I/O operations that you can avoid.

## Use Sparse Indexes

For any item in a table, Amazon DynamoDB writes a corresponding index entry only if the index sort key value is present in the item. If the sort key doesn't appear in every table item, the index is said to be sparse.

Sparse indexes are useful for queries over a small subsection of a table. It's faster and less expensive to query that index than to scan the entire table.

For example, suppose that you have a table in which you store all of your customer orders with the following key attributes:

Partition key: CustomerId

Sort key: OrderId

To track open orders, you can insert the OrderOpenDate attribute set to the date on which each order was placed and then delete it after the order is fulfilled. If you then create an index on CustomerId (partition key) and OrderOpenDate (sort key), only those orders with OrderOpenDate defined appear in it. That way, when you query the sparse index, the items returned are the orders that are unfulfilled and sorted by the date on which each order was placed.

## Avoid Scans as Much as Possible

In general, Scan operations are less efficient than other operations in DynamoDB. A *Scan operation* scans the entire table or secondary index. It then filters out values to provide the result you want.

If possible, avoid using a Scan operation on a large table or index with a filter that removes many results. Also, as a table or index grows, the Scan operation slows down. The Scan operation examines every item for the requested values and can use up the provisioned throughput for a large table or index in a single operation.

This usage of capacity units by a scan prevents other potentially more important requests for the same table from using the available capacity units. As a result, you'll likely get a ProvisionedThroughputExceeded exception for those requests.

For faster response times, design your tables and indexes so that your applications can use Query instead of Scan. (For tables, you can also consider using the GetItem and APIs.) GetItem is highly efficient because it provides direct access to the physical location of the item.

## Monitoring Costs

When you measure and monitor your users and applications and combine the data you collect with data from AWS monitoring tools, you can perform a gap analysis that tells you how closely aligned your system utilization is to your requirements. By working continually to minimize this utilization gap, you can ensure that your systems are cost effective.

Over time, you can continue to reduce cost with continuous monitoring and tagging. Similar to application development, cost optimization is an iterative process. Because your application and its usage will evolve over time and because AWS iterates frequently and regularly releases new options, it is important to evaluate your solution continuously.

## Cost Management Tools

AWS provides tools to help you identify those cost-saving opportunities and keep your resources right-sized. Use these tools to help you access, organize, understand, control, and optimize your costs.

### AWS Trusted Advisor

*AWS Trusted Advisor* is an online tool that provides you with real-time guidance to help you provision your resources following AWS best practices.

Whether you're establishing new workflows or developing applications, or as part of ongoing improvements, take advantage of the recommendations provided by Trusted Advisor on a regular basis. By reviewing the recommendations, you can look for opportunities to save money.

Here are some Trusted Advisor checks that help you determine how to reduce your bill:

- Low utilization of Amazon EC2 instances
- Idle resources, such as load balancers and Amazon RDS DB instances
- Underutilized Amazon EBS volumes and Amazon Redshift clusters
- Unassociated Elastic IP addresses
- Optimization, lease expiration—Amazon Reserved Instances
- Inefficiently configured Amazon Route 53 latency record sets

## AWS Cost Explorer

Use the *AWS Cost Explorer* tool to dive deeper into your cost and usage data to identify trends, pinpoint cost drivers, and detect anomalies. It includes Amazon EC2 usage reports, which let you analyze the cost and usage of your Amazon EC2 instances over the last 13 months. You can analyze your cost and usage data in aggregate (such as total costs and usage across all accounts) down to granular details (for example, `m2.2xlarge` costs within the Dev account tagged “*project: GuardDuty*”).

AWS Cost Explorer built-in reports include the following:

**Monthly Costs by AWS Service** Allows you to visualize the costs and usage associated with your top five cost-accruing AWS services and gives you a detailed breakdown on all services in the table view. The reports let you adjust the time range to view historical data going back up to 12 months to gain an understanding of your cost trends.

**Amazon EC2 Monthly Cost and Usage** Lets you view all AWS costs over the past two months, in addition to your current month-to-date costs. From there, you can drill down into the costs and usage associated with particular linked accounts, regions, tags, and more.

**Monthly Costs by Linked Account** Allows you to view the distribution of costs across your organization.

**Monthly Running Costs** Provides an overview of all running costs over the past three months and provides forecasted numbers for the coming month with a corresponding confidence interval. This report gives you good insight into how your costs are trending and helps you plan ahead.

AWS Cost Explorer Reserved Instance Reports include the following:

**RI Utilization Report** Visualize the degree to which you are using your existing resources and identify opportunities to improve your Reserved Instance cost efficiencies. The report shows how much you saved by using Reserved Instances, how much you overspent on Reserved Instances, and your net savings from purchasing Reserved Instances during the selected time range. This helps you to determine whether you have purchased too many Reserved Instances.

**RI Coverage Report** Discover how much of your overall instance usage is covered by Reserved Instances so that you can make informed decisions about when to purchase or modify a Reserved Instance to ensure maximum coverage. These show how much you spent on On-Demand Instances and how much you might have saved had you purchased more reservations. The report enables you to determine whether you have under-purchased Reserved Instances.

## AWS Cost Explorer API

Use *AWS Cost Explorer API* to query your cost and usage data programmatically (using AWS CLI or AWS SDKs). You can query for aggregated data such as total monthly costs or total daily usage. You can also query for granular data, such as the number of daily write operations for Amazon DynamoDB database tables in your production environment. All of the AWS

SDKs greatly simplify the process of signing requests and save you a significant amount of time when compared with using the AWS Cost Explorer API.

You can access your Amazon EC2 Reserved Instance purchase recommendations programmatically through the AWS Cost Explorer API. Recommendations for Reserved Instance purchases are calculated based on your past usage and indicate opportunities for potential cost savings.

The following example retrieves recommendations for Partial Upfront Amazon EC2 instances with a three-year term based on the last 60 days of Amazon EC2 usage.

Here's the AWS CLI command:

```
aws ce get-reservation-purchase-recommendation --service "Amazon Redshift"
--lookback-period-in-days SIXTY_DAYS --term-in-years THREE_YEARS --payment-
option PARTIAL_UPFRONT
```

Here's the output:

```
{
  "Recommendations": [],
  "Metadata": {
    "GenerationTimestamp": "2018-08-08T15:20:57Z",
    "RecommendationId": "00d59dde-a1ad-473f-8ff2-iexample3330b"
  }
}
```

## AWS Budgets

With *AWS Budgets*, you can set custom budgets that alert you when your costs or usage exceed (or are forecasted to exceed) your budgeted amount. You can also use AWS Budgets to set Reserved Instance utilization or coverage targets and receive alerts when your utilization drops below the threshold you define. Reserved Instance alerts support Amazon EC2, Amazon RDS, Amazon Redshift, and Amazon ElastiCache reservations.

Budgets can be tracked at the monthly, quarterly, or yearly level, and you can customize the start and end dates. You can further refine your budget to track costs associated with multiple dimensions, such as AWS service, linked account, tag, and others. You can send budget alerts through email or Amazon Simple Notification Service (Amazon SNS) topic. For example, you can set notifications that alert you if you accrue 80, 90, and 100 percent of your actual budgeted costs in addition to a notification that alerts you if you are forecasted to exceed your budget.

## AWS Cost and Usage Report

The *AWS Cost and Usage Report* tracks your AWS usage and provides estimated charges associated with that usage. You can configure this report to present the data hourly or daily. It is updated at least once a day until it is finalized at the end of the billing period. The AWS Cost and Usage report gives you the most granular insight possible into your costs and usage, and it is the source of truth for the billing pipeline. It can be used to develop advanced custom metrics using business intelligence, data analytics, and third-party cost optimization tools.

The AWS Cost and Usage report is delivered automatically to an S3 bucket that you specify, and it can be downloaded directly from there (standard Amazon S3 storage rates apply). It can also be ingested into Amazon Redshift or uploaded to Amazon QuickSight.

## Amazon CloudWatch

*Amazon CloudWatch* is a monitoring service for AWS Cloud resources and the applications you run on AWS. You can use Amazon CloudWatch to collect and track metrics and log files, set alarms, and automatically react to changes in your AWS resources. You can create an alarm to perform one or more of the following actions based on the value of the metric:

- Automatically stop or terminate Amazon EC2 instances that have gone unused or underutilized for too long
- Stop your instance if it has an EBS volume as its root device

For example, you may run development or test instances and occasionally forget to shut them off. You can create an alarm that is triggered when the average CPU utilization percentage has been lower than 10 percent for 24 hours, signaling that it is idle and no longer in use. You can create a group of alarms that first sends an email notification to developers whose instance has been underutilized for 8 hours and then terminates that instance if its utilization has not improved after 24 hours.

Amazon CloudWatch Events deliver a near real-time stream of system events that describe changes in AWS resources. Using simple rules, you can route each type of event to one or more targets, such as Lambda functions, Amazon Kinesis streams, and Amazon SNS topics.

## AWS Cost Optimization Monitor

*AWS Cost Optimization Monitor* is an automated reference deployment solution that processes detailed billing reports to provide granular metrics that you can search, analyze, and visualize in a customizable dashboard. The solution uploads detailed billing report data automatically to Amazon Elasticsearch Service (Amazon ES) for analysis and leverages its built-in support for Kibana, enabling you to visualize the first batch of data as soon as it's processed.

The default dashboard is configured to show specific cost and usage metrics. All of these metrics, as listed here, were selected based on best practices observed across AWS customers:

- Amazon EC2 Instances Running per Hour
- Total Cost
- Cost by Tag Key: Name
- Cost by Amazon EC2 Instance Type
- Amazon EC2 Elasticity
- Amazon EC2 Hours per Dollar Invested

## Cost Optimization: Amazon EC2 Right Sizing

*Amazon EC2 Right Sizing* is an automated AWS reference deployment solution that uses managed services to perform a right-sizing analysis and offer detailed recommendations for more cost-effective instances. The solution analyzes two weeks of utilization data to provide detailed recommendations for right sizing your Amazon EC2 instances.

# Monitoring Performance

After you have implemented your architecture, monitor its performance so that you can remediate any issues before your customers are aware of them. Use monitoring metrics to raise alarms when thresholds are breached. The alarm can trigger automated action to work around any components with poor performance.

AWS provides tools that you can use to monitor the performance, reliability, and availability of your resources on the AWS Cloud.

## Amazon CloudWatch

*Amazon CloudWatch* is essential to performance efficiency, which provides system-wide visibility into resource utilization, application performance, and operational health.

You can create an alarm to monitor any Amazon CloudWatch metric in your account. For example, you can create alarms on an Amazon EC2 instance CPU utilization, Elastic Load Balancing request latency, Amazon DynamoDB table throughput, or Amazon SQS queue length.

In the following example, AWS CLI is used to create an alarm to send an Amazon SNS email message when CPU utilization exceeds 70 percent:

```
aws cloudwatch put-metric-alarm --alarm-name cpu-mon --alarm-description "Alarm when CPU exceeds 70 percent" --metric-name CPUUtilization --namespace AWS/EC2 --statistic Average --period 300 --threshold 70 --comparison-operator GreaterThanThreshold --dimensions "Name=InstanceId,Value=i-12345678" --evaluation-periods 2 --alarm-actions arn:aws:sns:us-east-1:111122223333:MyTopic --unit Percent
```

Here are a few examples of when and how alarms are sent:

- Sends an email message using Amazon SNS when the average CPU use of an Amazon EC2 instance exceeds a specified threshold for consecutive specified periods
- Sends an email when an instance exceeds 10 GB of outbound network traffic per day
- Stops an instance and sends a text message (SMS) when outbound traffic exceeds 1 GB per hour
- Stops an instance when memory utilization reaches or exceeds 90 percent so that application logs can be retrieved for troubleshooting

## AWS Trusted Advisor

*AWS Trusted Advisor* inspects your AWS environment and makes recommendations that help to improve the speed and responsiveness of your applications.

The following are a few Trusted Advisor checks to improve the performance of your service. The Trusted Advisor checks your service limits, ensuring that you take advantage of provisioned throughput, and monitors for overutilized instances:

- Amazon EC2 instances that are consistently at high utilization can indicate optimized, steady performance, but this check can also indicate that an application does not have enough resources.
- Provisioned IOPS (SSD) volumes that are attached to an Amazon EC2 instance that is not Amazon EBS-optimized. Amazon EBS volumes are designed to deliver the expected performance only when they are attached to an EBS-optimized instance.
- Amazon EC2 security groups with a large number of rules.
- Amazon EC2 instances that have a large number of security group rules.
- Amazon EBS magnetic volumes (standard) that are potentially overutilized and might benefit from a more efficient configuration.
- CloudFront distributions for alternate domain names with incorrectly configured DNS settings.

Some HTTP request headers, such as Date or User-Agent, significantly reduce the cache hit ratio. This increases the load on your origin and reduces performance because CloudFront must forward more requests to your origin.

## Summary

In this chapter, you learned about the following:

- Cost-optimizing practices
- Right sizing your infrastructure
- Optimizing using Reserved Instances, Spot Instances, and AWS Auto Scaling
- Optimizing storage and data transfer
- Optimizing using NoSQL database (Amazon DynamoDB)
- Monitoring your costs and performance
- Tools, such as AWS Trusted Advisor, Amazon CloudWatch, and AWS Budgets

Achieving an optimized system is a continual process. An optimized system uses all the provisioned resources efficiently and achieves your business goal at the lowest price point. Engineers must know the cost of deploying resources and how to architect for cost optimization. Practice eliminating the waste and bring accountability in every step of the build process. Use mandatory cost tags on all of your resources to gain precise insights into

usage. Define IAM policies to enforce tag usage, and use tagging tools, such as AWS Config and AWS Tag Editor, to manage tags. Be cost-conscious, reduce the usage by terminating unused instances, and delete old snapshots and unused keys.

Right size your infrastructure by matching instance types and sizes, and set periodic checks to ensure that the initial provision remains optimum as your business changes over time. With Amazon EC2, you can choose the combination of instance types and sizes most appropriate for your applications. Amazon RDS instances are also optimized for memory, performance, and I/O.

Amazon EC2 Reserved Instances provide you with a significant discount (up to 75 percent) compared to On-Demand Instance pricing. Using Convertible Reserved Instances, you can change instance families, OS types, and tenancies while benefitting from Reserved Instance pricing. Reserved Instance Marketplace allows you to sell the unused Reserved Instances or buy them from other AWS customers, usually at lower prices and shorter terms. With size flexibility, discounted rates for Amazon RDS Reserved Instances are automatically applied to the usage of any size within the instance family.

Spot Instances provide an additional option for obtaining compute capacity at a reduced cost and can be used along with On-Demand and Reserved Instances. Spot Fleets enable you to launch and maintain the target capacity and to request resources automatically to replace any that are disrupted or manually terminated. Using the termination notices and persistent requests in your application design help to maintain continuity as the result of interruptions.

AWS Auto Scaling automatically scales if your application experiences variable load and uses one or more scalable resources, such as Amazon ECS, Amazon DynamoDB, Amazon Aurora, Amazon EC2 Spot requests, and Amazon EC2 scaling groups. Predictive Scaling uses machine learning models to forecast daily and weekly patterns. Amazon EC2 Auto Scaling enables you to scale in response to demand and known load schedules. It supports the provisioning of scale instances across purchase options, Availability Zones, and instance families to optimize performance and cost.

Containers provide process isolation and improve the resource utilization. Amazon ECS lets you easily build all types of containerized applications and launch thousands of containers in seconds with no additional complexity. With AWS Fargate technology, you can manage containers without having to provision or manage servers. It enables you to focus on building and running applications, not the underlying infrastructure.

AWS Lambda takes care of receiving events or client invocations and then instantiates and runs the code. That means there's no need to manage servers. Serverless services have built-in automatic scaling, availability, and fault tolerance. These features allow you to focus on product innovation and rapidly construct applications, such as web applications, websites, web-hooked systems, chatbots, and clickstream.

AWS storage services are optimized to meet different storage requirements. Use the Amazon S3 analytics feature to analyze storage access patterns to help you decide when to transition the right data to the right storage class and to yield considerable savings. Monitor Amazon EBS volumes periodically to identify ones that are unattached or appear to be underutilized or overutilized, and adjust provisioning to match actual usage.

Optimizing data transfer ensures that you minimize data transfer costs. Use options such as Amazon CloudFront, Amazon S3 transfer acceleration, and Amazon Route 53 to let data reach Regions faster and reduce latency issues.

NoSQL database systems like Amazon DynamoDB use alternative models for data management, such as key-value pairs or document storage. DynamoDB enables you to offload the administrative burdens of operating and scaling a distributed database so that you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling. Follow best practices, such as distributing data evenly, effective partition and sort keys usage, efficient data scanning, and using sparse indexes for maximizing performance and minimizing throughput costs, when working with Amazon DynamoDB.

AWS provides several tools to help you identify those cost-saving opportunities and keep your resources right-sized. AWS Trusted Advisor inspects your AWS environment to identify idle and underutilized resources and provides real-time insight into service usage to help you improve system performance and save money. Amazon CloudWatch collects and tracks metrics, monitors log files, sets alarms, and reacts to changes in AWS resources automatically. AWS Cost Explorer checks patterns in AWS spend over time, projects future costs, identifies areas that need further inquiry, and provides Reserved Instance recommendations.

Optimization is an ongoing process. Always stay current with the pace of AWS new releases, and assess your existing design solutions to ensure that they remain cost-effective.

## Exam Essentials

**Know the importance of tagging.** By using tags, you can assign metadata to AWS resources. This tagging makes it easier to manage, search for, and filter resources in billing reports and automation activities and when setting up access controls.

**Know about various tagging tools and how to enforce the tag rules.** With AWS Tag Editor, you can add tags to multiple resources at once, search for the resources that you want to tag, and then add, remove, or edit tags for the resources in your search results. AWS Config identifies resources that do not comply with tagging policies. You can use IAM policy conditions to force the usage of tags while creating the resources.

**Know the fundamental practices in reducing the usage.** Follow the best practices of cost optimization in every step of your build process, such as turning off unused resources, spinning up instances only when needed, and spinning them down when not in use. Use tagging to help with the cost allocation. Use Amazon EC2 Spot Instances, Amazon EC2, and Reserved Instances where appropriate, and use alerts, notifications, and cost-management tools to stay on track.

**Know the various usage patterns for right sizing.** By understanding your business use case and backing up the analysis with performance metrics, you can choose the most

appropriate options, such as steady state; variable; predictable, but temporary; and development, test, and production usage.

**Know the various instance families for right sizing and the corresponding use cases.**

Amazon EC2 provides a wide selection of instances to match capacity needs at the lowest cost and comes with different options for CPU, memory, and network resources. The families include General Purpose, Compute Optimized, Memory Optimized, Storage Optimized, and Accelerated Computing.

**Know Amazon EC2 Auto Scaling benefits and how this feature can make your solutions more optimized and highly available.** AWS Auto Scaling is a fast, easy way to optimize the performance and costs of your applications. It makes smart scaling decisions based on your preferences, automatically maintains performance even when your workloads are periodic, unpredictable, and continuously changing.

**Know how to create a single AWS Auto Scaling group to scale instances across different purchase options.** You can provision and automatically scale Amazon EC2 capacity across different Amazon EC2 instance types, Availability Zones, and On-Demand, Reserved Instances, and Spot purchase options in a single AWS Auto Scaling group. You can define the desired split between On-Demand and Spot capacity, select which instance types work for your application, and specify preferences for how Amazon EC2 Auto Scaling should distribute the AWS Auto Scaling group capacity within each purchasing model.

**Know how block, object, and file storages are different.** Block storage is commonly dedicated, low-latency storage for each host, and it is provisioned with each instance. Object storage is developed for the cloud, has vast scalability, is accessed over the web, and is not directly attached to an instance. File storage enables accessing shared files as a file system.

**Know key CloudWatch metrics available to measure the Amazon EBS efficiency and how to use them.** CloudWatch metrics are statistical data that you can use to view, analyze, and set alarms on the operational behavior of your volumes. Depending on your needs, set alarms and response actions that correspond to each data point. For example, if your I/O latency is higher than you require, check the metric `VolumeQueueLength` to make sure that your application is not trying to drive more IOPS than you have provisioned. Review and learn more about the available metrics that help optimize the block storage.

**Know tools and features that help in efficient data transfer.** Using Amazon CloudFront, you can locate data closer to users and reduce administrative efforts to minimize data transfer costs. Amazon S3 Transfer Acceleration enables fast data transfer over an optimized network path. Use the multipart upload file option while uploading a large file to improve network throughput.

**Know key differences between RDBMS and NoSQL databases to design efficient solutions using Amazon DynamoDB.** Schema flexibility and the ability to store related items together make DynamoDB a solution for solving problems associated with changing business needs and scalability issues unlike relational databases.

**Know the importance of distributing the data evenly when designing DynamoDB tables.** Use provisioned throughput more efficiently by making the partition key more distinct. That way, data spreads throughout the provisioned space. Use the sort key with the

partition key to make a unique key to achieve better performance while uploading data simultaneously.

**Know the different ways to read data from DynamoDB tables to avoid scans.** DynamoDB provides Query and Scan actions to read data from a table and does not support table joins. DynamoDB provides the GetItem action for retrieving an item by its primary key. GetItem is highly efficient because it provides direct access to the physical location of the item. The scan always scans the entire table and can consume large amounts of system resources.

**Know the AWS Cost Management tools and their features.** AWS provides tools to help you manage, monitor, and, ultimately, optimize your costs. Use AWS Cost Explorer for deeper dives into the cost drivers. Use AWS Trusted Advisor to inspect your AWS infrastructure to identify overutilized or idle resources. AWS Budgets enables you to set custom cost and usage budgets and receive alerts when budgets approach or exceed the limits. There are a wide range of tools to explore, such as AWS Cost Optimization – Amazon EC2 Right Sizing, and monitoring tools to identify additional savings opportunities.

**Know how the AWS Trusted Advisor features help in saving costs and improving the performance of your solutions.** AWS Trusted Advisor scans your AWS environment, compares it to AWS best practices, and makes recommendations for saving money, improving system performance, and more. Cost Optimization recommendations highlight unused and underutilized resources. Performance recommendations help to improve the speed and responsiveness of your applications.

**Know how to evaluate the reporting details in the AWS Cost Explorer default reports.** Cost Explorer provides you with default reports: Cost and Usage reports and Reserved Instance reports. Cost and Usage reports include your daily costs and monthly costs by service, listing the top five services. These reports help you to determine whether you have purchased too many Reserved Instances. The Reserved Instance Coverage reports show how many of your instance hours are covered by Reserved Instances, how much you spent on On-Demand Instances, and how much you might have saved had you purchased more reservations. This enables you to determine whether you have under-purchased Reserved Instances.

**Know how to extract recommendations using AWS Cost Explorer API.** The Cost Explorer API allows you to use either AWS CLI or SDKs to query your cost and usage data. You can query for aggregated data, such as total monthly costs or total daily usage. You can also query for granular data, such as the number of daily write operations for DynamoDB database tables in your production environment.

**Know all of the Amazon CloudWatch metrics and how to set alarms.** With Amazon CloudWatch, you can observe CPU utilization, network throughput, and disk I/O, and match the observed peak metrics to a new and cheaper instance type. You choose a CloudWatch metric and threshold for the alarm to watch. The alarm turns into the alarm state when the metric breaches the threshold for a specified number of evaluation periods. Use the Amazon CloudWatch console, AWS CLI, or AWS SDKs for creating or managing alarms.

**Know how AWS Lambda integrates with other AWS serverless services to build cost-effective solutions.** AWS Lambda provides the cloud-logic layer and integrates seamlessly with the other serverless services to build virtually any type of application or backend service. For

example, Amazon S3 automatically triggers Lambda functions when an object is created, copied, or deleted. Lambda functions can process Amazon SQS messages.

## Resources to Review

AWS Well-Architected Framework (Whitepaper):

[https://d1.awsstatic.com/whitepapers/architecture/AWS\\_Well-Architected\\_Framework.pdf](https://d1.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf)

Cost Optimization Pillar—AWS Well-Architected Framework (Whitepaper):

<https://d1.awsstatic.com/whitepapers/architecture/AWS-Cost-Optimization-Pillar.pdf>

Performance Efficiency Pillar—AWS Well-Architected Framework (Whitepaper):

<https://d0.awsstatic.com/whitepapers/architecture/AWS-Performance-Efficiency-Pillar.pdf>

Cost Management in the AWS Cloud (Whitepaper):

<https://d1.awsstatic.com/whitepapers/aws-tco-2-cost-management.pdf>

Architecting for the Cloud—AWS Best Practices (Whitepaper):

[https://d1.awsstatic.com/whitepapers/AWS\\_Cloud\\_Best\\_Practices.pdf](https://d1.awsstatic.com/whitepapers/AWS_Cloud_Best_Practices.pdf)

Creating a Culture of Cost Transparency and Accountability (Whitepaper):

<https://d1.awsstatic.com/whitepapers/cost-optimization-transparency-accountability.pdf>

Maximizing Value with AWS (Whitepaper):

<https://d1.awsstatic.com/whitepapers/total-cost-of-operation-benefits-using-aws.pdf>

Laying the Foundation: Setting Up Your Environment for Cost Optimization (Whitepaper):

<https://docs.aws.amazon.com/whitepapers/latest/cost-optimization-laying-the-foundation/introduction.html>

Right Sizing: Provisioning Instances to Match Workloads (Whitepaper):

<https://d1.awsstatic.com/whitepapers/cost-optimization-right-sizing.pdf>

AWS Storage Optimization (Whitepaper):

<https://docs.aws.amazon.com/whitepapers/latest/cost-optimization-storage-optimization/introduction.html>

AWS Storage Services Overview (Whitepaper):

[https://d1.awsstatic.com/whitepapers/AWS\\_Storage\\_Services\\_Whitepaper-v9.pdf](https://d1.awsstatic.com/whitepapers/AWS_Storage_Services_Whitepaper-v9.pdf)

Optimizing Enterprise Economics with Serverless Architectures (Whitepaper):

<https://d0.awsstatic.com/whitepapers/optimizing-enterprise-economics-serverless-architectures.pdf>

Serverless Architectures with AWS Lambda (Whitepaper):

<https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf>

Cloud Storage with AWS:

<https://aws.amazon.com/products/storage/>

Tagging Your Amazon EC2 Resources:

[https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Using\\_Tags.html](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Using_Tags.html)

Amazon EC2 Instance Types:

<https://aws.amazon.com/ec2/instance-types/>

Amazon EC2 Reserved Instances:

<https://aws.amazon.com/ec2/pricing/reserved-instances/>

Amazon RDS Reserved Instances:

<https://aws.amazon.com/rds/reserved-instances/>

Amazon EC2 Spot Instances:

<https://aws.amazon.com/ec2/spot/>

AWS Auto Scaling:

<https://aws.amazon.com/blogs/aws/category/auto-scaling/>

Containers on AWS:

<https://aws.amazon.com/containers/services/>

Automatic Scaling for Spot Fleet:

<https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-fleet-automatic-scaling.html>

Using Amazon Aurora Auto Scaling with Aurora Replicas:

<https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Integrating.AutoScale.html>

Amazon Elastic Container Service:

[https://docs.aws.amazon.com/ecs/index.html#lang/en\\_us](https://docs.aws.amazon.com/ecs/index.html#lang/en_us).

Best Practices for DynamoDB:

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-practices.html>

AWS Trusted Advisor:

[https://aws.amazon.com/premiumsupport/technology/trusted-advisor/.](https://aws.amazon.com/premiumsupport/technology/trusted-advisor/)

Analyzing Your Costs with Cost Explorer:

<https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/ce-what-is.html>

Using Amazon CloudWatch Alarms:

<https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/AlarmThatSendsEmail.html>

AWS re:Invent 2014 | (ENT302) Cost Optimization on AWS (Video):

<https://www.youtube.com/watch?v=mqY8xfKU5yE>

## Exercises



Before you begin this task, you must first create an SNS topic (name: myHighCpuAlarm) and subscribe to it.

### EXERCISE 16.1

#### Set Up a CPU Usage Alarm Using AWS CLI

In this exercise, you will use the AWS CLI to create a CPU usage alarm that sends an email message using Amazon SNS when the CPU usage exceeds 70 percent.

1. Set up an SNS topic with the name **myHighCpuAlarm** and subscribe to it. For more information, see this article:

[https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/US\\_SetupSNS.html](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/US_SetupSNS.html)

2. Create an alarm using the `put-metric-alarm` command as follows:

```
aws cloudwatch put-metric-alarm \
  --alarm-name cpu-mon \
  --alarm-description "Alarm when CPU exceeds 70%" \
  --metric-name CPUUtilization \
  --namespace AWS/EC2 \
  --statistic Average \
  --period 300 \
  --threshold 70 \
  --comparison-operator GreaterThanThreshold \
  --dimensions Name=InstanceId,Value=i-12345678 \
  --evaluation-periods 2 \
  --alarm-actions arn:aws:sns:us-east-1:111122223333:myHighCpuAlarm \
  --unit Percent
```



For Windows, replace the backslash (\) Unix continuation character at the end of each line with a caret (^).

3. Test the alarm by forcing an alarm state change using the set-alarm-state command.
  - a. Change the alarm state from INSUFFICIENT\_DATA to OK.

```
aws cloudwatch set-alarm-state --alarm-name cpu-mon --state-reason "initializing" --state-value OK
```
  - b. Change the alarm state from OK to ALARM.

```
aws cloudwatch set-alarm-state -alarm-name cpu-mon --state-reason "initializing" --state-value ALARM
```
  - c. Check that you have received an email notification about the alarm.

Using AWS CLI, you created a CPU alarm that sends an email notification when CPU usage exceeds 70 percent. You tested it by manually changing its alarm state to ALARM.

## EXERCISE 16.2

### Modify Amazon EBS Optimization for a Running Instance

In this exercise, you will use the Amazon EC2 console to enable the optimization for a running instance by modifying its Amazon EBS optimized instance attribute.

1. Open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
2. In the navigation pane, click **Instances**, and select the instance.
3. Choose **Actions > Instance State > Stop**.
4. In the **Confirmation** dialog box, choose **Yes > Stop**.  
It can take a few minutes for the instance to stop.
5. With the instance still selected, choose **Actions > Instance Settings** and then choose **Change Instance Type**.
6. In the **Change Instance Type** dialog box, do one of the following:
  - a. If the instance type of your instance is Amazon EBS-optimized, **EBS-optimized** is selected by default, and you cannot change it. Choose **Cancel**.
  - b. If the instance type of your instance supports Amazon EBS optimization, choose **EBS-optimized > Apply**.

(continued)

**EXERCISE 16.2 (continued)**

- c. If the instance type of your instance does not support Amazon EBS optimization, select an instance type from **Instance Type** that supports Amazon EBS optimization and then choose **EBS-optimized > Apply**.
7. Choose **Actions > Instance State > Start**.

You enabled the EBS optimization feature for a running Amazon EC2 instance using AWS Console.

---



When you stop an instance, the data on any instance store volumes is erased. To keep data in instance store volumes, back it up to persistent storage.

**EXERCISE 16.3****Create an AWS Config Rule**

In this exercise, using the AWS Management Console, you will create an AWS Config rule to monitor whether Elastic IP addresses are attached to Amazon EC2 instances.

1. Create an Elastic IP address to be used as part of this exercise, but do not attach it to any Amazon EC2 instance. See the following for instructions:  
<https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html#using-instance-addressing-eips-releasing>
  2. Open the AWS Config console at <https://console.aws.amazon.com/config/>.
  3. Choose **Get Started Now**.
  4. On the **Settings** page, for **Resource types to record**, select **All resources**.
  5. For **Amazon S3 Bucket**, select the Amazon S3 bucket to which AWS Config sends configuration history and configuration snapshot files.
  6. For **Amazon SNS Topic**, select whether AWS Config streams information by selecting the **Stream configuration changes and notifications to an Amazon SNS topic**.
  7. For **Topic Name**, type a name for your SNS topic.
  8. For **Bucket Name**, type a name for your Amazon S3 bucket.
  9. For **AWS Config role**, choose the IAM role that grants AWS Config permission to record configuration information and send this information to Amazon S3 and Amazon SNS.
  10. Choose **Create AWS Config service-linked role**, and then **Next**.
-

- 
11. On the **AWS Config Role** page, in the search bar, enter **eip** to find a specific rule from the list.

12. Select the **eip-attached** rule.

13. Choose **Next** and then **Confirm**.

AWS Config will run this rule against your resources. The rule flags the unattached EIP as non-compliant.

14. Delete the AWS Config rule.

15. Release the Elastic IP address. See the following for instructions:

<https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html#using-instance-addressing-eips-releasing>

From the AWS Config console, you used AWS Config to create a rule to determine whether an Elastic IP address is attached to an Amazon EC2 instance.

---

#### EXERCISE 16.4

### Create a Launch Configuration and an AWS Auto Scaling Group, and Schedule a Scaling Action

In this exercise, using AWS Management Console, you will create a launch configuration and AWS Auto Scaling policy, and verify the scheduled scaling action.

1. To create a launch configuration, complete the following steps:
  - a. Open the Amazon EC2 console at <https://console.aws.amazon.com/ec2/>.
  - b. On the navigation pane, under **AWS Auto Scaling**, choose **Launch Configurations**. On the next page, choose **Create launch configuration**.
  - c. On the **Choose AMI** page, select your custom AMI.
  - d. On the **Choose Instance Type** page, select a hardware configuration for your instance and choose **Next: Configure details**. Configure the remaining details.
  - e. On the **Configure Details** page, do the following:
    - (i) For **Name**, type a name for your launch configuration.
    - (ii) For **Advanced Details, IP Address Type**, select **Assign a public IP address to every instance**.
  - f. Choose **Skip to review**.
  - g. On the **Review** page, choose **Edit security groups**. Follow the instructions to choose an existing security group, and then choose **Review**.
  - h. On the **Review** page, choose **Create launch configuration**.

---

*(continued)*

**EXERCISE 16.4 (*continued*)**

- i. For **Select an existing key pair or create a new key pair** page, select one of the listed options.
- j. Select the acknowledgment check box, and then choose **Create launch configuration**.
2. To create an AWS Auto Scaling group, complete the following steps:
  - a. Select **Create an AWS Auto Scaling group using this launch configuration**.
  - b. On the **Create AWS Auto Scaling Group** page, follow these steps:
    - (i) For **Group name**, enter a name for your AWS Auto Scaling group.
    - (ii) For **Group size**, enter **1** as the initial number of instances for your AWS Auto Scaling group.
    - (iii) For **Network**, select the default VPC.
    - (iv) For **Subnet**, select one or more subnets from the listed subnets.
  - c. Choose **Next: Configure scaling policies**.
  - d. On the **Configure scaling policies** page, select **Keep this group at its initial size** and then choose **Review**.
  - e. On the **Review** page, choose **Create AWS Auto Scaling group**.
  - f. On the **AWS Auto Scaling group creation status** page, choose **Close**.
3. To schedule an AWS Auto Scaling action and verify that it's working, complete the following steps:
  - a. Select your AWS Auto Scaling group.
  - b. On the **Schedule Actions** tab, select **Create Scheduled Action**.
  - c. On **Schedule Action** page, follow these steps:
    - (i) For **Name**, type name of the action.
    - (ii) For **Max**, type **2**.
    - (iii) For **Desired Capacity**, type **2**.
    - (iv) For **Start Time**, select current day in Date (UTC), and type current UTC time + 2 minutes.
  - d. Select **Save**.
  - e. Select the **Instances** tab, refresh the tab in the next two minutes, and observe that a new Amazon EC2 instance was created.

In this exercise, you created a launch configuration and an AWS Auto Scaling group using the launch group that you just created. To test whether automatic scaling is working, you added a Scaling action to launch a new Amazon EC2 instance by increasing capacity. You also verified that a new instance was added to the current capacity.

# Review Questions

1. You are developing an application that will run across dozens of instances. It uses some components from a legacy application that requires some configuration files to be copied from a central location and held on a volume local to each of the instances. You plan to modify your application with a new component in the future that will hold this configuration in Amazon DynamoDB. Which storage option should you use in the interim to provide the lowest cost and the lowest latency for your application to access the configuration files?
  - A. Amazon S3
  - B. Amazon EBS
  - C. Amazon EFS
  - D. Amazon EC2 instance store
2. Similar to SQL, Amazon DynamoDB provides several operations for reading the data. Which operation is the most efficient way to retrieve a single item?
  - A. Query
  - B. Scan
  - C. GetItem
  - D. Join
3. AWS Trusted Advisor offers a rich set of best practice checks and recommendations across five categories: cost optimization, security, fault tolerance, performance, and service limits. Which of the following checks is NOT under Cost and Performance categories?
  - A. Amazon EBS Provisioned IOPS (SSD) volume attachment configuration
  - B. Amazon CloudFront header forwarding and cache hit ratio
  - C. Amazon EC2 Availability Zone balance
  - D. Unassociated Elastic IP address
4. Which of the following common partition schemas includes a partition key design that distributes I/O requests evenly across partitions and uses provisioned I/O capacity of an Amazon DynamoDB table efficiently?
  - A. Status code, where there are only a few possible status codes
  - B. User ID, where the application has many users
  - C. Item creation date, rounded to the nearest time period
  - D. Device ID, where even if there are many devices tracked, one is by far more popular than all the others

5. You are developing an application that consists of a set of Amazon EC2 instances hosting a web layer and a database hosting a MySQL instance. You are required to add a layer that can be used to ensure that the most frequently accessed data from the database is fetched in a faster and more efficient manner. Which of the following can be used to store the frequently accessed data?
  - A. Amazon Simple Queue Service (Amazon SQS) queue
  - B. Amazon Simple Notification Service (Amazon SNS) topic
  - C. Amazon CloudFront distribution
  - D. Amazon ElastiCache instance
6. You have an application deployed to the AWS platform. The application makes requests to an Amazon Simple Storage Service (Amazon S3) bucket. After monitoring the Amazon CloudWatch metrics, you notice that the number of GET requests has suddenly spiked. Which of the following can be used to optimize Amazon S3 cost and performance?
  - A. Add Amazon ElastiCache in front of the S3 bucket.
  - B. Use Amazon DynamoDB instead of Amazon S3.
  - C. Place an Amazon CloudFront distribution in front of the S3 bucket.
  - D. Place an Elastic Load Balancing load balancer in front of the S3 bucket.
7. You are writing an application that will store data in an Amazon DynamoDB table. The ratio of read operations to write operations will be 1,000 to 1, with the same data being accessed frequently. Which feature or service should you enable on the DynamoDB table to optimize performance and minimize costs?
  - A. Amazon DynamoDB Auto Scaling
  - B. Amazon DynamoDB cross-region replication
  - C. Amazon DynamoDB Streams
  - D. Amazon DynamoDB Accelerator
8. A developer is migrating an on-premises web application to the AWS Cloud. The application currently runs on a 32-processor server and stores session state in memory. On Mondays, the server runs at 80 percent CPU utilization, but at only about 5 percent CPU utilization at other times. How should the developer change the code to optimize running in the AWS Cloud?
  - A. Store session state on the Amazon EC2 instance store.
  - B. Encrypt the session state in memory.
  - C. Store session state in an Amazon ElastiCache cluster.
  - D. Compress the session state in memory.

9. A company is using an ElastiCache cluster in front of their Amazon RDS instance. The company would like you to implement logic into the code so that the cluster retrieves data from Amazon RDS only when there is a cache miss. Which strategy can you implement to achieve this?
  - A. Error retries
  - B. Lazy loading
  - C. Exponential backoff
  - D. Write-through
10. Your application will be hosted on an Amazon EC2 instance, which will be part of an AWS Auto Scaling group. The application must fetch the private IP of the instance. Which of the following can achieve this?
  - A. Query the instance metadata.
  - B. Query the instance user data.
  - C. Have the application run ifconfig.
  - D. Have an administrator get the IP address from the Amazon EC2 console.
11. You just developed code in AWS Lambda that uses recursive functions. You see some throttling errors in the metrics. Which of the following should you do to resolve the issue?
  - A. Use API Gateway to call the recursive code.
  - B. Use versioning for the recursive function.
  - C. Place the recursive function in a separate package.
  - D. Avoid using recursive code in your function.
12. A production application is making calls to an Amazon Relational Database Service (Amazon RDS) instance. The application's reporting module is experiencing heavy traffic, causing performance issues. How can the application be optimized to alleviate this issue?
  - A. Move the database to Amazon DynamoDB, and point the reporting module to the new DynamoDB table.
  - B. Enable Multi-AZ for the database, and point the reporting module to the secondary database.
  - C. Enable read replicas for the database, and point the reporting module to the read replica.
  - D. Place an Elastic Load Balancing load balancer in front of the reporting part of the application.
13. Your application uses Amazon S3 buckets. You have users in other countries accessing objects in those buckets. What can you do to reduce latency for those users outside of your country?
  - A. Host a static website.
  - B. Change the storage class.
  - C. Enable cross-region replication.
  - D. Enable encryption.

- 14.** You have an application that uploads objects to Amazon S3 between 200–500 MB. The process takes longer than expected, and you want to improve the performance of the application. Which of the following would you consider?
- A.** Enable versioning on the bucket.
  - B.** Use the multipart upload API.
  - C.** Write the items in batches for better performance.
  - D.** Create multiple threads to upload the objects.
- 15.** You must bootstrap your application script to instances that are launched inside an AWS Auto Scaling group. Which is the most optimal way to achieve this?
- A.** Create a Lambda function to install the script.
  - B.** Place a scheduled task on the instance that starts on boot.
  - C.** Place the script in the instance user data.
  - D.** Place the script in the instance metadata.

# Appendix



# Answers to Review Questions

# Chapter 1: Introduction to AWS Cloud API

1. B. The specific credentials include the access key ID and secret access key. If the access key is valid only for a short-term session, the credentials also include a session token.  
AWS uses the user name and passwords for working with the AWS Management Console, not for working with the APIs. Data encryption uses the customer master keys, not API access.
2. C. Most AWS API services are regional in scope. The service is running and replicating your data across multiple Availability Zones within an AWS Region. You choose a regional API endpoint either from your default configuration or by explicitly setting a location for your API client.
3. A. The AWS SDK relies on access keys, not passwords. The best practice is to use AWS Identity and Access Management (IAM) credentials and not the AWS account credentials. Comparing IAM users or IAM roles, only IAM users can have long-term security credentials.
4. C. Although you can generate IAM users for everyone, this introduces management overhead of a new set of long-term credentials. If you already have an external directory of your organization's users, use IAM roles and identity federation to provide short-term, session-based access to AWS.
5. A. The permissions for **DynamoDBFullAccess** managed policy grant access to *all* Amazon DynamoDB tables in your account. Write a custom policy to scope the access to a specific table. You can update the permissions of a user independently from the lifecycle of the table. DynamoDB does not have its own concept of users, but it uses the AWS API and relies on IAM.
6. B. You can view or manage your AWS resources with the console, AWS CLI, or AWS SDK. The core functionality of each SDK is powered by a common set of web services on the backend. Most AWS services are isolated by AWS Region.
7. B. If you look closely at the URL, the AWS Region string is incorrectly set as **us-east-1a**, which is specific to the Availability Zone. An AWS Region string ends in a number, and the correct configuration is **us-east-1**. If the error was related to API credentials, you would receive a more specific error related to credentials, such as **AccessDenied**.
8. B. This policy allows access to the **s3>ListBucket** operation on **example\_bucket** as a specific bucket. This does not grant access to operations on the objects within the bucket. IAM is granular. The date in the **Version** attribute is a specific version of the IAM policy language and not an expiration.
9. D. The long-term credentials are not limited to a single AWS Region. IAM is a global service, and IAM user credentials are valid across different AWS Regions. However, when the API call is made, a signing key is derived from the long-term credentials, and that signing key is scoped to a region, service, and day.

10. B. The AssumeRole method of the AWS Security Token Service (AWS STS) returns the security credentials for the role that include the access key ID, secret access key, and session token. AWS Key Management Service (AWS KMS) is not used for API signing. The identity provider may provide a SAML assertion, but AWS STS generates the AWS API credentials.
11. D. The DynamoDBReadOnlyAccess policy is a built-in policy that applies to the resource \* wildcard, which means that it applies to any and all DynamoDB tables accessible from the account regardless of when those tables were created. Because IAM policies are related to the IAM user, not the access key, rotating the key does not affect the policy. IAM policies are also global in scope, so you do not need a custom one per AWS Region. You can add IAM users to IAM groups but not IAM roles. Instead, roles must be assumed for short-term sessions.
12. B. The IAM trust policy defines the principals who can request role credentials from the AWS STS. Access policies define what API actions can be performed with the credentials from the role.
13. C. You can define an IAM user for your new team member and add the IAM user to an IAM group to inherit the appropriate permissions. The best practice is *not* to use AWS *account root user* credentials. Though you can use AWS Directory Service to track users, this answer is incomplete, and the AWS KMS is not related to permissions. Roles can be assumed only for short-term sessions—there are no long-term credentials directly associated with the role.
14. C. The AWS API backend is accessed through web service calls and is operating system- and programming language-agnostic. You do not need to do anything special to enable specific programming languages other than downloading the appropriate SDK.
15. B. The primary latency concern is for customers accessing the data, and there are no explicit dependencies on existing infrastructure in the United States. Physically locating the application resources closer to these users in Australia reduces the distance that the information must travel and therefore decreases the latency.

## Chapter 2: Introduction to Compute and Networking

1. B. You launch Amazon Elastic Compute Cloud (Amazon EC2) instances into specific subnets that are tied to specific Availability Zones. You can look up the Availability Zone in which you have launched an Amazon EC2 instance. While an Availability Zone is part of a region, this answer is not the most specific. You do not get to choose the specific data center, and edge locations do not support EC2.
2. B. When you stop an Amazon EC2 instance, its public IP address is released. When you start it again, a new public IP address is assigned. If you require a public IP address to be persistently associated with the instance, allocate an Elastic IP address. SSH key pairs and security group rules do not have any built-in expiration, and SSH is enabled as a service by default. It is available even after restarts. Security groups do not expire.

3. A. A restricted rule that allows RDP from only certain IP addresses may block your request if you have a new IP address because of your location. Because you are trying to connect to the instance, verify that an appropriate inbound rule is set as opposed to an outbound rule. For many variants of Windows, RDP is the default connection mechanism, and it defaults to enabled even after a reboot.
4. A, D. The NAT gateway allows outbound requests to the external API to succeed while preventing inbound requests from the internet. Configuring the security group to allow only inbound requests from your web servers allows outbound requests to succeed because the default rule for the security group allows outbound requests to the APIs that your web service needs. Option B is incorrect because security group rules cannot explicitly deny traffic; they can only allow it. Option C is incorrect because network ACLs are stateless, and this rule would prevent all of the replies to your outbound web requests from entering the public subnet.
5. C. You are in full control over the software on your instance. The default user that was created when the instance launched has full control over the guest operating system and can install the necessary software. Instance profiles are unrelated to the software on the instance.
6. D. You can query the Amazon EC2 metadata service for this information. Networking within the Amazon Virtual Private Cloud (Amazon VPC) is based on private IP addresses, so this rules out options A and B. Because the metadata service is available, you are not required to use a third-party service, which eliminates option C.
7. A. You can implement user data to execute scripts or directives that install additional packages. Even though you can use Amazon Simple Storage Service (Amazon S3) to stage software installations, there is no special bucket. You have full control of EC2 instances, including the software. AWS KMS is unrelated to software installation.
8. A. Amazon EC2 instances are resizable. You can change the RAM available by changing the instance type. Option B is incorrect because you can change this attribute only when the instance is stopped. Although option C is one possible solution, it is not required. Option D is incorrect because the RAM available on the host server does not change the RAM allocation for your EC2 instance.
9. A. AWS generates the default password for the instance and encrypts it by using the public key from the Amazon EC2 key pair used to launch the instance. You do not select a password when you launch an instance. You can decrypt this with the private key. IAM users and IAM roles are not for providing access to the operating system on the Amazon EC2 instance.
10. A, B, E. For an instance to be directly accessible as a web server, you must assign a public IP address, place the instance in a public subnet, and ensure that the inbound security group rules allow HTTP/HTTPS. A public subnet is one in which there is a direct route to an internet gateway. Option C defines a private subnet. Because security groups are stateful, you are not required to set the outbound rules—the replies to the inbound request are automatically allowed.

11. A, D. You can use an AMI as a template for launching any number of Amazon EC2 instances. AMIs are available for various versions of Windows and Linux. Option B is false because AMIs are local to the region in which they were created unless they are explicitly copied. Option C is false because, in addition to AWS-provided AMIs, there are third-party AMIs in the marketplace, and you can create your own AMIs.
12. B, D. Option B is true; Amazon Elastic Block Store (Amazon EBS) provides persistent storage for all types of EC2 instances. Option D is true because hardware accelerators, such as GPU and FGPA, are accessible depending on the type of instance. Option A is false because instance store is provided only for a few Amazon EC2 instance types. Option C is incorrect because Amazon EC2 instances can be resized after they are launched, provided that they are stopped during the resize. Hardware accelerators, such as GPU and FGPA, are accessible depending on the type of instance.
13. B, D. Only instances in the running state can be started, stopped, or rebooted.
14. D. Both the web server and the database are running on the same instance, and they can communicate locally on the instance. Option A is incorrect because security groups apply to only network traffic that leaves the instance. Option C is incorrect because network ACLs apply only to traffic leaving a subnet. Similarly, option B is incorrect because the public IP address is required for inbound requests from the internet but is not necessary for requests local to the same instance.
15. C. A public subnet is one in which there is a route that directs internet traffic (0.0.0.0/0) to an internet gateway. None of the other routes provides a direct route to the internet, which is required to be a public subnet.
16. D. A private subnet that allows outbound internet access must provide an indirect route to the internet. This is provided by a route that directs internet traffic to a NAT gateway or NAT instance. Option C is incorrect because a route to an internet gateway would make this a public subnet with a direct connection to the internet. The remaining options do not provide access to the internet.
17. D. Amazon VPC Flow Logs have metadata about each traffic flow within your Amazon VPC and show whether the connection was accepted or rejected. The other responses do not provide a log of network traffic.
18. C. Amazon CloudWatch is the service that tracks metrics, including CPU utilization for an Amazon EC2 instance. The other services are not responsible for tracking metrics.
19. B. EBS volumes provide persistent storage for an Amazon EC2 instance. The data is persisted until the volume is deleted and therefore persists on the volume when the instance is stopped.
20. F. You can install any software you want on an Amazon EC2 instance, including any interpreters required to run your application code.
21. B, C. Web requests are typically made on port 80 for HTTP and port 443 for HTTPS. Because security groups are stateful, you must set only the inbound rule. Options A and D are unnecessary because the security group automatically allows the outbound replies to the inbound requests.

- 22.** B, D. The customer is responsible for the guest operating system and above. Options C and E fall under AWS responsibility. AWS is responsible for the virtualization layer, underlying host machines, and all the way down to the physical security of the facilities.

## Chapter 3: Hello, Storage

- 1.** D. Amazon EC2 instance store is directly attached to the instance, which will give you the lowest latency between the disk and your application. Instance store is also provided at no additional cost on instance types that have it available, so this is the lowest-cost option. Additionally, since the data is being retrieved from somewhere else, it can be copied back to an instance as needed.

Option A is incorrect because Amazon S3 cannot be directly mounted to an Amazon EC2 instance.

Options B and C are incorrect because Amazon EBS and Amazon Elastic File System (Amazon EFS) would be a higher-cost option with higher latency than instance store.

- 2.** D, E. Objects are stored in buckets and contain both data and metadata.

Option A is incorrect because Amazon S3 is object storage, not block storage.

Option B is incorrect because objects are identified by a URL generated from the bucket name, service region endpoint, and key name.

Option C is incorrect because Amazon S3 object can range in size from a minimum of 0 bytes to a maximum of 5 TB.

- 3.** B. The volume is created immediately, but the data is loaded lazily, meaning that the volume can be accessed upon creation, and if the data being requested has not yet been restored, it will be restored upon first request.

Options A and C are incorrect because it does not matter what the size of the volume is or the amount of the data that is stored on the volume. Lazy loading will get data upon first request as needed while the volume is being restored.

Option D is incorrect because an Amazon EBS-optimized instance provides additional, dedicated capacity for Amazon EBS I/O. This minimizes contention, but it does not increase or decrease the amount of time before the data is made available while restoring a volume.

- 4.** A, B, D. Option C is incorrect because Amazon S3 is accessible through a URL. Amazon EFS is an AWS service that can be mounted to the file system of multiple Amazon EC2 instances. Amazon S3 can be accessible to multiple EC2 instances, but not through a file system mount.

Option E is incorrect because, unlike Amazon EBS volumes, storage in a bucket does not need to be pre-allocated and can grow in a virtually unlimited manner.

- 5.** A, C. Amazon Simple Storage Service Glacier is optimized for long-term archival storage and is not suited to data that needs immediate access or short-lived data that is erased within 90 days.

6. B. Option B is correct because pre-signed URLs allow you to grant time-limited permission to download objects from an Amazon S3 bucket.

Option A is incorrect because static web hosting requires world-read access to all content.

Option C is incorrect because AWS IAM policies do not know who are the authenticated users of your web application, as these are not IAM users.

Option D is incorrect because logging can help track content loss, but not prevent it.

7. A, D. Option A is correct because the data is automatically replicated within an availability zone.

Option D is correct because Amazon EBS volumes persist when the instance is stopped.

Option B is incorrect. There are no tapes in the AWS infrastructure.

Option C is incorrect because Amazon EBS volumes can be encrypted upon creation and used by an instance in the same manner as if they were not encrypted.

8. C. The Max I/O performance mode is optimized for applications where tens, hundreds, or thousands of EC2 instances are accessing the file system. It scales to higher levels of aggregate throughput and operations per second with a trade-off of slightly higher latencies for file operations.

Option A is incorrect because the General-Purpose performance mode in Amazon EFS is appropriate for most file systems, and it is the mode selected by default when you create a file system. However, when you need concurrent access from 10 or more instances to the file system, you may need to increase your performance.

Option B is incorrect. This is an option to increase I/O throughput for Amazon EBS volumes by connecting multiple volumes and setting up RAID 0 to increase overall I/O.

Option D is incorrect. Changing to a larger instance size will increase your cost for compute, but it will not improve the performance for concurrently connecting to your Amazon EFS file system from multiple instances.

9. A, B, D. Options A, B, and D are required, and optionally you can also set a friendly CNAME to the bucket URL.

Option C is incorrect because Amazon S3 does not support FTP transfers.

Option E is incorrect because HTTP does not need to be enabled.

10. C. A short period of heavy traffic is exactly the use case for the bursting nature of general-purpose SSD volumes—the rest of the day is more than enough time to build up enough IOPS credits to handle the nightly task.

Option A is incorrect because to set up a Provisioned IOPS SSD volume to handle the peak would mean overprovisioning and spending money for more IOPS than you need during off-peak time.

Option B is incorrect because instance stores are not durable.

Option D is incorrect because magnetic volumes cannot provide enough IOPS.

- 11.** C, D, E. Option A is incorrect because you store data in Amazon S3 Glacier as an archive. You upload archives into vaults. Vaults are collections of archives that you use to organize your data. Amazon S3 stores data in objects that live in buckets.  
Option B is incorrect because archives are identified by system-created archive IDs, not key names like in S3.
- 12.** A. Amazon EFS supports one to thousands of Amazon EC2 instances connecting to a file system concurrently.  
Options B and C are incorrect because Amazon EBS and Amazon EC2 instance store can be mounted only to a single instance at a time.  
Option D is incorrect because Amazon S3 does not provide a file system connection, but rather connectivity over the web. It cannot be mounted to an instance directly.
- 13.** B. There is no delay in processing when commencing a snapshot.  
Options A and C are incorrect because the size of the volume or the amount of the data that is stored on the volume does not matter. The volume will be available immediately.  
Option D is incorrect because an Amazon EBS-optimized instance provides additional, dedicated capacity for Amazon EBS I/O. This minimizes contention, but it does not change the fact that the volume will still be available while taking a snapshot.
- 14.** B, C, E. Amazon S3 bucket policies can specify a request IP range, an AWS account, and a prefix for objects that can be accessed.  
Options A and D are incorrect because bucket policies cannot be restricted by company name or country of origin.
- 15.** B, D. Option B is incorrect because Amazon S3 cannot be mounted to an Amazon EC2 instance like a file system.  
Option D is incorrect because Amazon S3 should not serve as primary database storage because it is object storage, not transactional block-based storage. Databases are generally stored on disk in one or more large files. If you needed to change one row in a database, the entire database file would need to be updated in Amazon S3, and every time you needed to access a record, you'd need to download the whole database.
- 16.** B, C, E. Option A is incorrect because static web hosting does not restrict data access. You can host a website on Amazon S3, but the bucket must have public read access, so everyone in the world will have read access to this bucket.  
Option B is correct because creating a presigned URL for an object optionally allows you to share objects with others.  
Option C is correct because Amazon S3 access control lists (ACLs) enable you to manage access to buckets and objects, defining which AWS accounts or groups are granted access and the type of access.  
Option D is incorrect because using an Amazon S3 lifecycle policy does not restrict data access. Lifecycle policies can be used to define actions for Amazon S3 to take during an object's lifetime (for example, transition objects to another storage class, archive them, or delete them after a specified period of time).

Option E is correct because a bucket policy is a resource-based AWS IAM policy that allows you to grant permission to your Amazon S3 resources for other AWS accounts or IAM users.

17. C, E. Option A is incorrect because even though you get increased redundancy with using cross-region replication, that does not protect the object from being deleted.

Option B is incorrect because vault locks are a feature of Amazon S3 Glacier, not a feature of Amazon S3.

Option D is incorrect because a lifecycle policy would move the object to Amazon Glacier, moving it out of your intended storage in S3 and reducing the time to access the data, and it does not prevent it from being deleted once it arrives in Amazon S3 Glacier.

C and E are correct. Versioning protects data against inadvertent or intentional deletion by storing all versions of the object, and MFA Delete requires a one-time code from a multi-factor authentication (MFA) device to delete objects.

18. C. To track requests for access to your bucket, enable access logging. Each access log record provides details about a single access request, such as the requester, bucket name, request time, request action, response status, and error code (if any). Access log information can be useful in security and access audits. It can also help you learn about your customer base and understand your Amazon S3 bill.

19. A, B, D. Option A is correct because cross-region replication allows you to replicate data between distance AWS Regions to satisfy these requirements.

Option B is correct because this can minimize latency in accessing objects by maintaining object copies in AWS Regions that are geographically closer to your users.

Option D is correct because you can maintain object copies in both regions, allowing lower latency by bringing the data closer to the compute.

Option C is incorrect because cross-region replication does not protect against accidental deletion.

Option E is incorrect because Amazon S3 is designed for 11 nines of durability for objects in a single region. A second region does not significantly increase durability.

20. C. If data must be encrypted before being sent to Amazon S3, client-side encryption must be used.

Options A, B, and D are incorrect because they use server-side encryption. This will only encrypt the data at rest in Amazon S3, not prior to transit to Amazon S3.

21. B. Data is automatically replicated across at least three Availability Zones within a single region.

Option A is incorrect because you can optionally choose to replicate data to other regions, but that is not done by default.

Option C is incorrect because versioning is optional, and data in Amazon S3 is durable regardless of turning on versioning.

Option D is incorrect because there are no tapes in the AWS infrastructure.

## Chapter 4: Hello, Databases

1. B, D, E. Amazon Relational Database Service (Amazon RDS) manages the work involved in setting up a relational database, from provisioning the infrastructure capacity to installing the database software. After your database is up and running, Amazon RDS automates common administrative tasks, such as performing backups and patching the software that powers your database. Option A is incorrect. Because Amazon RDS provides native database access, you interact with the relational database software as you normally would. This means that you're still responsible for managing the database settings that are specific to your application. Option C is incorrect. You need to build the relational schema that best fits your use case and are responsible for any performance tuning to optimize your database for your application's workflow and query patterns.
2. B. Amazon Neptune is a fast, reliable, fully managed graph database to store and manage highly connected datasets. Option A is incorrect because Amazon Aurora is a managed SQL database that is meant for transactional workloads that are ACID-compliant. Option C is incorrect because this is a managed NoSQL database service, which is meant for more key-value datasets with no relationships. Option D is incorrect because Amazon Redshift is a data warehouse that can be used for running analytical queries (OLAP) on data warehouses that are petabytes in scale.
3. B. NoSQL databases, such as Amazon DynamoDB, excel at scaling to hundreds of thousands of requests with key-value access to user profile and session. Option A is incorrect because the session state is typically suited for small amounts of data, and DynamoDB can scale more effectively with this type of dataset. Option C is incorrect because Amazon Redshift is a data warehouse service that is used for analytical queries on petabyte scale datasets, so it would not be a good solution. Option D is incorrect because DynamoDB provides scale, whereas MySQL on Amazon EC2 eventually becomes bottlenecked. Additionally, NoSQL databases are much faster and more scalable for this type of dataset.
4. A. 1 RCU = One strongly consistent read per second of 4 KB.  
15 KB is four complete chunks of 4 KB ( $4 \times 4 = 16$ ).  
So you need  $25 \times 4 = 100$  RCUs.
5. C. 1 RCU = Two eventually consistent reads per second of 4 KB.  
15 KB is four complete chunks of 4 KB ( $4 \times 4 = 16$ ).  
So you need  $(25 \times 4) / 2 = 50$  RCUs.
6. D. 1 WCU = 1 write per second of 1 KB (1024 bytes).  
512 bytes uses one complete chunk of 1 KB ( $512/1024 = 0.5$ , rounded up to 1).  
So you need  $100 \times 1 = 100$  WCUs.
7. B. Amazon DynamoDB Accelerator (DAX) is a write-through caching service that quickly integrates with DynamoDB with a few quick code changes. DAX will seamlessly intercept the API call, and your caching solution will be up and running in a short amount of

time. Option A is incorrect because you could implement your own solution; however, this would likely take a significant amount of development time. Option C is incorrect because your company would like to get the service up and running quickly. Implementing Redis on Amazon EC2 to meet your application's needs would take additional time. Option D is incorrect for many of the same reasons as option C, as time is a factor here. Additionally, your company would like to refrain from managing more EC2 instances, if possible.

8. B. With Amazon ElastiCache, only Redis can be run in a high-availability configuration. Option A is incorrect because this would add complexity to your architecture. It would also likely introduce additional latency, as the company is already using Amazon RDS. Option C is incorrect because ElastiCache for Memcached does not support a high-availability configuration. Option D is incorrect because DAX is a caching mechanism that is used for DynamoDB, not Amazon RDS.
9. C. Amazon Redshift is the best option. It is a managed AWS data warehouse service that allows you to scale up to petabytes worth of data, which would definitely meet their needs. Option A is incorrect because Amazon RDS cannot store that much data; the limit of Amazon RDS for Aurora is 64 TB. Option B is incorrect because DynamoDB is not meant for analytical-type queries—it is meant for simple queries and key-value pair data, which is more transactional based. You can query based on only the partition and sort key in DynamoDB. Option D is incorrect because Amazon ElastiCache is a caching solution that is meant for temporary data. However, you could store queries that ran in Amazon Redshift inside ElastiCache. This would improve the performance of frequently run queries, but by itself is not a solution.
10. A. Scans are less efficient than queries. When possible, always use queries with DynamoDB. Option B is incorrect because doing nothing isn't a good solution; the problem is unlikely to go away. Option C is incorrect because a strongly consistent read would actually be a more expensive query in terms of compute and cost. Strongly consistent reads cost twice as much as eventually consistent reads. Option D is incorrect because the concern is with reading data, not writing data. WCUs are write capacity units.

## Chapter 5: Encryption on AWS

1. B, D, E. Option A is incorrect because data can be encrypted in any location (on-premises or in the AWS Cloud). Option C is incorrect because encryption keys should be stored in a secured hardware security module (HSM). Option B is correct because there must be data to encrypt in order to use an encryption system. Option D is correct because tools and a process must be in place to perform encryption. Option E is correct because encryption requires a defined algorithm.
2. A, C. Option B is incorrect because KMI does not have a concept of a data layer. Option D is incorrect because KMI does not have a concept of an encryption layer. Option A is correct because the storage layer is responsible for storing encryption keys. Option C is correct because the management layer is responsible for allowing authorized users to access the stored keys.

3. A, C, D. Option A is correct because this is a common method to offload the responsibility of key storage while maintaining customer-owned management processes. Option C is correct because customers can use this approach to fully manage their keys and KMI. Option D is correct because AWS Key Management Service (AWS KMS) supports both encryption and KMI. Option B is incorrect because this would imply significant overhead to manage the storage while not providing customer benefits.
4. D. Option A is incorrect; with SSE-S3, Amazon S3 is responsible for encrypting the objects, not AWS KMS. Option B is incorrect because the customer provides the key to the Amazon S3 service. Option C is incorrect because the question specifically states that server-side encryption is used. Option D is correct because none of the other options listed server-side encryption with AWS KMS (SSE-KMS), whereby AWS KMS manages the keys.
5. B. Option A is incorrect. AWS KMS does not currently support asymmetric encryption. Option B is correct because AWS CloudHSM supports both asymmetric and symmetric encryption. Options C and D are incorrect because CloudHSM supports asymmetric encryption.
6. A, B. Option A is correct because AWS KMS uses AES-256 as its encryption algorithm. Option B is correct because CloudHSM supports a variety of symmetric encryption options. Options C and D are incorrect because AWS KMS and CloudHSM support symmetric encryption options.
7. C. Option A is incorrect because the organization does *not* want to manage any of the encryption keys. With AWS KMS, it will have to create customer master keys (CMKs). Option B is incorrect because by using customer-provided keys, the organization would have to manage the keys. Option C is correct because Amazon S3 manages the encryption keys and performs rotations periodically. Option D is incorrect because SSE-S3 provides this option.
8. C. Option A is incorrect because AWS KMS provides a centralized key management dashboard; however, this feature does not leverage CloudHSM. Option B is incorrect because you want to use AWS KMS *with* CloudHSM and not use it as a replacement for AWS KMS. Option C is correct because custom key stores allow AWS KMS to store keys in an CloudHSM cluster. Option D is incorrect because S3DistCp is a feature for Amazon Redshift whereby it copies data from Amazon S3 to the cluster.
9. A. Option A is correct because AWS KMS provides the simplest solution with little development time to implement encryption on an Amazon EBS volume. Option B is incorrect because even though you can use open source or third-party tooling to encrypt volumes, there would be some setup and configuration involved. Using CloudHSM would also require some configuration and setup, so option C is incorrect. Option D is incorrect because AWS KMS enables you to encrypt Amazon EBS volumes.
10. D. Options A, B, and C are incorrect because AWS KMS integrates with all these services.

# Chapter 6: Deployment Strategies

1. D. Option D is correct because AWS CodePipeline is a continuous delivery service for fast and reliable application updates. It allows the developer to model and visualize the software release process. CodePipeline automates your build, test, and release process when there is a code change.

Option A is incorrect because AWS CodeCommit is a secure, highly scalable, managed source control service that hosts private Git repositories.

Option B is incorrect because AWS CodeDeploy automates code deployments to any instance and handles the complexity of updating your applications.

Option C is incorrect because AWS CodeBuild compiles source code, runs tests, and produces ready-to-deploy software packages.
2. A, B, C, D. A, B, C, and D are correct because you can use them all to create a web server environment with AWS Elastic Beanstalk.

Option E is incorrect because AWS Lambda is an event-driven, serverless computing platform that runs code in response to events. Lambda automatically manages the computing resources required by that code.
3. C. Elastic Beanstalk supports Java, Node.js, and Go, so options A, B, and D are incorrect. It does not support Objective C, so option C is the correct answer.
4. A. Elastic Beanstalk deploys application code and the architecture to support an environment for the application to run.
5. A, C. Elastic Beanstalk supports Linux and Windows. No support is available for an Ubuntu-only operating system, Fedora, or Jetty.
6. A, B. Elastic Beanstalk can run Amazon EC2 instances and build queues with Amazon SQS.
7. A, B. Elastic Beanstalk can access Amazon S3 buckets and connect to Amazon RDS databases. It cannot install Amazon GuardDuty agents or create or manage Amazon WorkSpaces.
8. C. By using IAM policies, you can control access to resources attached to users, groups, and roles.
9. B, C. Elastic Beanstalk creates a service role to access AWS services and an instance role to access instances.
10. C. Elastic Beanstalk runs at no additional charge. You incur charges only for services deployed.
11. D. Charges are incurred for all accounts that use the allocated resources.
12. C. An existing Amazon RDS instance is deleted if the environment is deleted. There is no auto-retention of the database instance. You must create a snapshot to retain the data and to restore the database.

## Chapter 7: Deployment as Code

1. A. Options B and D are incorrect because the deployment is already in progress, and this would not be possible if the AWS CodeDeploy agent had not been installed and running properly. The CodeDeploy agent sends progress reports to the CodeDeploy service. The service does not attempt to query instances directly, and the Amazon EC2 API does not interact with instances at the operating system level. Thus, option C is incorrect, and option A is correct.
2. B. Option B is correct because the `ApplicationStop` lifecycle event occurs before any new deployment files download. For this reason, it will not run the first time a deployment occurs on an instance. Option C is incorrect, as this is a valid lifecycle event. Option A is incorrect. Option D is incorrect because lifecycle hooks are not aware of the current state of your application. Lifecycle hook scripts execute any listed commands.
3. A. Option B requires precise timing that would be overly burdensome to add to a CI/CD workflow. Option C would not include edge cases where both sources are updated within a small time period and would require separate release cadences for both sources. Option D is incorrect, as AWS CodePipeline supports multiple sources. When multiple sources are configured for the same pipeline, the pipeline will be triggered when any source is updated.
4. C. Option A is incorrect because storing large binary objects in a Git-based repository can incur massive storage requirements. Any time a binary object is modified in a repository, a new copy is saved. Comparing cost to Amazon S3 storage, it is more expensive to take this approach. By building the binary objects into an Amazon Machine Image (AMI), you are required to create a new AMI any time changes are made to the objects; thus, option B is incorrect. Option D and E introduce unnecessary cost and complexity into the solution. By using both an AWS CodeCommit repository and Amazon S3 archive, the lowest cost and easiest management is achieved.
5. D. Option A is incorrect because rolling deployments without an additional batch would result in less than 100 percent availability, as one batch of the original set of instances would be taken out of circulation during the deployment process. Option B is incorrect because if you add an additional batch, it would ensure 100 percent availability at the lowest cost but would require a longer update process than replacing all instances at once. Option C is incorrect because, by default, blue/green deployments will leave the original environment intact, accruing charges until it is manually deleted. Option D is correct as immutable updates would result in the fastest deployment for the lowest cost. In an immutable update, a new Auto Scaling group is created and registered with the load balancer. Once health checks pass, the existing Auto Scaling group is terminated.
6. D. Option C is incorrect because Amazon S3 does not have a concept of service roles. When a pipeline is initiated, it is done in response either to a change in a source or when a previous change is released by an authorized AWS IAM user or role. However, after the pipeline has been initiated, the AWS CodePipeline service role is used to perform pipeline actions. Thus, options A and B are incorrect. Option D is correct, because the pipeline's service role requires permissions to download objects from Amazon S3.

7. B. Option A is incorrect because this output is used only in the CodeBuild console. Option D is incorrect because CodeBuild natively supports this functionality. Though option C would technically work, CodeBuild supports output artifacts in the `buildspec.yml` specification. The BuildSpec includes a `files` directive to indicate any files from the build environment that will be passed as output artifacts. Thus, option B is correct.
8. C. Option A is incorrect because a custom build environment would expose the secrets to any user able to create new build jobs using the same environment. Option B is also incorrect. Though uploading the secrets to Amazon S3 would provide some protection, administrators with Amazon S3 access may still be able to view the secrets. Option D is incorrect because AWS does not recommend storing sensitive information in source control repositories, as it is easily viewed by anyone with access to the repository. Option D is correct. By encrypting the secrets with AWS KMS and storing them in AWS Systems Manager Parameter Store, you ensure that the keys are protected both at rest and in transit. Only AWS IAM users or roles with permissions to both the key and parameter store would have access to the secrets.
9. A. Options B, C, D, and E are incorrect. AWS Lambda functions can execute as part of a pipeline only with the `Invoke` action type.
10. A, B. Options D and E are incorrect because FIFO/LIFO are not valid pipeline action configurations. Option C is incorrect because pipeline stages support multiple actions. Pipeline actions can be specified to occur both in series and in parallel within the same stage. Thus, options A and B are correct.
11. D. Option A is incorrect because it will only create or update a stack, not delete the existing stack. Option B is incorrect because the desired actions are in the wrong order. Option C is incorrect because the final action, “Replace a failed stack,” is not needed. Option D is correct. Only two actions are required. First, the stack must be deleted. Second, the replacement stack can be created. Unless otherwise required, however, both actions can be essentially accomplished by using one “Create or update a stack” action.
12. D. Option A is incorrect. AWS CodeCommit is fully compatible with existing Git tools, and it also supports authentication with AWS Identity and Access Management (IAM) credentials. Options B and C are incorrect. These are the only protocols over which you can interact with a repository. You can use the CodeCommit credential helper to convert an IAM access key and secret access key to valid Git credentials for SSH and HTTPS authentication. Thus, option D is correct.
13. C. Options A, B, and D are all valid Amazon Simple Notification Service (Amazon SNS) notification event sources for CodeCommit repositories. Option C is correct because Amazon SNS notifications cannot be configured to send when a commit is made to a repository.
14. C, E. Options A, B, and D are incorrect because these action types do not support CodeBuild projects. Options C and E are correct because CodeBuild projects can be executed in a pipeline as part of build and test actions.
15. D. Environment variables in CodeBuild projects are not encrypted and are visible using the CodeBuild API. Thus, options A, B, and C are incorrect. If you need to pass sensitive information to build containers, use Systems Manager Parameter Store instead. Thus, option D is correct.

16. A. Because AWS does not have the ability to create or destroy infrastructure in customer data centers, options B, C, and D are incorrect. Option A is correct because on-premises instances support only in-place deployments.
17. C. Options A and B are incorrect because AWS CodeDeploy will not modify files on an instance that were not created by a deployment. Option D is incorrect because this approach could result in failed deployments because of missing settings in your configuration file. Option C is correct. By default, CodeDeploy will not remove files that it does not manage. This is maintained as a list of files on the instance.
18. C. Option A is incorrect because function versions cannot be modified after they have been published. Option B is also incorrect because function version numbers cannot be changed. Aliases can be used to point to different function versions; however, the alias itself cannot be overwritten (it is a pointer to a function version). Thus, option D is incorrect. AWS Lambda does not support in-place deployments. This is because, after a function version has been published, it cannot be updated. Option C is correct.
19. C. AWS CodePipeline requires that every pipeline contain a source stage and at least one build or deploy stage. Thus, the minimum number of stages is 2.
20. C. Option A is not correct because deleting the old revisions will temporarily resolve the issue. However, future deployments will continue to consume disk space. The same reasoning applies to options B and D, which are also temporary solutions to the problem. The CodeDeploy agent configuration file includes a number of useful settings. Among these, a limit can be set on how many revisions to store on an instance at any point in time. Thus, option C is correct.

## Chapter 8: Infrastructure as Code

1. D. Only the Resources section of a template is required. If this section is omitted, AWS CloudFormation has no resources to manage. However, a template does not require Parameters, Metadata, or AWSTemplateFormatVersion. Thus, options A, B, C, and E are incorrect.
2. E. The return value of the Ref intrinsic function for an AWS::ElasticLoadBalancing:: LoadBalancer resource is the load balancer name, which is not valid in a URL, so option A is incorrect. Since the application server instances are in a private subnet, neither will have a public DNS name; thus, option B is incorrect. Option C uses incorrect syntax for the Ref intrinsic function. Option D attempts to output a URL for the database instance. Thus, option E is correct.
3. A, C, D. If account limits were preventing the launch of additional instances, the stack creation process would fail as soon as AWS CloudFormation attempts to launch the instance (the Amazon EC2 API would return an error to AWS CloudFormation in this case). Thus, option B is incorrect. Any issues preventing the instance from calling cfn-signal and sending a success/failure message to AWS CloudFormation would cause the creation policy to time out. Thus, options A, C, and D are correct answers.

4. C. Option A is incorrect because AWS CloudFormation does not monitor the status of your database and would not be able to determine whether the database is corrupted. It also does not track whether there are currently running transactions before attempting updates. Thus, option E is incorrect. If an invalid update is submitted, the stack generates an error message when attempting the database update. Thus, option D is incorrect. Though option B would work, it is not needed to remove the database from the stack and manage it separately. Option C is correct because an AWS CloudFormation service role extends the default timeout value for stack actions to allow you to manage resources with longer update periods.
5. A. Custom resource function permissions are obtained by a function execution role, not the service role invoking the stack update; thus, option B is incorrect. When the AWS Lambda function corresponding to a custom resource no longer exists, the custom resource will fail to update immediately; thus, option C is incorrect. However, if the custom resource function is executed but does not provide a response to the AWS CloudFormation service endpoint, the resource times out with the aforementioned error. Thus, option A is correct.
6. A. AWS CloudFormation processes transformations by creating a change set, which generates an AWS CloudFormation supported template. Without the `AWS::Serverless` transform, AWS CloudFormation cannot process the AWS SAM template. For any stack in your account, the current template can be downloaded using the `get-stack-template` AWS CLI command. This command will return templates as processed by AWS CloudFormation; thus, option B is incorrect. Option C is also incorrect, because the original template is not saved before executing the transform. Option D is also incorrect, as AWS CloudFormation saves the current template for all stacks.
7. E. AWS SAM supports other AWS CloudFormation resources, and it is not limited to defining only `AWS::Serverless::*` resource types; thus, option D is incorrect, and option A is correct. However, the `AWS::Serverless` transform will not automatically associate serverless functions with `AWS::ApiGateway::RestApi` resources. The transform will automatically associate any functions with the serverless API being declared, or it will create a new one when the transform is executed. Thus, option B is also correct. Option C is also correct because AWS Serverless also supports Swagger definitions to outline the endpoints of your OpenAPI specification.
8. A. The `cfn-init` helper script is used to define which packages, files, and other configurations will be performed when an instance is first launched. The `cfn-signal` helper script is used to signal back to AWS CloudFormation when a resource creation or update has completed, so options B and C are incorrect. Option D is incorrect because `cfn-update`, is not a valid helper script. The `cfn-hup` helper script performs updates on an instance when its parent stack is updated. Thus, option A is correct.
9. C. Wait conditions accept only one signal and will not track additional signals from the same resource; thus, options A and B are incorrect. `WaitCount` is an invalid option type, so option D is incorrect. Option C is correct because creation policies enable you to specify a count and timeout.

- 10.** A. Options B and C will affect resources in your account. Option D would let you see the syntax differences between two template versions, but this does not indicate what type of updates will happen on the resources themselves. Thus, option D is incorrect. Change sets create previews of infrastructure changes without actually executing them. After reviewing the changes that will be performed, the change set can be executed on the target stack.
- 11.** B. Option A is incorrect, as this is a supported feature of nested stacks. Option C creates a circular dependency between the parent and child stacks (the parent stack needs to import the value from the child stack, which cannot be created until the parent begins creation). Option D is incorrect because cross-stack references are not possible without exporting and importing outputs. Option B uses intrinsic functions to access resource properties in the same manner as any other stack resource.
- 12.** B. AWS CloudFormation does not assume full administrative control on your account, and it requires permissions to interact with resources you own. AWS CloudFormation can operate using a service role; however, this must be explicitly passed as part of the stack operation. Otherwise, it will execute with the same permissions as the user performing the stack operation. Thus, option B is the correct answer.
- 13.** C. Because the reference to the Amazon DynamoDB table is made as part of an arbitrary string (the function code), AWS CloudFormation does not recognize this as a dependency between resources. To prevent any potential errors, you would need to declare explicitly that the function depends on the table. Thus, option C is correct.
- 14.** E. Replacing updates results in the deletion of the original resource and the creation of a replacement. AWS CloudFormation creates the replacement first with a new physical ID and verifies it before deleting the original. Because of this, option E is correct (all of the above).
- 15.** B, C. Option A is incorrect, as it states that no interruption will occur. Options D and E are not valid update types. Replacing updates delete the original resource and provision a replacement. Updates with some interruption have resource downtime, but the original resource is not replaced. Thus, options B and C are correct.
- 16.** A. The export does not need to be removed from the stack before it can be deleted, so option B is incorrect. Options C and D are also incorrect, as the stack does not need to be deleted. However, the stack cannot be deleted until any other stacks that import the value remove the import. Thus, option A is correct.
- 17.** B, D, E. If a stack update fails for any reason, the next state would be UPDATE\_ROLLBACK\_IN\_PROGRESS, which must occur before the rollback fails or completes. A stack that is currently updating can either complete the update, fail to update, or complete and clean up old resources. Thus, options B, D, and E are correct.
- 18.** B. Because the stack status shows the update has completed, you know that the update did not fail. This means that options A and D are incorrect. When a stack updates and resources are created, they will not be deleted unless the update fails. Thus, option C is incorrect. Old resources that are no longer required are removed during the cleanup phase. Thus, option B is correct.
- 19.** A, C. AWS CloudFormation currently supports JSON and YAML template formats only.

20. E. AWS CloudFormation provides a number of benefits over procedural scripting. The risk of human error is reduced because templates are validated by AWS CloudFormation before deployment. Infrastructure is repeatable and versionable using the same process as application code development. Individual users provisioning infrastructure need a reduced scope of permissions when using AWS CloudFormation service roles. Thus, option E is correct.
21. B. Option C is incorrect because, though on-premises servers can be part of a custom resource's workflow, they do not receive requests directly. Options D and E are incorrect because specific actions are not declared in custom resource properties. Option A is incorrect because AWS services themselves do not process custom resource requests. Specifically, Amazon SNS topics and AWS Lambda functions can act as recipients to custom resource requests. Thus, option B is correct.
22. C. Options A and B are incorrect because they would require interacting with other AWS services using the AWS CLI. For certain situations, such as running arbitrary commands in Amazon EC2 instance user data scripts, this would work. However, not all resource types have this ability. Option D is incorrect, as this is a built-in functionality of AWS CloudFormation. Option C is correct because any data that is declared in a custom resource response is accessible to the remainder of the template using the Fn::GetAtt intrinsic function.

## Chapter 9: Configuration as Code

1. E. You can raise all of the limits listed by submitting a limit increase request to AWS Support.
2. D. Option A is incorrect because instances do not attempt to download new cookbooks when performing Chef runs. Option B is incorrect because AWS OpsWorks Stacks does not have a concept of cookbook caching. Option C is incorrect because lifecycle events do not allow you to specify cookbook versions. Option D is correct because after updating a custom cookbook repository, any currently online instances will not automatically receive the updated cookbooks. To upload the modified cookbooks to the instances, you must first run the Update Custom Cookbooks command.
3. B. Options A, C, and D are incorrect because OpsWorks Stacks provides integration with Elastic Load Balancing to handle automatic registration and deregistration. Option B is correct as the Elastic Load Balancing layers for OpsWorks Stacks automatically register instances when they come online and deregister them when they move to a different state. You can also enable connection draining to prevent deregistration until any active sessions end.
4. A, B. Option C is incorrect because changing the cluster capacity will not affect service scaling. Option D is incorrect because submitting a replacement will result in the same behavior. If there are insufficient resources to launch replacement tasks when a service updates, Amazon Elastic Container Service (Amazon ECS) will continue to attempt to launch the tasks until it is able to do so. If you increase the cluster size, additional resources add to the pool to allow the new task to start. After it has done so, the old task will terminate. After it terminates, the cluster can scale back to its original size. If the downtime of this service does not concern you, set the minimum in-service percentage to 0 percent to allow Amazon ECS to terminate the currently running task before it launches the new one. Thus, options A and B are correct.

5. B. Options A, C, and D are incorrect because no other parties have access to the underlying clusters in AWS Fargate. When you use the Fargate launch type, AWS provisions and manages underlying cluster instances for your containers. You do not need to manage maintenance and patching. Thus, option B is correct.
6. A. Option B is incorrect, as this is a matter of personal preference. Option C is also incorrect because instances can be stopped and started individually, not only in layers at a time. Option D is incorrect because the configure lifecycle event runs on all instances in a stack, regardless of layer. Assigning recipes is performed at the layer level, meaning that all instances in the same layer will run the same configuration code. Organizing instances into layers based on purpose removes the need to add complex conditional logic. Thus, option A is correct.
7. C. Option A is incorrect because AWS OpsWorks Stacks does not include a central Chef Server. Option B is incorrect because storing recipes as part of an AMI would introduce considerable complexity for regular recipe code updates. Option D is incorrect because Amazon EC2 is not a valid storage location for cookbooks. A custom cookbook repository location is configured for a stack. When instances in the stack are first launched, they will download cookbooks from this location and run them as part of lifecycle events. Thus, option C is correct.
8. A. Option B is incorrect because you cannot associate a single Amazon RDS database instance with multiple stacks at the same time. Option C is incorrect because this approach would require manual snapshotting and data migration that is not necessary. Option D is incorrect. Migration of database instances between stacks is a common workflow. To migrate an Amazon RDS layer, you must remove it from the first layer before you add it to the second. Thus, option A is correct.
9. C. Option A is incorrect because 24/7 instances are normally recommended for constant demand. Option B is incorrect because load-based instances are recommended for variable, unpredictable demand changes. Option D is incorrect because On-Demand is an Amazon ECS instance type, not an OpsWorks Stacks instance type. You configure time-based instances to start and stop on a specific schedule. AWS recommends this for a predictable increase in workload throughout a day. Thus, option C is correct.
10. B. Option A is incorrect because 24/7 instances are normally recommended for constant demand. Option C is incorrect because time-based instances are recommended for changes in load that are predictable over time. Option D is incorrect because Spot is an Amazon ECS instance type, not an OpsWorks Stacks instance type. Option B is correct because load-based instances are recommended for unpredictable changes in demand.
11. A. Option B is incorrect because the Amazon ECS service role is used to create and manage AWS resources on behalf of the customer. Option C is incorrect because AWS Systems Manager is not part of Amazon ECS. Option D is incorrect because Amazon ECS automates the process of stopping and starting containers within a cluster. The Amazon ECS agent is responsible for all on-instance tasks such as downloading container images and starting or stopping containers. Thus, option A is correct.

12. B. Option A is incorrect. Though high availability is a tenet of SOA, it is not a requirement. Option C is incorrect because SOA does not define how development teams are organized. Option D is incorrect because SOA does not define what should or should not be procured from vendors. Service-oriented architecture involves using containers to implement discrete application components separately from one another to ensure availability and durability of each component. Thus, option B is correct.
13. D. A single task definition can describe up to 10 containers to launch at a time. To launch more containers, you need to create multiple task definitions. Task definitions should group containers by similar purpose, lifecycle, or resource requirements. Thus, option D is correct.
14. A. Option B is incorrect because PAT cannot be configured within your VPC (it must be configured using a proxy instance of some kind). Option C is incorrect because containers can be configured to bind to a random port instead of a specific one. Dynamic host port mapping allows you to launch multiple copies of the same container listening on different ports. Classic Load Balancers do not support dynamic host port mapping. Thus, option D is incorrect. Option A is correct because the Application Load Balancer is then responsible for mapping requests on one port to each container's specific port.
15. A. Options B and C are incorrect because they do not consider the Availability Zone of each cluster instance when placing tasks. Option D is incorrect because least cost is not a valid placement policy. The spread policy distributes tasks across multiple availability zones and cluster instances. Thus, option A is correct.

## Chapter 10: Authentication and Authorization

1. D. You need to use a third-party IdP as the confirmation of identity. Based on that confirmation, a policy can be assigned. Option A is incorrect because roles cannot be assigned to users outside of your account. Option B is incorrect because you cannot assign an IAM user ID to a user that is external to AWS. Option C is incorrect because it makes provisioning an identity a manual process.
2. D. An identity provider (IdP) answers the question “Who are you?” Based on this answer, policies are assigned. Those policies control the level of access to the AWS infrastructure and applications (if using AWS for managed services).  
Option A is incorrect; it is one of the functions of a service provider—to control access to applications. Option B is incorrect; policies are used to control access to APIs, which is how access to the AWS infrastructure is controlled. Option C is incorrect; identity providers do no error checking on policy assignment.
3. A. Where possible, using multi-factor authentication (MFA) minimizes the impact of lost or compromised credentials. Option B is incorrect in that embedding credentials is both a security risk and makes credential administration much more difficult. Option C would decrease the opportunity for misuse. It would not address any misuse that was a result of internal users. Option D is a good step but not as secure as option A.

4. D. If you want to use Security Assertion Markup Language (SAML) as an identity provider (IdP), use SAML 2.0. With Amazon Cognito, you can use Google (option A), Microsoft Active Directory (option B), and your own identity store (option C) as identity providers.
5. C. By using AWS Cloud services, such as Amazon Cognito, you are able to view the API calls in AWS CloudTrail. Amazon CloudWatch Logs are generated if you are using Amazon Cognito to control access to AWS resources. Option A is incorrect as AWS can act as an IdP for non-AWS services. Option B is incorrect in that Amazon CloudWatch allows you to monitor the creation and modification of identity pools. It will not show activity. Option D is incorrect because the service provider assigns the policies, not the identity provider (IdP).
6. A, C. AD Connector is easy to set up, and you continue to use the existing AD console to do configuration changes on Active Directory. Option B is incorrect because you cannot connect to multiple Active Directory domains with AD Connector, only a single one. AD Connector requires a one-to-one relationship with your on-premises domains. You can use AD Connector for AWS-created applications and services. Option D is incorrect because AD Connector is used to support AWS services.
7. A. To use AWS Single Sign-On (AWS SSO), you must set up AWS Organizations Service and enable all the features. AWS SSO uses Microsoft Active Directory (either AWS Managed Microsoft Active Directory or Active Directory Connector [AD Connector] but not Simple Active Directory). AWS SSO does not support Amazon Cognito. Option B is incorrect because AWS SSO does not use SAML. Options C and D are incorrect because you do not need to deploy either Simple AD or Amazon Cognito as a prerequisite for using AWS SSO.
8. C. Option C is correct because `GetFederationToken` returns a set of temporary security credentials (consisting of an access key ID, a secret access key, and a security token) for a federated user. You call the `GetFederationToken` action using the long-term security credentials of an IAM user. This is appropriate in contexts where those credentials can be safely stored, usually in a server-based application. Option D is incorrect because `GetSessionToken` provides only temporary security credentials. Option A is incorrect because `AssumeRole` is shorter lived (the default is 60 minutes; can be extended to 720 minutes). Options B and D are incorrect because `GetUserToken` and `GetSessionToken` are nonexistent APIs.
9. B. Because it is a managed service, you are not able to access the Amazon EC2 instances directly running AWS Managed Microsoft AD. AWS Managed Microsoft AD provides for daily snapshots, monitoring, and the ability to sync with an existing on-premises Active Directory.
10. A. Amazon Active Directory Connector (AD Connector) allows you to use your existing RADIUS-based multi-factor authentication (MFA) infrastructure to provide authentication.

# Chapter 11: Refactor to Microservices

1. B. Option B is correct because a **Parallel** state enables you to execute several different execution paths at the same time in parallel. This is useful if you have activities or tasks that do not depend on each other and can execute in parallel. This can make your workflow complete faster. Option A is incorrect because it executes only one of the branches, not all. Option C is incorrect because it can execute one task, not multiple. Option D is incorrect because it waits and does not execute any tasks.
2. B. The messages move to the dead-letter queue if they have met the **Maximum Receives** parameter (the number of times that a message can be received before being sent to a dead-letter queue) and have not been deleted.
3. A. Amazon Simple Queue Service (Amazon SQS) attributes supports 256 KB messages. Refer to Table 11.2, Table 11.3, and Table 11.4.
4. B. Option B is correct because to send a message larger than 256 KB, you use Amazon SQS to save the file in Amazon S3 and then send a link to the file on Amazon SQS. Option A is incorrect because using the technique in option B, this is possible. Option C is incorrect because AWS Lambda cannot push messages to Amazon SQS that exceed the size limit of 256 KB. Option D is incorrect because it does not address the question.
5. C. Option C is correct if you need to send messages to other users. Create an Amazon SQS queue and subscribe all the administrators to this queue. Configure an Amazon CloudWatch event to send a message on a daily cron schedule into the Amazon SQS queue. Option A is not correct because Amazon SQS queues do not support subscriptions. Option B is not correct because the message is sent without any status information. Option D is not correct because AWS Lambda does not allow sending outgoing email messages on port 22. Email servers use port 22 for outgoing messages. Port 22 is blocked on Lambda as an anti-spam measure.
6. A. Amazon SNS supports the same attributes and parameters as Amazon SQS. Refer to Table 11.2, Table 11.3, and Table 11.4.
7. D. Option D is correct because there is no limit on the number of consumers as long as they stay within the capacity of the stream, which is based on the number of shards. For a single shard, the capacity is 2 MB of read or five transactions per second. Options A and B are incorrect because there is no limit on the number of consumers that can consume from the stream. Option C is incorrect because together the consumers can consume only 2 MB per second or five transactions per second.
8. C. Option C is correct because Amazon Kinesis Data Streams is a service for ingesting large amounts of data in real time and for performing real-time analytics on the data. Option A is not correct because you use Amazon SQS to ingest events, but it does not provide a way to aggregate them in real time. Option B is incorrect because Amazon SNS is a notification service that does not support ingesting. Option D is incorrect because Amazon Kinesis Data Firehose provides analytics; however, it has a latency of at least 60 seconds.

9. A. Options B, C, and D are incorrect because there are no guarantees about where the records for Washington and Wyoming will be relative to each other. They could be on the same shard, or they could be on different shards. Option A is correct because the records for Washington will not be distributed across multiple shards.
10. E. Option E is correct because all the options from A through D are correct. Options A, B, C, and D are all valid options for writing Amazon Kinesis Data Streams producers.

## Chapter 12: Serverless Compute

1. D. Option D is correct because it enables the company to keep their existing AWS Lambda functions intact and create new versions of the AWS Lambda function. When they are ready to update the Lambda function, they can assign the PROD alias to the new version. Option A is possible; however, this adds a lot of unnecessary work, because developers would have to update all of their code everywhere. Option B is incorrect because moving regions would require moving all other services or introducing latency into the architecture, which is not the best option. Option C is possible; however, creating new AWS accounts for each application version is not a best practice, and it complicates the organization of such accounts unnecessarily.
2. B. At the time of this writing, the maximum amount of memory for a Lambda function is 3008 MB.
3. A. At the time of this writing, the default timeout value for a Lambda function is 3 seconds. However, you can set this to as little as 1 second or as long as 300 seconds.
4. C. Options A, B, and D are all viable answers; however, the question asks what is the best *serverless* option. Lambda is the only serverless option in this scenario; therefore, option C is the best answer.
5. D. At the time of this writing, the maximum execution time for a Lambda function is 300 seconds (5 minutes).
6. A. At the time of this writing, Ruby is not supported for Lambda functions.
7. A. At the time of this writing, the default limit for concurrent executions with Lambda is set to 1000. This is a soft limit that can be raised. To do this, you must open a case through the AWS Support Center page and send a Server Limit Increase request.
8. C. There are two types of policies with Lambda: a function policy and an execution policy, or AWS role. A function policy defines which AWS resources are allowed to invoke your function. The execution role defines which AWS resources your function can access. Here, the function is invoked successfully, but the issue is that the Lambda function does not have access to process objects inside Amazon S3. Option A is not correct because a function policy is responsible for invoking or triggering the function; here, the function is invoked and executes properly. Option B is not correct, as the scenario states that the trust policy is valid. The execution policy or AWS role is responsible for providing Lambda with access to other services; thus, the correct answer is option C.

9. A. Option A is correct because Lambda automatically retries failed executions for asynchronous invocations. You can also configure Lambda to forward payloads that were not processed to a DLQ, which can be an Amazon SQS queue or Amazon SNS topic. Option B is incorrect because a VPC network is an AWS service that allows you to define your own network in the AWS Cloud. Option C is incorrect because this is dealing with concurrency issues, and here you have no problems with Lambda concurrency. Additionally, concurrency is enabled by default with Lambda. Option D is incorrect because Lambda does support SQS.
10. C. Option C is correct because the environment variables enable you to pass settings dynamically to your function code and libraries without changing your code. Option A is not correct, because dead-letter queries are used for events that could not be processed by Lambda and need to be investigated later. Option B is not correct because it can be done. Option D is incorrect because this can be accomplished through environment variables.

## Chapter 13: Serverless Applications

1. D. Option A is incorrect. While AWS CloudFormation can help you provision infrastructure, AWS Serverless Application Model (AWS SAM) is optimized for deploying AWS serverless resources by making it easy to organize related components and resources that operate on a single stack; therefore, option A is not the best answer. Option C is incorrect because AWS OpsWorks is managed by Puppet or Chef, which you can use to deploy infrastructure. However, these are not the optimal answers given that you are specifically looking for serverless technologies. The same is true for Ansible in option B. Option D is correct because AWS SAM is an open-source framework that you can use to build serverless applications on AWS.
2. B. CORS is responsible for allowing cross-site access to your APIs. Without it, you will not be able to call the Amazon API Gateway service. You use a stage to deploy your API, and a resource is a typed object that is part of your API's domain. Each resource may have an associated data model and relationships to other resources and can respond to different methods. Option A is incorrect because you do need to enable CORS. Option B is correct because CORS is responsible for allowing one server to call another server or service. For more information on CORS, see: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>. Option C is incorrect, as deploying a stage allows you to deploy your API. Option D is incorrect, as a resource is where you can define your API, but it is not yet deployed to a stage and "live."
3. A, C. There are three benefits to serverless stacks: no server management, flexible scaling, and automated high availability. Costs vary case by case. For these reasons, option A and option C are the best answers.
4. D. Option A is incorrect; API Gateway only supports HTTPS endpoints. Option B is incorrect because API Gateway does not support creating FTP endpoints. Option C is incorrect; API Gateway does not support SSH endpoints. API Gateway only creates HTTPS endpoints.

5. C. Option A is incorrect because Amazon CloudFront supports a variety of sources, including Amazon S3. Option B is incorrect, because serverless applications contain both static and dynamic data. Additionally, CloudFront supports both static and dynamic data. Option C is correct because CloudFront supports a variety of origins. For the serverless stack, it supports Amazon S3. Option D is incorrect because Amazon S3 is a valid origin for CloudFront.
6. D. Option A, option B, and option C are each not the only language/platform supported. Option D is correct because all of these languages/platforms are supported.
7. C. Option C is correct because Amazon Cognito supports SMS-based MFA.
8. D. Options A, B, and C are incorrect because Amazon Cognito supports device tracking and remembering.
9. A. Option A is correct because the events property allows you to assign Lambda to an event source. Option B is incorrect because handler is the function handler in an Lambda function. Option C is incorrect because context is the context object for a Lambda function. Option D is incorrect because runtime is the language that your Lambda function runs as.
10. D. Option A is incorrect. You can run React in an AWS service. Option B is incorrect. You can run your web server with Amazon S3. With option C, you do not need to load balance Lambda functions because Lambda scales automatically. Option D is correct. You can run a fully dynamic website in a serverless fashion. You can also use JavaScript frameworks such as Angular and React. The NoSQL database may need to be refactored to run in Amazon DynamoDB.

## Chapter 14: Stateless Application Patterns

1. B. Option B is correct because the maximum size of an item in an DynamoDB table is 400 KB. Option C is incorrect because 4 KB is the capacity of a strongly consistent read per second, or two eventually consistent reads per second, for an item up to 4 KB in size. Option D is incorrect because 1,024 KB is not the size limit of an DynamoDB item. The maximum item size is 400 KB.
2. C. Option C is correct because when creating a new bucket, the bucket name must be globally unique. Option A is incorrect because versioning is disabled by default. Option B is incorrect because the maximum size for an object stored in Amazon S3 is 5 TB, not 5 GB. Option D is incorrect because you cannot change a bucket name after you have created the bucket.
3. B. Option B is correct because storage class is the only factor that is not considered when determining which region to choose. Option A is incorrect because latency is a factor when choosing a bucket region. Option C is incorrect because prices are different between regions; thus, you might consider cost when choosing a bucket region. Option D is incorrect because you may be required to store your data in a bucket in a particular region based on legal requirements or compliance.

4. C. Option C is correct because the recommended technique for protecting your table data at rest is the server-side encryption. Option A is incorrect because fine-grained access controls are a mechanism for providing access to resources and API calls, but the mechanism is not used to encrypt or protect data at rest. Option B is incorrect because TLS protects data in transit, not data at rest. Option D is incorrect because client-side encryption is applied to data before it is transmitted from a user device to a server.
5. D. Option D is correct because versioning-enabled buckets enable you to recover objects from accidental deletion or overwrite. Option A is incorrect because lifecycle policies are used to transition data to a different storage class and do not protect objects against accidental overwrites or deletions. Option B is incorrect because enabling MFA Delete on the bucket requires an additional method of authentication before allowing a deletion. Option C is incorrect because using a path-style URL is unrelated to protecting overwrites or accidental deletions.
6. C, D. Options C and D are correct because Amazon S3 stores objects in buckets, and each object that is stored in a bucket is made up of two parts: the object itself and the metadata. Option A is incorrect because Amazon S3 stores data as objects, not in fixed blocks. Option B is incorrect because the size limit of an object is 5 TB.
7. C. Option C is correct because DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table, and the service stores this information in a log for up to 24 hours. Options A, B, and D are incorrect because 24 hours is the maximum time that data persists on an Amazon DynamoDB stream.
8. B. Option B is correct because DynamoDB Streams ensures that each stream record appears exactly once in the stream. Options A and C are incorrect because each stream record appears exactly once. Option D is incorrect because you cannot set the retention period.
9. A. Option A is correct because your bucket can be in only one of three versioning states: versioning-enabled, versioning-disabled, or versioning-suspended. Thus, versioning-paused is a state that is not a valid configuration. Options A, B, and C are incorrect—they are all valid bucket states for versioning.
10. A. Option A is correct because QueryTable is the DynamoDB operation used to find items based on primary key values. Option B is incorrect because UpdateTable is the DynamoDB operation used to modify the provisioned throughput settings, global secondary indexes, or DynamoDB Streams settings for a given table. Option C is incorrect because DynamoDB does not have a Search operation. Option D is incorrect because Scan is the DynamoDB operation used to read every item in a table.
11. A, B, C. Option D is incorrect because when compared to the other options, a bank balance is not likely to be stored in a cache; it is probably not data that is retrieved as frequently as the others are fetched. Options A, B, and C are all better data candidates to cache because multiple users are more likely to access them repeatedly. Although, you could also cache the bank account balance for shorter periods if the database query is not performing well.

12. A, D. Options A and D are correct because Amazon ElastiCache supports both the Redis and Memcached open-source caching engines. Option B is incorrect because MySQL is not a caching engine—it is a relational database engine. Option C is incorrect because Couchbase is a NoSQL database and not one of the caching engines that ElastiCache supports.
13. C. Option C is correct because the default limit is 20 nodes per cluster.
14. C. Option C is correct because ElastiCache is a managed in-memory caching service. Option A is incorrect because the description aligns more closely to the Elasticsearch Service. Option B is incorrect because this is not an accurate description of the ElastiCache service. Option D is incorrect because, as a managed service, ElastiCache does not manage Amazon EC2 instances.
15. B, D, E. Option B is correct because DynamoDB is a NoSQL low-latency transactional database that you can use to store state. Option D is correct because Amazon Elastic File System (Amazon EFS) is an elastic file system that you can also use to store state. Option E is correct because ElastiCache is an in-memory cache that is also a good solution for storing state. Option A is incorrect because Amazon CloudFront is a content delivery network that is used more for object caching, not in-memory caching. Option C is incorrect because Amazon CloudWatch is a metric repository and does not provide any kind of user-accessible storage. Option F is incorrect because Amazon SQS is used for exchanging messages.
16. C. Option C is correct because Amazon DynamoDB is a nonrelational database that delivers reliable performance at any scale. Option A is incorrect because Amazon S3 Glacier is for data archiving and long-term backup. It is also an object store and not a database store. Option B is incorrect because Amazon RDS is designed for relational workloads. Option D is incorrect because Amazon Redshift is a data warehousing service.
17. D. Option D is correct because local secondary indexes on a table are created when the table is created. Options A and C are incorrect because you can have five local secondary indexes or five global secondary indexes per table. Option B is incorrect because you can create global secondary indexes after you have created the table.

## Chapter 15: Monitoring and Troubleshooting

1. B. Option A is incorrect because you do not want to scale in to reduce your capacity when you are experiencing a high load. Option C is incorrect because you do not want to scale in to reduce your capacity when your application is taking a long time to respond. Option D is incorrect because metrics are required for triggering AWS Auto Scaling events. Option B is correct because scaling out should occur when more resources are being consumed than normal, and scaling in should occur when less resources are being consumed.

2. D. Options A, B, C, and D are all incorrect because data points with a period of 300 seconds are stored for 63 days in Amazon CloudWatch.
3. D. Option A is incorrect because AWS CloudTrail events show who made the request. Option B is incorrect because CloudTrail shows when the request was made, and option C is incorrect because CloudTrail shows what was requested. Option E is incorrect because CloudTrail shows what resource was acted on. Option D is correct because CloudTrail can provide no insight into why a request was made.
4. C. Option A would work; however, it is not the most cost-effective way because logs stored in CloudWatch cost more than logs stored in Amazon S3. Option B is incorrect because CloudWatch cannot ingest logs without access to your servers. Option C is correct because archiving logs from CloudWatch to Amazon S3 reduces overall data storage costs.
5. A, B, D. Option C is incorrect because CloudWatch has no way to access data in your applications or servers. You must push the data either by using the CloudWatch SDK or AWS CLI or by installing the CloudWatch agent. Option A is correct because the CloudWatch agent is required to send operating system and application logs to CloudWatch. Option B is likewise correct because metrics logs are sent to CloudWatch using the PutMetricData and PutLogEvents API actions. Option D is also correct because the AWS CLI can be used to send metrics to CloudWatch using the put-metric-data and put-log-events commands.
6. C. Options A and B are incorrect because the strings must match a filter pattern equal to 404. Option C is correct because 404 matches the error code present in the example logs.
7. A. AWS X-Ray color-codes the response types you get from your services. For 4XX, or client-side errors, the circle is orange. Thus, option B is incorrect. Application failures or faults are red, and successful responses, or 2XX, are green. Thus, options C and D are incorrect. For throttling, or 5XX series errors, the circle is purple. Thus, option A is correct.
8. C. Option A is incorrect because CloudTrail logs list security-related events and do not provide a dashboard feature. Option B is incorrect because CloudWatch alarms are used to notify you when something isn't operating based on your specifications. Option D is incorrect because Amazon CloudWatch Logs are for sending and storing server logs to the CloudWatch service; however, you could use these logs to create a metric and then place it on the CloudWatch dashboard. Option C is the correct answer. Use CloudWatch dashboards to create a single interface where you can monitor all the resources.
9. D. CloudTrail stores the CloudTrail event history for 90 days; however, if you would like to store this information permanently, you can create an CloudTrail trail, which stores the logs in Amazon S3.
10. D. Option C is incorrect because the LookupEvents API action can be used to query event data. Options A and B are also incorrect because the AWS CLI and the AWS Management Console use the same CloudTrail APIs to query event data. Thus, option D is correct.

11. B, D. Management events are operations performed on resources in your AWS account. Data events are operations performed on data stored in AWS resources. For example, modifying an object in Amazon S3 would qualify as a data event, and changing a bucket policy would qualify as a management event. Because options A, C, and E involve sending or receiving data, not modifying or creating AWS resources, they are data events. Thus, options B and D are correct.
12. A, C, D. When installing the CloudWatch Logs agent, no additional networking configuration is required as long as your instance can reach the CloudWatch API endpoint. Therefore, option B is incorrect. You can use AWS Systems Manager to install and start the agent, but it is not required to install the Systems Manager agent alongside the CloudWatch Logs agent; thus, option E is incorrect. When installing the agent, you must configure the specific logs to send. The agent must be started before new log data is sent to CloudWatch Logs.
13. A. CloudWatch alarms support triggering actions in Amazon EC2, EC2 Auto Scaling, and Amazon SNS. Thus, options B, C, and D are incorrect. It is possible to trigger AWS Lambda functions from an alarm, but only by first sending the alarm notification to an Amazon SNS topic. Thus, option A is correct.
14. D. CPU, network, and disk activity are metrics that are visible to the underlying host for an instance. Thus, options A, B, and C are incorrect. Because memory is allocated in a single block to an instance and is managed by the guest OS, the underlying host does not have visibility into consumption. This metric would have to be delivered to CloudWatch as a custom metric by using the agent. Thus, option D is correct.
15. A. No namespace starts with an Amazon prefix; therefore, options B and D are incorrect. Option C is incorrect because namespaces are specific to a service (Amazon EC2), not a resource (an instance). Option A is correct because the Amazon EC2 service uses the AWS prefix, followed by EC2.

## Chapter 16: Optimization

1. D. Amazon EC2 instance store is directly attached to the instance, which gives you the lowest latency between the disk and your application. Instance store is also provided at no additional cost on instance types that have it available, so this is the lowest-cost option. Additionally, because the data is being retrieved from somewhere else, it can be copied back to an instance as needed. Option A is incorrect because Amazon S3 cannot be directly mounted to an Amazon EC2 instance. Options B and C are incorrect because Amazon EBS and Amazon EFS would be higher-cost options, with a higher latency than an instance store.
2. C. GetItem retrieves a single item from a table. This is the most efficient way to read a single item because it provides direct access to the physical location of the item. Options A and B are incorrect. Query retrieves all the items that have a specific partition key. Within those items, you can apply a condition to the sort key and retrieve only a subset of the

data. Query provides quick, efficient access to the partitions where the data is stored. Scan retrieves all of the items in the specified table, and it can consume large amounts of system resources based on the size of the table. Option D is incorrect. DynamoDB is a nonrelational NoSQL database, and it does not support table joins. Instead, applications read data from one table at a time.

3. C. Option C is a fault-tolerance check. By launching instances in multiple Availability Zones in the same region, you help protect your applications from a single point of failure. Options A and B are performance checks. Provisioned IOPS volumes in the Amazon EBS are designed to deliver the expected performance only when they are attached to an Amazon EBS optimized instance. Some headers, such as Date or User-Agent, significantly reduce the cache hit ratio (the proportion of requests that are served from a CloudFront edge cache). This increases the load on your origin and reduces performance because CloudFront must forward more requests to your origin. Option D is a cost check. Elastic IP addresses are static IP addresses designed for dynamic cloud computing. A nominal charge is imposed for an Elastic IP address that is not associated with a running instance.
4. B. Options A, C, and D are incorrect because partition keys used in these options could cause “hot” (heavily requested) partition keys because of lack of uniformity. Design your application for uniform activity across all logical partition keys in the table and its secondary indexes. Use distinct values for each item.
5. D. Option A is incorrect because SQS is a messaging service. Option B is incorrect because SNS is a notification service. Option C is incorrect because CloudFront is a web distribution service. Option D is correct because ElastiCache improves the performance of your application by retrieving data from high throughput and low latency in-memory data stores. For details, see <https://aws.amazon.com/elasticsearch/>.
6. C. Option C is correct because CloudFront optimizes performance if your workload is mainly sending GET requests. There are also fewer direct requests to Amazon S3, which reduces cost. For details, see <https://docs.aws.amazon.com/AmazonS3/latest/dev/request-rate-perf-considerations.html>.
7. D. Option A is incorrect because AWS Auto Scaling is optimal for unpredictable workloads. Option B is incorrect because cross-region replication is better for disaster recovery scenarios. Option C is incorrect because DynamoDB streams are better suited to stream data to other sources. Option D is correct because Amazon DynamoDB Accelerator (DAX) provides fast in-memory performance. For details, see <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DAX.html>.
8. C. Option A is incorrect because EC2 instance store is too volatile to be optimal. Option B is incorrect because this is a security solution and will not impact performance positively. Option C is correct because ElastiCache is ideal for handling session state. You can abstract the HTTP sessions from the web servers by using Redis and Memcached. Option D is incorrect because compression is not the optimal solution given the choices. For details, see <https://aws.amazon.com/caching/session-management/>.
9. B. Option B is correct because lazy loading only loads data into the cache when necessary. This avoids filling up the cache with data that isn’t requested. Options A, C, and D are

incorrect because they do not match the requirement of the question. For details, see <https://docs.aws.amazon.com/AmazonElastiCache/latest/mem-ug/Strategies.html>.

10. A. Option A is correct because information about the instance, such as private IP, is stored in the instance metadata. Option B is incorrect because private IP information is not stored in the instance user data. Option C is incorrect because running `ifconfig` is manual and not automated. Option D is incorrect because it is not clear on what type of instance the application is running. For details, see <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html>.
11. D. Options A, B, and C are incorrect because they are not recommended best practices. Option D is correct because it is one of the recommendations in the best practices documentation, “Avoid using recursive code.” For details, see <https://docs.aws.amazon.com/lambda/latest/dg/best-practices.html>.
12. C. Option A is incorrect because changing the entire architecture is not ideal. Option B is incorrect because Multi-AZ is used for fault tolerance. Option C is correct because loads can be reduced by routing read queries from your application to the read replica. Option D is incorrect because using an Elastic Load Balancing load balancer will not reduce the query load. For details, see <https://aws.amazon.com/rds/details/read-relicas/>.
13. C. Option A is incorrect because this is relevant only when you need a static website. Option B is incorrect because changing the storage class does not help with latency. Option C is correct because cross-region replication maintains object copies in regions that are geographically closer to your users, reducing latency. Option D is incorrect because encryption is necessary only for securing data at rest. For details, see <https://docs.aws.amazon.com/AmazonS3/latest/dev/crr.html>.
14. B. Options A, C, and D are incorrect because they are not optimal for handling large object uploads to Amazon S3. Option B is correct because a multipart upload enables you to upload large objects in parts to Amazon S3. For details, see <https://docs.aws.amazon.com/AmazonS3/latest/dev/mpuoverview.html>.
15. C. Option A is incorrect because this is not the optimal approach for bootstrapping. Option B is incorrect because, while possible, bootstrapping in the user data is optimal. Option C is correct because instance user data is used to perform common automated configuration tasks and run scripts after boot. Option D is incorrect because bootstrapping is done in instance user data, not instance metadata. For details, see <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/user-data.html>.

# Index

---

## A

accelerated computing instances, 39  
access control  
  Amazon DynamoDB, IAM policy and, 732–735  
  Amazon EFS (Elastic File System), 776  
  Amazon S3  
    ACLs (access control lists), 124  
    bucket policies, 123  
    defense in depth, 124–125  
    user policies, 123–124  
  ElastiCache, 747  
access keys, 14, 16  
  KMI (key management infrastructure), 263  
ACLS (access control lists), 58–61, 105, 124  
AD Connector (Active Directory Connector), 506–507  
AD DS (Active Directory Domain Services), 506  
ADM (Amazon Device Messaging), 537  
Advanced Message Queuing Protocol.  
  *See* AMQP  
AES-256 (Advanced Encryption Standard), 95, 120, 187, 261, 263, 271, 272, 731  
AFR (annual failure rate), 93  
ALB (Application Load Balancer), 287, 479  
all-at-once deployment, 300  
Amazon API Gateway, 623, 627–628  
  Amazon CloudWatch and, 632–633  
  API keys, 631  
  authorizers, 630  
  AWS Lambda, integration, 631  
  CORS (cross-origin resource sharing), 631  
  definition support, 634  
  endpoints, 628  
  HTTP methods, 630  
  monitoring metrics, 807  
  OpenAPI specification, 634  
  resources, 629  
  RESTful APIs, 631  
  security, 633–634  
  stages, 630

## Amazon Aurora

  automatic scaling, 848  
  databases, 176  
  DB clusters, 190–191  
    cluster volume, 191  
    instances, 191  
  global databases, 192  
  serverless, 192  
Amazon Aurora Serverless, 642–643  
Amazon CloudFront  
  AWS Elastic Beanstalk and, 297–298  
  content delivery, 626–627  
Amazon CloudWatch, 189–190  
  alarms, 814–817  
  Amazon API Gateway and, 632–633  
  Amazon SQS, queue monitoring  
    and, 533  
  AWS Lambda functions, 602–603  
  cases, 800  
  cost management and, 867  
  dashboards, 817–818  
  log aggregation, 811–812  
  log processing, 814  
  log searches, metric filters, 812–814  
  metrics  
    aggregations, 804  
    Amazon API Gateway, 807  
    Amazon DynamoDB, 806  
    Amazon EC2, 805  
    Amazon Lambda, 807  
    Amazon S3, 806  
    Amazon SNS, 808  
    Amazon SQS, 808  
  AWS Auto Scaling groups, 805  
  built-in, 802  
  custom, 808–810  
  data points, 802–803  
  Elastic Load Balancing, 804  
  repository, 801  
    statistics, 803–804  
    statistics retrieval, 810–811  
  microservices, 521  
  monitoring, 798  
  performance monitoring, 868

- Amazon Cognito, 498, 505
- Amazon SNS, endpoints, 539
- authentication
  - device tracking, 636–637
  - identity pools, 639
  - multi-factor authentication (MFA), 636
  - password policies, 636
  - SDK, 639–640
  - SMS messages, 636
  - UI (user interface) customization, 637–639
  - user pools, 634–635
- authorizers, 630
- Amazon Device Messaging (ADM), 537
- Amazon DynamoDB, 569, 664
  - access control, fine grained, 214–216
  - adding to tables, 692
  - atomic counters, 715
  - attribute projects, 687–688
  - attributes, 197, 198–199, 669
  - AWS Auto Scaling, 848
  - automatic scaling, 707–711
  - backfilling, 693
  - backups, on-demand, 216, 737
  - base table, 688–689
  - best practices, 216–217
  - burst capacity, 682, 710
  - condition keys, 735–736
  - conditional writes, 716–717, 721
  - control plane operations, 678
  - data plane operations, 679–680
  - data retrieval, 209–212
  - data types, 669–671
  - deleting databases, 694
  - encryption, 216
  - encryption at rest, 730–732
  - error handling, 720, 721
  - expressions, 724–729
  - global secondary indexes, 686–687
  - hash attribute, 665
  - IAM and, 214–216
  - index key violations, 694
  - index name, 688–689
  - item attributes, 722–723
  - items, 198, 669, 715
  - local secondary indexes, 694–700
  - managing, 691
  - monitoring metrics, 806
  - nonrelational database, 177
  - NoSQL databases, 177
  - object persistent model, 214
  - optimistic locking, 713–714
  - partition key, 665–668
  - partitions, 197, 711–713
  - permissions, IAM policy conditions, 732–735
  - PITR (point-in-time-recovery), 738–739
  - primary key, 199–200, 665–666
  - provisioned throughput, 689–690
  - queries, 688
    - filter expressions, 730
    - key condition expressions, 729–730
    - read consistency, 730
  - range attributes, 666
  - read capacity units, 690
  - read consistency, 206, 207
  - read/write throughput, 207, 672–673
    - adaptive capacity, 209
    - burst capacity, 209
    - on-demand, 208
  - provisioned throughput, 208
  - RCU (read capacity unit), 207
  - reserved capacity, 208
  - WCU (write capacity unit), 207–208
- resource allocation, 693
- restore, point-in-time recovery, 216
- restores, on-demand, 737–738
- return values, 680–681
- scanning, 688–689
- secondary indexes, 201, 665, 683
  - alternate key, 684
  - base table, 683
  - configuration, 685
  - global secondary indexes, 202–205, 682, 684
  - local secondary indexes, 201–202, 204–205, 682, 684
- shards, 668
- sort key, 666
- state, 665, 678
- status in table, 692
- streams, 205
- synchronizing, 689
- tables, 197, 198, 665, 672
  - creating, 691–692
  - global, 212–213

- names, 665
- replica, 213
- state, 678
- tags, 714–715
- throttle capacity, 682
- throughput, provisioned, 672
  - capacity, reads/writes, 672–673
  - capacity unit consumption, 674–675
  - item sizes, 674–675
  - reads capacity unit (RCU)
    - consumption, 675–676
  - settings, 674
  - writes capacity unit (WCU)
    - consumption, 676–678
- TTL (time to live), 719–720
- version number, 713–714
- write capacity units, 690
- write cost, 690–691
- Amazon DynamoDB Local downloadable database, 214
- Amazon DynamoDB Streams, 665
  - API (application programming interface), 705
  - AWS Lambda triggers, 706–707
  - concurrency, 547
  - consumers, 546–547
  - cross-region replication, 701
  - data, retention limit, 705
  - endpoints, 701–702
  - Kinesis Adapter, 703–704
  - shards, 547
    - stream records, 700–701
      - shards, 703
    - streams, 702–704
  - use case, 546
- Amazon EBS (Elastic Block Store), 40, 93, 155, 157, 158
  - Amazon EFS comparison, 144
  - Amazon S3 comparison, 144
  - AWS OpsWorks Stacks, layers, 454
  - block storage, 87
  - Elastic Volumes, 94–95
  - encryption, 95–96, 265, 274
  - HDD (hard disk drive)-backed volumes, 93
    - SSD comparison, 94
  - instance store comparison, 143–144
  - performance optimization, 95–97
  - snapshots, 95
- SSD (solid-state drive)-backed volumes, 93–94
- storage, persistent, 40–41
- storage optimization, 855–857
- troubleshooting, 97
- use cases, 94
- Amazon EC2 (Elastic Compute Cloud), 38, 67, 91, 158, 235–236, 587
  - AD DS (Active Directory Domain Services), 506
  - Amazon VPC and, 67
  - answers to review questions, 887–890
  - Auto Scaling, 847–848
  - Availability Zones, 38
  - bare-metal access, 38
  - elastic network interfaces, 42
  - instance store, 97–99, 155
  - instance types, 39
  - instances
    - accelerated computing, 39
    - access, 43
    - CloudWatch, 50
    - compute optimized, 39
    - connecting to, 45–46
    - families, 39
    - general purpose, 39
    - key pairs, 43
    - lifecycle, 43–44
    - memory optimized, 39
    - monitoring, 50
    - storage optimized, 39
    - metadata, IMDS, 47–48
    - monitoring metrics, 805
    - on-premises AppSpec, 362–366
    - on-premises configuration, 359–361
    - primary network interfaces, 42
    - private IP addresses, 42
    - public IP addresses, 42
    - RDP (Remote Desktop Protocol) and, 43
    - security groups, 42
    - users, default, 43
    - VPC, default, 42
    - webpages, custom, 49–50
  - Amazon ECR (Elastic Container Repository), 476, 481, 487
  - Amazon ECS (Elastic Container Service), 38, 446
    - Amazon ECR, 476, 481
    - architecture, 473–474

- AWS CodePipeline and, 321, 482–483
- AWS Fargate, 475–476, 484
  - clusters, 472–475, 486
  - container agent, 481
  - containers, 476
    - deployment, 471–472
    - task definition, 477–478
- Docker, 471, 473, 474, 484
  - Docker containers, 476, 481
  - images, 476, 481
  - overview, 472
  - service limits, 482
  - services, 478–479
  - task definition, 476–478
  - task scheduling, 479–480
- Amazon ECS Service Discovery, 480
- Amazon EFS (Elastic File System), 136–137, 157, 773
  - access control, 776
  - Amazon EBS comparison, 144
  - Amazon S3 comparison, 144
  - authentication, 776
  - AWS DataSync, 139–140
  - AWS DX (Direct Connect) and, 775–776
    - data consistency, 776
    - file storage, 87
    - file sync, 139
    - file system, 137, 778–779
    - file system access, 137–139
  - IAM, user creation, 777
  - performance, 140–141, 779–780
  - resources, 777
  - scaling, throughput scaling, 780–781
  - security, 141–142
  - VPC, 773–775
- Amazon EKS (Elastic Container Service for Kubernetes), 38
- Amazon Elastic Container Service, 325
- Amazon ElastiCache. *See* ElastiCache
- Amazon EMR
  - encryption, 267–268
  - S3DistCp, 272
- Amazon Kinesis, 86
- Amazon Kinesis Data Analytics, 544–545, 569
  - Amazon Kinesis Data Firehose, 151–152, 158, 543–544, 569
  - Amazon Kinesis Data Streams, 540, 569
    - applications, 541
    - consumer options, 543
    - data blob, 541
    - Fluentd, 542
    - Flume, 542
    - Kinesis Video Streams, 542
    - messages, deleting, 541
    - open source tools, 542
    - partition key, 541
    - producers, 542–543
    - real-time analytics, 542
    - streams, names, 541
    - throughput, 541–542
  - Amazon Kinesis Video Streams, 545, 569
  - Amazon Lambda, monitoring metrics, 807
  - Amazon Lightsail, 38
  - Amazon Machine Image. *See* AMI (Amazon Machine Image)
  - Amazon MQ, 570
    - active/standby broker for high availability, 550
    - AMQP (Advanced Message Queuing Protocol), 551
    - single-instance broker, 550
  - Amazon Neptune, 231–232
    - graph database, 177
  - Amazon Polly, 5, 11–12
  - Amazon QLDB (Quantum Ledger Database), ledger database, 177
  - Amazon RDS (Relational Database Service), 55, 180, 238, 274
    - Amazon Aurora, 190–192
    - Amazon CloudWatch, 189–190
    - availability, 181–182
    - AWS Elastic Beanstalk and, 298
    - backups, 181, 185–186
    - best practices, 192–194
    - configuration, 181
    - database migration, 489
    - encryption, 187–188, 266–267
    - engines, 182–185
    - hosting and, 182
    - IAM DB authentication, 188–189
    - instances, 464

- Multi-AZ, 186–187, 238
- procurement, 181
- relational databases, 177
- security, 181–182
  - implementing, 193–194
- Amazon Redshift
  - 256-bit AES keys, 272
  - architecture, 220–222
  - AWS CloudHSM cluster master key, 272
  - AWS KMS cluster master key, 272–273
  - data warehouse, 177
  - loading data, 224
  - querying data, 224
  - Redshift Spectrum, 225–226
  - security, 224–225
  - snapshots, 224
  - table, 222–224
- Amazon Resource Name. *See ARN (Amazon Resource Name)*
- Amazon Route 53, domain names, 625–626
- Amazon S3 (Simple Storage Service), 10, 64, 157, 747, 782
  - access control, 123–125
  - Amazon EBS comparison, 144
  - Amazon EFS comparison, 144
  - authentication, 129
  - AWS CLI, 128
  - AWS CodePipeline and, 321
  - AWS Elastic Beanstalk and, 297
  - AWS explorers, 128
  - AWS SDKs, 128
  - buckets, 99–105, 155–156, 748–760
  - consistency model, 114–118, 755–756
  - CORS (cross-origin resource sharing), 107–108
  - CRR (cross-region replication), 127–128
  - data consistency, 156
  - data lake architecture, 129–130
  - encryption, 156, 264–265, 274
    - client-side, 121–123
    - data protection, 760
    - envelope encryption, 119–120
    - server-side, 271, 760
  - SSE (Server-Side Encryption), 120–121
  - lifecycle configuration, 157
  - MFA Delete, 127
  - monitoring metrics, 806
- object operations, 108–109, 765–770
- object storage, 87
- object tagging, key-value pairs, 106
- objects, 105, 761–765, 769, 783
- performance, 130–134
  - Amazon CloudFront, 133
  - GET requests and, 772
  - multipart uploads, 133
  - object key naming, 131–132
  - range GETs, 133
  - request rate and, 770–771
  - TCP scaling, 133–134
  - TCP selective acknowledgment, 133–134
- transfer acceleration, 132–133
- workloads and, 130, 771–772
- presigned URLs, 118
  - query string authentication, 125
- pricing, 134
- query string authentication, 125–126
- requests, 129
- serverless applications, 129
- stateless applications, 129
- static website, 126, 156, 623–624
- storage classes, 156
  - Amazon S3 Glacier, 111–113
  - Amazon S3 Standard, 109–110
  - comparison, 114
  - frequently access objects, 757–758
  - GLACIER, 759
  - infrequently access objects, 758
  - OneZone\_IA, 111
  - RRS (Reduced Redundancy Storage), 110
  - RTO (recovery time objective), 111
  - setting, 759
  - Standard\_IA, 110
- storage optimization, 853–855
- uses, 155
- values, large attribute, 772
- VPC (virtual private cloud) endpoints, 128
- web server, 622–623
- web traffic logs, 624–625

- encryption, 113
  - object storage, 87
  - objects, restoring, 113
  - RTO (recovery time objective), 111
  - Vault Lock, 111–112
    - vaults, 111
  - Amazon SNS (Simple Notification Service),
    - 325, 534, 569
  - Amazon SQS comparison, 540
  - API owner operations
    - `AddPermission`, 536
    - `CreateTopic`, 536
    - `DeleteTopic`, 536
    - `GetTopicAttributres`, 536
    - `ListSubscriptions`, 536
    - `ListSubscriptionsByTopic`, 536
    - `ListTopics`, 536
    - `RemovePermission`, 536
    - `SetTopicAttributes`, 536
  - API subscriber operations
    - `ConfirmSubscription`, 537
    - `ListSubscriptions`, 537
    - `Subscribe`, 537
    - `UnSubscribe`, 537
  - APIs, clean up, 537
  - billing, 539–540
  - clients, 534–535
  - device tokens, 538–539
  - DLQ (dead letter queue), 599
  - endpoints, 535
    - Amazon Cognito, 539
    - mobile, 538
    - proxy server, 539
  - Free Tier, 539–540
  - limits, 539–540
  - messages, topics, 534
  - mobile, 537–539
  - monitoring metrics, 808
  - registration IDs, 538–539
  - restrictions, 539–540
  - subscriptions, 534
  - topics, 534, 536
  - transport protocols
    - email, 537
    - Email-JSON, 537
    - HTTP, 537
    - HTTPS, 537
  - workflow, 535
- Amazon SQS (Simple Queue Service), 294, 523, 569
  - Amazon SNS comparison, 540
  - `ChangeMessageVisibility` action, 527
  - consumers, 523
  - `DelaySeconds` action, 528
  - `DeleteMessage` action, 528
  - distributed cluster of servers, 525
  - DLQ (dead letter queue), 599
  - log server, 524
  - `MessageRetentionPeriod` action, 528
  - messages
    - attributes, 532
    - storage, 525–526
  - monitoring metrics, 808
  - producers, 523
  - queue, 525
    - Amazon CloudWatch and, 533
    - dead-letter, 531–532
    - dead-letter queue, 528–530
    - dead-letter troubleshooting, 531
    - FIFO (first-in, first-out), 526, 529–530
    - SSE settings, 533
    - standard, 526
    - standard queues, 529–530
  - `ReceiveMessage` action, long polling, 526
  - `ReceiveMessageWaitTimeSeconds` action, 527
  - responses, 523
  - servers, distributed cluster, 525
  - `VisibilityTimeout` action, 526
  - `WaitTimeSeconds`, 527
- Amazon Timestream, time series
    - database, 177
  - Amazon VPC (Virtual Private Cloud), 38, 67, 268–269
    - CIDR notation, 51
    - connection types, 52
    - default, 42
    - DHCP (Dynamic Host Configuration Protocol), 63
    - IP addresses, 52–53
    - NAT (network address translation), 61–63
    - network ACLs (access control lists), 58–61
    - network traffic monitoring, 64

- route tables, 55–56
  - security groups, 56–58
  - stacks, AWS OpsWorks Stacks, 453
  - subnets, 54–55
  - AMI (Amazon Machine Image), 41–42, 506, 593
    - Amazon EBS and, 97
    - AWS Elastic Beanstalk, 306
  - AMQP (Advanced Message Queuing Protocol), 551
  - AFR (annual failure rate), 93
  - API keys, 631
  - APIs (application programming interfaces), 2
  - Amazon SNS
    - AddPermission, 536
    - ConfirmSubscription, 537
    - CreateTopic, 536
    - DeleteTopic, 536
    - GetTopicAttributres, 536
    - ListSubscriptions, 536, 537
    - ListSubscriptionsByTopic, 536
    - ListTopics, 536
    - RemovePermission, 536
    - SetTopicAttributes, 536
    - Subscribe, 537
    - UnSubscribe, 537
  - answers to review questions, 886–887
  - AWS Lambda functions, 589
  - AWS STS
    - AssumeRole, 503
    - AssumeRoleWithSAML, 504
    - AssumeRoleWithWebIdentity, 504
    - DecodeAuthorizationMessage, 504
    - GetCallerIdentity, 504
    - GetFederationToken, 504
    - GetSessionToken, 505
  - control plane, 497
  - credentials, 14–15
    - assigning, 48
  - endpoints, 10–12, 13
  - microservices, 521
  - requests, 6
  - responses, 7
  - APNS (Apple Push Notification Service), 537, 538–539
  - Application Load Balancer (See ALB (Application Load Balancer))
- applications
    - Amazon Kinesis Data Streams, 541–542
    - Amazon S3, 129
    - AWS OpsWorks Stacks, 459–460
    - capacity, 289
    - deployment, 288–289
    - mapping to AWS database service, 178
    - running on instances, 44–50
    - serverless, 129, 622
    - stateless, 129
  - AppSpec configuration file, 299
  - architecture
    - data lake, 129–130
    - three-tier, 282
      - versus* serverless stack, 640–642
  - ARN (Amazon Resource Name), 22–23
    - Amazon SNS, 536
    - AWS Lambda functions, 600
  - attributes, nested, 722–723
  - authentication
    - Amazon Cognito and, 634–640
    - answers to review questions, 905–907
    - versus* authorization, 496
    - control planes, 497
    - federation, 496
    - IAM, 19–20
    - MFA (multi-factor authentication), 15–16, 636
    - RDP, 497
    - SSH, 497
  - authoritative data, 90
  - authorization, 497–498
    - answers to review questions, 905–907
    - versus* authentication, 496
  - AWS SSO (Single Sign-On), 500–501
    - control planes, 497
    - cross-account access, 499
    - federation, 496
    - IAM, 19–20
    - permissions policy, 499
    - RDP, 497
    - source accounts, 499
    - SSH, 497
    - target accounts, 499
    - trust policy, 499
  - Auto Scaling, 845–849

- Availability Zones, 705
  - Amazon EC2 (Elastic Compute Cloud), 38
  - AWS Region, 9, 10
- AWS (Amazon Web Services)
  - cloud services, calling, 5–9
  - resource management, 4
  - SOAP support, 128–129
- AWS Amplify JavaScript library, 128
- AWS ASG (Auto Scaling Group), 383
- AWS Auto Scaling, 289
  - groups, 289–290
  - groups, monitoring metrics, 805
  - microservices, 521
- AWS Budgets, 2, 866
- AWS CLI (Command Line Interface), 3, 4, 128, 382
  - AWS Lambda functions, 589
  - credentials, assigning, 48
- AWS Cloud, 2, 86, 176, 284–287
- AWS Cloud9, 66, 334
- AWS CloudFormation, 382
  - application deployment, 289
  - AWS CloudFormation Designer, 406
  - AWS CodePipeline and, 321, 429–432
  - change sets, 384, 434–435
  - condition functions
    - `Fn::AND`, 398
    - `Fn::IF`, 398
    - `Fn::NOT`, 398
    - `Fn::OR`, 398
  - creation policies, 436
  - custom resource providers, 406–407
  - helper scripts
    - `cfn-get-metadata`, 425
    - `cfn-hup`, 425–426
    - `cfn-init`, 424
    - `cfn-signal`, 424–425
  - infrastructure and, 382–384
  - intrinsic functions
    - `Fn::Base64`, 395
    - `Fn::Cidr`, 395
    - `Fn::FindInMap`, 395
    - `Fn::GetAtt`, 396
    - `Fn::GetAZs`, 396
    - `Fn::Join`, 396–397
    - `Fn::Select`, 397
    - `Fn::Split`, 397
- Fn::Sub, 397–398
- Ref, 398
- metadata keys
  - `AWS::CloudFormation::Designer`, 405
  - `AWS::CloudFormation::Init`, 399–404
  - `AWS::CloudFormation::Interface`, 404–405
- overview, 382–383
- permissions, 385–386, 435
- resource relationships, 408, 435
- resources, 435, 408–411
- service limits, 429
- stacks, 384
  - `CREATE_COMPLETE`, 411
  - `CREATE_FAILED`, 412
  - `CREATE_IN_PROGRESS`, 412
  - `DELETE_COMPLETE`, 412
  - `DELETE_FAILED`, 412
  - `DELETE_IN_PROGRESS`, 412
  - deletion policies, 416–417
  - export output, 417–418
  - exports, 417
  - import output, 417–418
  - nested, 417, 418–419
  - policies, 420–422
  - `ROLLBACK_COMPLETE`, 412
  - `ROLLBACK_FAILED`, 412
  - `ROLLBACK_IN_PROGRESS`, 412
  - `UPDATE_COMPLETE`, 412
  - `UPDATE_COMPLETE_CLEANUP_IN_PROGRESS`, 412–413
  - `UPDATE_IN_PROGRESS`, 412
  - `UPDATE_ROLLBACK_COMPLETE`, 413
  - `UPDATE_ROLLBACK_COMPLETE_CLEANUP_IN_PROGRESS`, 413
  - `UPDATE_ROLLBACK_IN_PROGRESS`, 413
  - updates, 413–416, 436
- StackSets, 427–429
  - templates, 386–394, 435
  - wait conditions, 436
- AWS CloudFormation CLI, 422–423
- AWS CloudHSM, 262, 268–269
- AWS CloudTrail
  - events, 818–820
  - monitoring, 798
  - trails, 820

- AWS Code services, 318
  - AWS CodePipeline, 318
- AWS CodeBuild, 318, 319, 344–345, 373
  - AWS CodePipeline and, 321, 352
  - build environments, 350–351
  - build projects, 345–349
  - service limits, 351
- AWS CodeCommit, 292, 318, 319, 332–333, 372, 373
  - AWS CodePipeline and, 321, 344
  - branches, 341
  - commits, 339–340
  - credentials, 333–334
  - development tools, 334
  - files, 337
  - migration to, 341–343
  - pull requests, 337–338
  - repository, 335–337
  - service limits, 343
- AWS CodeDeploy, 299, 319, 352–353, 373
  - applications, 362
  - AppSpec file, 362–369
  - AWS CodeDeploy agent, 369–370
  - AWS CodePipeline and, 321, 371
  - deployment configurations, 359–361
  - deployment groups, 356–359
  - deployments, 354–356
  - in-place deployment, 300
  - revision, 353–354
  - service limits, 370
- AWS CodePipeline, 318, 319, 372
  - actions, 323
  - Amazon ECS and, 321
  - Amazon S3, 321
  - approval actions, 325–238
  - artifacts, 326–327
- AWS CloudFormation and, 321, 430–432
- AWS CodeBuild and, 321, 352
- AWS CodeCommit and, 321, 344
- AWS CodeDeploy and, 321, 371
- AWS Elastic Beanstalk and, 321
  - AWS Lambda, 321
  - AWS OpsWorks Stacks, 321
  - build actions, 324
  - CI/CD (continuous integration/continuous deployment), 318
  - deploy actions, 325
  - GitHub and, 324
- invoke actions, 326
- pipelines, 322, 330–332
- revisions, 322–323
- service limits, 329
- source actions, 323–324
- stages, 323
- tasks, 329–332
- test actions, 324
- transactions, 326–327
- workflow, 320
- AWS compute, 17
- AWS Config
  - AWS Elastic Beanstalk and, 298
  - tagging and, 836
- AWS Cost and Usage Report, cost management and, 866–867
- AWS Cost Explorer, cost management and, 865
- AWS Cost Explorer API, cost management and, 865–866
- AWS Cost Optimization Monitor, cost management and, 867
- AWS Database Migration Service, 176
- AWS database service, application mapping, 178
- AWS DataSync, 86
- AWS Direct Connect, 86, 128, 152–153, 158, 159
- AWS Directory Service, 509
- AWS DMS (Database Migration Service), 177, 233–235
  - database migration, 177
- AWS DX (Direct Connect), 774–776
- AWS EB CLI (Elastic Beanstalk CLI), 296
- AWS Elastic Beanstalk, 38, 290–291, 325.
  - See also* AWS EB CLI (Elastic Beanstalk CLI)
  - Amazon CloudFront and, 297–298
  - Amazon RDS, 298
  - Amazon S3 and, 297
  - applications, 289, 293
  - AWS CodePipeline and, 321
  - AWS Config, 298
  - components, 307
  - deployment, 307
  - ebextensions directory, 296–297
  - ElastiCache, 298–299
  - environment, 293–297

- environment tier, 293–294, 307
- health dashboard, 303–306
- IAM and, 299
- implementation, 291–292
- metrics, 304
- resources, 307
- source repository and, 292–293
- AWS Fargate, 475–476, 484, 486
- AWS Free Tier, 2
- AWS General Reference, 13
- AWS Import/Export, 146–147, 158
- AWS IoT (Internet of Things), 570
- AWS IoT Device Management
  - device shadow, 550
  - message broker, 549–550
  - MQTT (Message Queueing Telemetry Transport), 547
  - OTA (over-the-air) updates, 547
  - rules engine, 548–549
- AWS IoT (Internet of Things) Device SDK, 4
- AWS KMS (Key Management Service), 95, 260–262, 269–270, 760
- AWS Lambda, 38, 586–587
  - Amazon API Gateway integration, 631
  - Amazon CloudWatch and, metrics, 602–603
  - Amazon DynamoDB Streams, 706–707
  - AWS CodePipeline and, 321
  - AWS X-Ray, 603–604
  - environment variables, 599
  - functions
    - aliases, 600–601
    - concurrency, 597–598
    - concurrency limits, 598–599
    - context object, 595
    - creating, 589–590
    - descriptions, 596
    - DLQ (dead letter queue), 599
    - even objects, 595
    - execution methods, 590–592
    - execution permissions, 592
    - function handler, 594
    - function package, 593–594
    - invocation models, 590–592
    - invocation permissions, 593
    - InvocationType parameter, 591
    - invoking, 601–602
    - memory, 596
  - network configuration, 596–597
  - Nonstreaming Event Source (Push Model), 590–591
  - Streaming Event Source (Pull Model), 590, 592
  - tags, 596
  - timeouts, 596
  - versioning, 599–600
  - languages supported, 589
  - optimization and, 851
- AWS Managed Microsoft AD, 507–508
- AWS Management Console, 3–4, 12, 303, 590
  - access, 15–16
  - authentication, multi-factor authentication, 15–16
- AWS Elastic Beanstalk, health dashboard, 303–305
- AWS Lambda functions, 589
  - health monitoring, 303–305
  - IAM roles, 305–306
- AWS Mobile SDK, 4, 128
- AWS OpsWorks
  - Amazon EC2 auto scaling, 448
  - application deployment, 289
  - AWS CodePipeline and, 321
  - Chef compliance, 448
  - code repository, 448
- AWS OpsWorks Agent, lifecycle events, 461–462
- AWS OpsWorks for Chef Automate, 447
  - application deployment, 289
- AWS OpsWorks for Puppet Enterprise, 447
- AWS OpsWorks Stacks, 446, 484, 485
  - apps, 459–460
  - attribute files, 449
  - auto healing, 486
  - AWS CodePipeline and, 470
  - Chef 11, 464–465
  - Chef 12, 464–465
  - Chef Server, 450
  - Chef Solo, 447
  - components, 485
  - cookbooks, 456, 449–452
  - deployment, 470–471
  - instances, 456–459, 464, 467–469, 485–486
  - layers, 453–456

- lifecycle events, 461–463, 486
  - Permissions, 460–461, 486
  - recipes, 449–450, 461–462
  - resource management
    - Amazon EBS volumes, 463
      - elastic IP addresses, 464
    - service limits, 469
    - stacks, 452–453, 471
    - templates, custom, 456
  - AWS Region, 9–10, 23
    - API endpoints, 10–12
    - Availability Zones, 9, 10
    - planned regions, 9
    - samples, 13
    - selecting region, 14
  - AWS SAM (Serverless Application Model), 643–645
  - AWS SAM CLI, 645–647
  - AWS SCT (Schema Conversion Tool), 233–235
  - AWS SDK for Python, Boto, 4
  - AWS SDKs (software development kits), 3–4, 7–12, 128
    - AWS Lambda functions, 589
      - instances, 48
  - AWS Serverless Application Repository, 647
  - AWS Signature Version 4, 7
  - AWS Snow family, 86
  - AWS Snowball, 147–148, 158
  - AWS Snowball Edge, 148–150, 158
  - AWS Snowmobile, 150–151, 158
  - AWS SSO (Single Sign-On), 500–502
  - AWS Step Functions, 570
    - Choice Rules, 559–561
    - Choice state, 556–557
    - end state, 564
    - error handling, 564
    - input/output, 564–568
    - Parallel state, 561–564
    - state machines, 551–554
    - tasks, 554–556
    - use case, 568
  - AWS Storage Gateway, 86, 158
    - cached volume mode, 146
    - encryption, 266
    - file gateway, 146
    - migration and, 145–146
    - stored volume mode, 146
  - tape gateway, 146
  - volume gateway, 146
  - AWS STS (Security Token Service), 18
    - APIs, 503–505
    - credentials, 48
      - temporary, 502–503
  - AWS Systems Manager Parameter Store, 346
  - AWS Tag Editor, 836
  - AWS Trusted Advisor
    - cost management and, 864
    - performance monitoring, 869
  - AWS VPN, 158, 775
  - AWS X-Ray, 820
    - application request tracking, 821–823
    - AWS Lambda functions and, 603–604
    - monitoring, 798
    - use cases, 821
  - `AWS::CloudFormat::Init`, 400, 403–404
  - `AWS::CloudFormation::Designer`, 405
  - `AWS::CloudFormation::Init`, 435
  - `AWS::CloudFormation::Interface`, 404–405
  - `AWS::CloudFormation::Stack`, nesting, 418–419
- 
- B**
- Baidu Cloud Push, 537
  - bare-metal access, 38
  - binary scalar types, 670
  - BlazeMeter, 324
  - BLOB (binary large object) data, 88
  - block storage, 86, 91, 155, 782, 852
    - Amazon EBS, 87, 93
    - Amazon EC2 (Elastic Compute Cloud), instance store, 97–98
  - DAS (direct-attached storage), 91
  - ERP (enterprise resource planning systems), 91
  - NAS (network-attached storage), 91
  - SAN (storage area network), 91
  - block-level encryption, 265
  - blue/green deployment, 301, 310, 355
  - Boolean scalar types, 670
  - Bouncy Castle, 266

buckets (Amazon S3), 155–156  
 limitations, 99–100  
 namespace, universal, 100  
 operations, 103–105  
 regions, 103  
 versioning, 101–103  
 buffers, Amazon Kinesis Data Firehose, 544  
 build phase of release lifecycle, 283

## C

C# (.NET Core 1.0), AWS Lambda and, 589  
 C# (.NET Core 2.0), AWS Lambda and, 589  
 C++, AWS SDKs (AWS software development kits), 4  
 canary release, 630  
 CAP theorem (consistency, availability, partition tolerance), 115–116  
 CD (continuous delivery), 285  
*cfn-get-metadata* helper script, 425  
*cfn-hup* helper script, 425–426  
*cfn-init* helper script, 424  
*cfn-signal* helper script, 424–425  
 Chef, 446, 485  
 Chef 11, 464–467  
 Chef 12, 464–465  
 Chef Client, 447  
 Chef Server, 447, 450  
 Chef Solo, 447  
 Chef Zero, 447  
 CI (continuous integration), 285  
 CIA (confidentiality, integrity, availability) model, storage and, 91–92  
 CI/CD (continuous integration/continuous deployment), 285–286, 318  
 AWS CodeBuild, 286  
 AWS CodeCommit, 286  
 AWS CodeDeploy, 287  
 AWS CodePipeline, 286  
 CIDR (Classless Inter-Domain Routing) notation, 51  
 Classic Load Balancer, 287  
 client-side encryption, 121–123  
 cloud, database migration, 232–233  
   AWS DMS, 233–234  
   AWS SCT, 234–235  
 cloud services, calling, 5–9

CloudBees, 324  
 cloud-init directive, 47  
 CloudWatch, 50  
 CMK (customer master key), 96  
 code, configuration as, 446  
 cold data, 89  
 compliance, AWS KMS, 262  
 compute optimized instances, 39  
 condition functions, AWS CloudFormation  
   `FN::AND`, 398  
   `FN::IF`, 398  
   `FN::NOT`, 398  
   `FN::OR`, 398  
 configuration  
   answers to review questions, 903–905  
   Chef, 447–448  
   as code, 446  
   Puppet, 447–448  
 configuration management, 447–448  
 containers  
   deployments, 302–303  
   microservers, 522  
   optimization and, 849–850  
 continuous delivery, 319  
 continuous integration. *See* CI (continuous integration)  
 CI/CD (continuous integration/continuous deployment). *See* CI/CD  
 control planes, 497  
 cookbooks, 485  
   AWS OpsWorks Stacks  
     custom, 456  
     dependencies, 451–452  
     management, 450–451  
 CORS (cross-origin resource sharing), 631  
 cost management  
   Amazon CloudWatch, 867  
   AWS Budgets, 866  
   AWS Cost and Usage Report, 866–867  
   AWS Cost Explorer, 865  
   AWS Cost Explorer API, 865–866  
   AWS Cost Optimization Monitor, 867  
   AWS Trusted Advisor, 864  
   EC2 Right Sizing, 868  
 cost optimization, 834  
   AWS usage reduction, 836–838  
   tagging, 835–836  
 critical/regulated data, 90

cross-origin resource sharing. *See* CORS  
CRR (cross-region replication), 127–128  
custom builds, identity provider, 499

---

**D**

DAS (direct-attached storage), 91  
data, structure, 88  
data at rest, encryption, 119  
data dimensions, 87–88, 154  
data in transit, encryption, 119  
data lake architecture, 129–130  
data lakes, 86  
data migration, 145, 158  
    Amazon Kinesis Data Firehose, 151–152  
    AWS Direct Connect, 152–153  
    AWS Import/Export, 146–147  
    AWS Snowball, 147–148  
    AWS Snowball Edge, 148–150  
    AWS Snowmobile, 150–151  
    AWS Storage Gateway, 145–146  
    VPN connections, 153  
data plane, 497  
data protection, 118. *See also* encryption  
data temperature, 89  
data transfer, 858  
    Amazon CloudFront, 858  
    Amazon Kinesis, 86  
    Amazon S3 transfer acceleration, 858  
    AWS DataSync, 86  
    AWS Direct Connect, 86  
    AWS Snow family, 86  
    AWS Storage Gateway, 86  
    caching, 858–859  
    S3 Transfer Acceleration, 86  
data types  
    document, 671  
    scalar, 670  
data value, 89–90  
data warehousing  
    Amazon Redshift, 177, 220–226  
    architecture, 217–220  
    benefits, 217  
    data lake comparison, 219  
    data mart comparison, 219–220  
    database comparison, 218  
    databases, 176

database migration  
    Amazon RDS, 489  
    AWS DMS (Database Migration Service), 177  
    cloud, 232–235  
    heterogeneous, 233  
    homogenous, 233  
database services, mapping to database types, 176–177  
databases  
    Amazon Aurora, 176  
    Amazon EC2, 235–236  
    answers to review questions, 894–895  
    AWS OpsWorks Stacks, deployments, 471  
    compliance, IAM, 236–237  
    data warehouse, 176  
    DAX (Amazon DynamoDB Accelerator), 230  
    ElastiCache, 229–230  
    graph, 176, 230–232  
    IAM (AWS Identity and Access Management), 188–189  
    in-memory data stores, 176  
        caching, 226–227  
        in-memory key-value store, 228  
    ledger, 176  
    nonrelational, 176, 237  
        Amazon DynamoDB, 196–217  
        NoSQL, 195–196  
    relational, 237, 176, 178–180  
        Amazon Aurora, 190–192  
        Amazon CloudWatch, 189–190  
        Amazon RDS, 177, 180–188  
        Amazon RDS best practices, 192–194  
        IAM DB authentication, 188–189  
    security, IAM, 236–237  
        time-series, 176  
    DAX (Amazon DynamoDB Accelerator), 230  
dead letter queue). *See* DLQ (dead letter queue)  
decrypting passwords, Windows, 45–46  
deployment  
    all-at-once deployment, 300  
    answers to review questions, 897  
    applications, 288–290  
    AppSpec file, 299  
    AWS CloudFormation, AWS CodePipeline and, 430–432

AWS CodeDeploy, 299  
 AWS CodePipeline, CI/CD, 318  
 AWS Elastic Beanstalk, 290–291  
     implementation, 291, 292  
     container deployments, 302–303  
     continuous delivery, 319  
     ELB (Elastic Load Balancing)  
         Application Load Balancer, 287  
         environment variables, 284  
         highly available applications, 287–288  
         in-place deployment, 300  
         rolling, 301–302  
         scalable applications, 287–288  
         source repository, 292–293  
     deployment phase of release lifecycle, 283  
     dereference operators, 722–723  
     developer tools, AWS Cloud9, 66  
     DHCP (Dynamic Host Configuration  
         Protocol), 63  
     direct-attached storage. *See* DAS (direct-  
         attached storage)  
     DLQ (dead letter queue), 599  
     dm-crypt, 265  
     DNS (domain name servers), 63, 506  
     Docker containers, 295–296  
         Amazon ECR, 481  
         CLI tools, 481  
     document data types, 671  
     domain names, Amazon Route 53, 625–626  
     dual-stack mode, IPv6 addresses, 53

**E**

ebextensions directory, 296–297, 307  
 EC2 Right Sizing, cost management and,  
     868  
 Eclipse, 334  
 eCryptfs, 265  
 elastic IP addresses, 53  
 Elastic Load Balancing  
     AWS OpsWorks Stacks, layers, 454  
     monitoring metrics, 804  
 elastic network interfaces, 42  
 Elastic Volumes (Amazon EBS), 94–95  
 ElastiCache  
     access control, 747  
     application state, 739

AWS Elastic Beanstalk and,  
     298–299  
     backups, snapshots, 746–747  
     cache hits, 742–743  
     cache misses, 742–743  
     clusters, 741–742  
     data access patterns, 745  
     distributed cache, 740–741  
     endpoints, 742  
     in-memory data store, 177  
     in-memory key-value store, 739  
     lazy loading, 744  
     Memcached, 229–230, 739  
     Multi-AZ replication groups, 746  
     nodes, 741  
     Redis, 229–230, 739  
     replication groups, 742, 746  
     scaling, 745  
     snapshots, 746–747  
     TTL (time to live), 742  
     write-through, 744  
 ELB (Elastic Load Balancing), 287, 383  
 EncFs, 265  
 encryption  
     Amazon EBS, 95–96, 265–266, 274  
     Amazon EMR, 267–268  
     Amazon RDS, 266–267  
         AWS KMS, 187–188  
     Amazon S3, 156, 264–265, 271, 274  
     answers to review questions, 895–896  
     AWS CloudHSM, 262  
     AWS KMS (Key Management Services),  
         95, 260–262, 269–271  
     AWS managed, 263, 268–269  
     AWS Storage Gateway, 266  
     client-side, 122–123  
     customer managed, 263, 264–268  
     data at rest, 119  
     data in transit, 119  
     data protection, 760  
     dm-crypt, 265  
     eCryptfs, 265  
     EncFs, 265  
     file systems, accessing, 779  
     Loop-AES, 265  
     server-side, 271, 760  
     SSE (Server-Side Encryption), 119  
     TrueCrypt, 265

- endpoints
  - Amazon SNS, 535
  - API regional endpoints, 10–12
  - ElastiCache, 742
- envelope encryption, 270
- environment
  - AWS Elastic Beanstalk, 293–297
  - variables
    - AWS Lambda, 599
    - deployment, 284
- ERP (enterprise resource planning systems), 91
- exercises
  - account sign up, 26
  - Amazon API Gateway, running locally, 659
  - Amazon Cloud Directory setup, 514–515
  - Amazon CloudTrail, 827–828
  - Amazon CloudWatch
    - alarms, 826–827
    - dashboard, 828
  - Amazon Cognito setup, 516
  - Amazon DynamoDB table
    - backup, 791
    - creation, 250–251, 789
    - global tables, 790
    - removal, 255
    - restore, 792
    - scanning, 254–255
    - users, 252
  - Amazon DynamoDB user lookup, 253
  - Amazon EBS optimization, 877–878
  - Amazon EC2 (Elastic Compute Cloud)
    - instance connection, 73
    - key pairs, 69
    - private subnet, 75–76
    - as web server, 71–73
  - Amazon ECS
    - clusters, 488–489
    - containers, 488–489
  - Amazon EFS, volumes, 787–788
  - Amazon Kinesis Data Stream, 575–577
  - Amazon RDS
    - database migration, 489
    - database tier security, 242–243
    - endpoint value, 245–246
    - removal, 249–250
- Amazon S3
  - AWS Lambda function invocation, 615–616
  - buckets, uploading to, 788
  - event triggers, 616–617
- Amazon S3 buckets, 163
  - AWS Lambda functions and, 608
  - deleting, 167–169
  - emptying, 167–169
  - final output, JSON, 608–609
  - HTML file edits, 653–655
  - object load, 164–166
  - Swagger template, 652–653
  - unencrypted uploads, 275–276
  - verifying buckets, 609–610
- Amazon S3 versioning, 789
- Amazon SNS, SMS text message, 575
- Amazon SQS, 573–574
- Amazon VPC, 70
  - application version update, 311–312
  - auto scaling groups, 879–880
- AWS CLI
  - configuration, 28
  - CPU usage alarm, 876–877
  - installation, 28
- AWS Cloud9, 77–78
- AWS CloudFormation, 437–439
- AWS CodeBuild project creation, 375–376
- AWS CodeCommit repository, pull request, 374
- AWS CodeDeploy, application creation, 375
- AWS Config rule creation, 878–879
- AWS IAM role creation, 612–614
- AWS KMS
  - CMK (customer master key), 277–278
  - create/disable key, 276–277
- AWS Lambda
  - event source generation, 657
  - function creation, 614–615
  - function modification, 658–659
  - function preparation, 610–612
  - function testing, 617
  - invocation by Amazon S3, 615–616
  - local function definition, 656
  - running, 657
- AWS Managed Microsoft AD, 512–514

AWS OpsWorks Stacks  
 auto healing event notification, 490  
 environment launch, 488  
 AWS SAM template, 655–656  
 local API, 658  
 AWS Step Function, 578–581  
 batch processes, writing data, 253–254  
 blue/green solution deployment, 310  
 cleanup, 78–79  
 code samples, downloading, 28–29  
 cross-region replication, 791  
 deployment, 309  
 ElastiCache, Memcached cluster,  
 786, 787  
 environment, AWS Elastic Beanstalk,  
 310–311  
 IAM administrator group creation, 26–27  
 IAM administrator user creation, 26–27  
 IAM roles, API calls, 71  
 instances, private, requests, 76–77  
 launch configuration, 879–880  
 MariaDB database instance setup,  
 243–245  
 NAT, instances in private subnet, 74–75  
 profiles, 30–32  
 Python script, API calls, 29  
 regions, 29–30  
 scaling actions, 879–880  
 Simple AD setup, 510–512  
 SQL table creation, 246–248  
 SQL table queries, 248–249

---

**F**

FaaS (function-as-a-service), 587  
 federation, 496, 498–500, 509  
 file gateways, 146  
 file storage, 86, 91, 155, 853  
 Amazon EFS, 87  
 file-system encryption, 265  
 FIPS (Federal Information Processing  
 Standards), 260  
 Fn::Base64, 395  
 Fn::Cidr, 395  
 Fn::FindInMap, 395  
 Fn::GetAtt, 396

Fn::GetAZs, 396  
 Fn::Join, 396–397  
 Fn::Select, 397  
 Fn::Split, 397  
 Fn::Sub, 397–398  
 FPGA (Field Programmable Gate  
 Array), 39  
 frozen data, 89  
 function-as-a-service). *See* FaaS (function as  
 a service)

---

**G**

GCM (Google Cloud Messaging for  
 Android), 538–539  
 general purpose instances, 39  
 Ghost Inspector, 324  
 GitHub, AWS CodePipeline and, 324  
 Go, AWS SDKs (AWS software development  
 kits), 4  
 Go 1.x, AWS Lambda and, 589  
 GPU (Graphics Processing Unit), 39  
 graph databases, 176–177, 230–232

---

**H**

Hadoop, Amazon EMR, 266  
 HDD (hard disk drive)-backed  
 volumes, 93  
 SSD comparison, 94  
 helper scripts, AWS CloudFormation,  
 425–426  
 heterogeneous database migration, 233  
 highly available applications, deployment,  
 287–288  
 highly structured data, 88  
 HIPAA (Health Insurance Portability and  
 Accountability Act), 508  
 HMAC (hash message authentication  
 code), 266  
 homogenous database migration, 233  
 hot data, 89  
 HPE (Hewlett Packard Enterprise) Storm  
 Runner Load, 324  
 HSM (hardware security module), 260

---

IaC (infrastructure as code), 382, 434  
IAM (AWS Identity and Access Management), 5, 13, 14–15, 496  
access keys, 16  
Amazon DynamoDB, 732–736  
authentication, 19–20  
authorization, 19–20  
AWS Elastic Beanstalk and, 299  
condition element, 734  
database security, 236–237  
DB authentication, 188–189  
dev tools, 16  
groups, 16–17  
identities, 19–20  
as IdP (identity provider), 496  
Management Console, 15–16  
many-to-many relationships, users and groups, 16  
metadata, 48  
permissions, 20–21, 733–735  
policies, 20–24  
roles, 17–18, 20, 24  
users, 15, 24  
    Amazon EFS, 777  
    roles, 20  
identity, 497–498  
    identity consumer, 498  
    identity provider, 498–499, 505–506  
        Microsoft Active Directory, 500  
identity services, federation, 496  
IdP (identity provider), 496, 509  
    federation, 496  
images  
    AMI (Amazon Machine Language), 41–42  
    software images, 41–42  
IMDS (instance metadata service), 47–48  
immutable deployment, 301–302  
infrastructure  
    answers to review questions, 900–903  
    AWS CloudFormation and, 382  
    repeatable, 383  
    templates and, 384  
    versionable, 383

infrastructure as code. *See* IaC (infrastructure as code)  
in-memory data stores, 176–177, 226–228  
in-place deployment, 300, 354  
instance metadata service, *See* IMDS (instance metadata service)  
instance reservations  
    EC2 reservations, 841–842  
    pricing, 840–841  
    RDS reservations, 842  
instance store  
    Amazon EBS comparison, 143–144  
    Amazon EC2 (Elastic Compute Cloud), 97–99, 155  
    volumes, 98  
instances, 38  
    accelerated computing, 39  
    access, 43  
    Amazon EC2 (Elastic Compute Cloud), 45–46, 50  
    applications, running on, 44–50  
    AWS OpsWorks Stacks, 456–459, 464, 467–469  
    CloudWatch, 50  
    compute optimized, 39  
    families, 39  
    general purpose, 39  
    memory optimized, 39  
    metadata, 67  
    storage optimized, 39  
    stores, 40–41  
    types, 39  
IntelliJ, 334  
intrinsic functions, AWS CloudFormation  
    Fn::Base64, 395  
    Fn::Cidr, 395  
    Fn::FindInMap, 395  
    Fn::GetAtt, 396  
    Fn::GetAZs, 396  
    Fn::Join, 396–397  
    Fn::Select, 397  
    Fn::Split, 397  
    Fn::Sub, 397–398  
    Ref, 398  
IOPS (input/output operations per second), 773  
IP addresses, 42, 52–53

IPv6 addresses, 53  
 iSCSI (internet Small Computer System Interface), 145–146

**J**

Java, AWS SDKs (AWS software development kits), 4  
 Java 8, AWS Lambda and, 589  
 JavaScript, AWS SDKs (AWS software development kits), 4  
 JCE (Java Cryptography Extension), 261  
 Jenkins, 324  
 JSON (JavaScript Object Notation), identity, 497

**K**

key pairs, Amazon EC2 (Elastic Compute Cloud), 43  
 KMI (key management infrastructure), 263, 273

**L**

latency, 157  
 layers, AWS OpsWorks Stacks, 453–456  
 LDAP (Lightweight Directory Access Protocol), 506, 508  
 ledger databases, 176–177  
 lexicon, 11  
 lifecycle, release lifecycle, 282–284  
 lifecycle configuration, 134–135  
 lifecycle policies, 102  
 Lightweight Directory Access Protocol. *See* LDAP (Lightweight Directory Access Protocol)  
 load balancers, 287, 479  
 local secondary indexes, 201–202, 204–205, 682, 684  
 logs, web traffic, 624–625  
 Loop-AES, 265  
 loosely structured data, 88

**M**

Memcached, 229–230  
 memory optimized instances, 39  
 message infrastructure, refactor to microservices and, 522  
 message-oriented middleware. *See MoM*  
 (message-oriented middleware)  
 metadata, 67  
 MFA (multi-factor authentication), 15–16, 127, 636  
 microservices, 521  
 answers to review questions, 907–908  
 containers, 522  
 monolithic architectures and, 588  
 refactor to, 522  
 Microsoft Active Directory, 500  
 AD Connector (Active Directory Connector), 506–507  
 AD DS (Active Directory Domain Services), 506  
 AWS Managed Microsoft AD, 505–507, 507–508  
 AWS SSO (Single Sign-On) and, 501–502  
 migration, 145, 158  
 Amazon Kinesis Data Firehose, 151–152  
 to AWS CodeCommit, 341–343  
 AWS Direct Connect, 152–153  
 AWS Import/Export, 146–147  
 AWS Snowball, 147–148  
 AWS Snowball Edge, 148–150  
 AWS Snowmobile, 150–151  
 AWS Storage Gateway, 145–146  
 VPN connections, 153  
 MoM (message-oriented middleware), 523  
 monitor phase of release lifecycle, 284  
 monitoring, 303, 798  
 Amazon CloudWatch, 189–190, 798  
 alarms, 814–815, 815–817  
 cases, 800  
 dashboards, 817–818  
 log aggregation, 811, 812  
 log processing, 814  
 log searches, 812–814  
 metrics, 802–811

metrics repository, 801–811  
microservices, 521  
monitoring, 798  
answers to review questions, 912–914  
AWS CloudTrail, 798  
events, 818–820  
trails, 820  
AWS Management Console, 303–306  
AWS X-Ray, 798, 820–823  
metrics, 799–800  
monolithic architectures *versus*  
  microservices, 588  
MPNS (Microsoft Push Notification Service)  
  Windows Phone, 538  
multi-factor authentication. *See* MFA (multi-factor authentication)  
MVC (Model-View-Controller)  
  architecture, 623

---

**N**

namespaces, buckets (Amazon S3),  
  100–101  
NAS (network-attached storage), 91  
NAT (network address translation),  
  61–63, 128  
nesting  
  attributes, 722–723  
  AWS::CloudFormation::Stack,  
    418–419  
.NET, AWS SDKs (AWS software  
  development kits), 4  
network ACLs (network access control lists),  
  58–61, 68  
network address translation. *See* NAT  
  (network address translation)  
network-attached storage. *See* NAS  
  (network-attached storage)  
Network Load Balancer, 287  
networks  
  Amazon VPC, 51  
  connecting to others, 51–52  
  connection types, 52  
  DHCP (Dynamic Host Configuration  
    Protocol), 63  
  IP addresses, 52–53

NAT (network address translation),  
  61–63  
network ACLs (access control lists),  
  58–61  
network traffic monitoring, 64  
primary network interfaces, 42  
route tables, 55–56  
security groups, 56–58  
subnets, 54–55  
  virtual, elastic network interfaces, 42  
Node.js 4.3, AWS Lambda and, 589  
Node.js 6.10, AWS Lambda and, 589  
Node.js 8.10, AWS Lambda and, 589  
nonrelational databases, 176, 237  
  Amazon Document DB, 177  
  Amazon DynamoDB, 196–217  
  NoSQL, 177, 195–196  
Nouvolà, 324  
null scalar types, 670  
number scalar types, 670

---

**O**

object storage, 86, 91, 155, 782  
  Amazon S3, 87, 99–105  
  Amazon S3 Glacier, 87  
  DEEP\_ARCHIVE, 852  
  GLACIER, 852  
  INTELLIGENT\_TIERING, 852  
  ONEZONE\_IA, 852  
  STANDARD\_IA, 852  
objects, 99, 108–109, 761–765  
OIDC (OpenID Connect), 498, 500  
OP (OpenID provider), 500  
OpenSSL, 266  
OpsWorks Stacks, 325  
optimistic locking, 713–714  
optimization  
  answers to review questions, 914–916  
  Auto Scaling, 845–846  
    accessing, 848–849  
    Amazon Aurora, 848  
    Amazon EC2 Auto Scaling, 846–847  
    AWS Auto Scaling, 847–848  
    DynamoDB, 848  
  AWS Lambda and, 851

containers, 849–850  
cost optimization, 834–838  
costs, 864–868  
data transfer, 858–859  
instance reservations, 841–842  
RDBMS (relational database management system), 859–864  
right sizing, 838–840  
serverless approaches, 850–851  
Spot Instances, 843–845  
storage, 851–857

---

**P**

partitions  
Amazon DynamoDB, 197, 711  
distribution, 711–713  
partition key, 665–668, 712–713  
primary key, 199–200  
sort key, 713  
ARN, 23  
CAP theorem (consistency, availability, partition tolerance), 115–116  
nonrelational databases, 197  
partition key, Amazon Kinesis Data Streams, 541  
passwords, decrypting, Windows, 45–46  
PCI DSS (Payment Card Industry Data Security Standard), 508  
performance monitoring  
Amazon CloudWatch, 868  
AWS Trusted Advisor, 869  
permissions  
Amazon DynamoDB, IAM policy and, 732–735  
AWS CloudFormation, 385–386  
StackSets, 428–429  
AWS OpsWorks Stacks, 460–461  
wildcards, 22  
persistent storage, 40–41  
PHP, AWS SDKs (AWS software development kits), 4  
policies, IAM, 20–24  
POSIX (Portable Operating System Interface), 773  
presigned URLs, 118, 125

primary network interfaces, 42  
private IP addresses, 53  
private subnets, 55, 67  
privileges, IAM policies, 21  
programmatic access, 16  
public IP addresses, 53  
public subnets, 55, 67  
push notifications, Amazon SNS mobile, 537–539  
Python 2.7, AWS Lambda and, 589  
Python 3.6, AWS Lambda and, 589

---

**Q**

query string authentication, 125–126

---

**R**

RDBMS (relational database management system), optimization and, 859–860  
fewer tables, 860  
indexes, 862–863  
NoSQL and, 860  
projections, 863  
query frequency, 863  
related data, 860  
scan operations, 863–864  
sort keys, version control, 862  
workload distribution, 861–862  
RDP (Remote Desktop Protocol), 238  
Amazon EBS, 97  
Amazon EC2 (Elastic Compute Cloud) instances, 43  
data plane, 497  
read consistency, 206–207  
reads per second. See RPS (reads per second)  
Redis, 229–230  
Ref, 398  
refactor to microservices, 522  
regions, ARN, 23  
relational databases, 178–179, 237. *See also* RDBMS (relational database management system)

ACID  
atomicity, 179  
consistency, 180  
durability, 180  
isolation, 180  
Amazon Aurora, 190–192  
Amazon CloudWatch, 189–190  
Amazon RDS, 177, 180–189, 238  
    Amazon Aurora, 190–192  
    Amazon CloudWatch, 189–190  
    best practices, 192–194  
columns, 178–179  
data integrity, 179  
fields, 179  
foreign keys, 179  
IAM DB authentication, 188–189  
managed, 176, 180  
nonrelational, Amazon Document DB,  
    177  
objects, 178  
primary keys, 179  
rows, 179  
SQL (Structured Query Language), 179  
transactions, 179  
    unmanaged, 180  
release lifecycle, 282–284  
Remote Desktop Protocol. *See* RDP (Remote  
    Desktop Protocol)  
repeatable infrastructure, 383  
repositories, deployment and, 292–293  
Representational State Transfer. *See* REST  
reproducible data, 90  
resource management, security, shared  
    responsibility model, 64–65, 155  
resources, 24  
    ARN, 24  
    usage reduction, 836–838  
REST (Representational State Transfer), 623  
RESTful APIs, 631  
right sizing, 838–840  
roles, IAM, 17–18  
rolling deployment, 301–302  
route tables, 55  
RPS (reads per second), 88  
Ruby, AWS SDKs (AWS software  
    development kits), 4  
Runscope, 324

---

## S

S3 Transfer Acceleration, 86  
S3DistCp, 272  
same-origin policy, 631  
SAML (Security Assertion Markup  
    Language), 498, 499  
SAN (storage area network), 91  
scalable applications, deployment,  
    287–288  
scalar data types, 670  
SDLC (software development lifecycle), 282  
    AWS Cloud, 284–285  
    environment variables, 284  
    release lifecycle, 282–284  
secondary indexes, 201, 665, 683  
    alternate key, 684  
    base table, 683  
    configuration, 685  
    global secondary indexes, 202–205,  
        682, 684  
    local secondary indexes, 201–202,  
        204–205, 682, 684  
security, shared responsibility model, 64–65,  
    155  
Security Assertion Markup Language. *See*  
    SAML (Security Assertion Markup  
    Language)  
security groups, 56–58, 68  
serverless applications, 622  
    Amazon Aurora Serverless, 642–643  
    Amazon S3, 129  
    answers to review questions, 909–910  
    AWS SAM (Serverless Application  
    Model), 643–645  
    AWS SAM CLI, 645–647  
    AWS Serverless Application Repository,  
        647  
    user cases, 647  
serverless compute, 586. *See also* AWS  
    Lambda  
    answers to review questions,  
        908–909  
serverless stack *versus* three-tier  
    architecture, 640–642  
server-side encryption, AWS KMS, 271  
service token acts, 406

- services
  - ARN, 23
  - managed, 65–66
  - unmanaged, 65–66
- shared responsibility security model, 64–65, 155
  - storage and, 91
- Simple AD (Simple Active Directory), 507
- snapshots, Amazon EBS, 95
- SOA (service-oriented architecture), 476–477, 798
- SOAP, 128–129
- software
  - customization, user data and, 46–47
  - images, 41–42
- Solano CI, 324
- source phase of release lifecycle, 283
- source repository, deployment and, 292–293
- Spot Instances, 844–845
- SSD (solid-state drive)-backed volumes, 93
  - HDD comparison, 94
- SSDs (solid-state drives), 665
- SSE (Server-Side Encryption), 119
  - SSE-C (customer-provided keys), 120
  - SSE-KMS (AWS KMS), 120–121
  - SSE-S3 (Amazon S3 managed keys), 120, 760
- SSH (Secure Shell), data plane, 497
- SSL (Secure Sockets Layer), ELB (Elastic Load Balancing), 287
- SSML (Speech Synthesis Markup Language), 5
- stacks, AWS OpsWorks Stacks, 452–453
- StackSets (AWS CloudFormation), 427–429
- state machines, AWS Step Functions, 551–554
  - stateless application pattern, 129, 664
    - answers to review questions, 910–912
- static websites, 126
  - Amazon S3, 156
- stop deployments, 355
- storage
  - Amazon EBS, 40–41, 94–97, 855–857
  - Amazon EFS, 136–142
  - Amazon S3, 853–855
    - access control, 123–125
    - authentication, 129
  - AWS CLI, 128
- AWS explorers, 128
- AWS SDKs, 128
- buckets, 99–105
- classes, 109–114, 156
- consistency model, 114–118
- CORS (cross-origin resource sharing), 107–108
- CRR (cross-region replication), 127–128
- data lake architecture, 129–130
- encryption, 118–123
- MFA Delete, 127
- objects, 105–108
- performance, 130–134
- presigned URLs, 118
- pricing, 134
- query string authentication, 125–126
- requests, 129
- serverless applications, 129
- stateless applications, 129
- static websites, 126
- VPC (virtual private cloud)
  - endpoints, 128
- answers to review questions, 890–893
- block, 86, 91, 155, 782
- block storage, 852
- CIA (confidentiality, integrity, availability) model, 91–92
- comparisons, 142–144
- DAS (direct-attached storage), 91
- data dimensions, 87–88
- data lakes, 86
- data temperature, 89
- data transfer, 86
- data value, 89–90
- data volume and, 157
- ERP (enterprise resource planning systems), 91
- file, 86, 91, 155
- file storage, 853
- highly structured data, 88
- item size and, 157
- latency, 157
- loosely structured data, 88
- mental model, 87
- NAS (network-attached storage), 91
- object, 86, 91, 155, 782, 852
- optimization, 851–857

---

persistent, 40–41  
products, 142–143  
SAN (storage area network), 91  
shared responsibility model, 91  
temporary, 41  
unstructured data, 88  
storage area network (SAN), 91  
storage optimized instances, 39  
string scalar types, 670  
subnets, 55, 67  
Sun Java JCE, 261  
system-level encryption, 265

---

## T

tape gateways, 146  
TeamCity, 324  
templates  
  AWS CloudFormation, 386–394  
  AWS CloudFormation CLI,  
    transforms, 423  
  AWS OpsWorks Stacks, custom, 456  
infrastructure, 384  
  transforms, AWS CloudFormation  
    CLI, 423  
temporary storage, 41  
test phase of release lifecycle, 283  
three-tier architecture, 282  
  *versus* serverless stack, 640–642  
time series databases, Amazon  
  Timestream, 177  
time-series databases, 176  
TLS (Transport Layer Security),  
  ELB, 287  
transient data, 90  
troubleshooting, 303, 798  
  Amazon EBS, 97  
  answers to review questions, 912–914  
TrueCrypt, 265  
trust policies, IAM roles, 18

---

## U

unstructured data, 88  
user data, 46–47, 67  
users, IAM, 15

---

## V

variables, environment variables  
  AWS Lambda, 599  
  deployment, 284  
variety, data, 88  
velocity, data, 88  
versionable infrastructure, 383  
versioning, buckets (Amazon S3),  
  100–101  
virtual networks, interfaces, elastic network  
  interfaces, 42  
Visual Policy Editor, 24  
Visual Studio, 334  
volume, data, 88  
volume gateways, 146  
VPC (virtual private cloud)  
  Amazon EFS, 773–775  
  endpoints, 128  
VPNs (virtual private networks), 128, 159  
  data migration, 153

---

## W

warm data, 89  
web servers  
  Amazon S3, 622–623  
  traffic logs, 624–625  
webpages, custom, Amazon EC2 (Elastic  
  Compute Cloud) and, 49–50  
websites, static, 126  
wildcards, permissions, 22  
Windows, passwords, decrypting,  
  45–46  
WNS (Windows Push Notification  
  Services), 538  
WORM (Write Once Read Many), 111  
WPS (writes per second), 88

---

## X–Y–Z

x-amzn-requestid header, 7  
Xebia Labs, 325  
  
zeroization, 268

# Comprehensive Online Learning Environment

Register to gain one year of FREE access to the online interactive learning environment and test bank to help you study for your AWS Certified Developer - Associate exam— included with your purchase of this book!

---

The online test bank includes the following:

- **Assessment Test** to help you focus your study to specific objectives
- **Chapter Tests** to reinforce what you've learned
- **Practice Exams** to test your knowledge of the material
- **Digital Flashcards** to reinforce your learning and provide last-minute test prep before the exam
- **Searchable Glossary** to define the key terms you'll need to know for the exam

Go to <http://www.wiley.com/go/sybextestprep> to register and gain access to this comprehensive study tool package.

## Register and Access the Online Test Bank

To register your book and get access to the online test bank, follow these steps:

1. Go to [bit.ly/SybexTest](http://bit.ly/SybexTest).
2. Select your book from the list.
3. Complete the required registration information, including answering the security verification to prove book ownership. You will be emailed a PIN code.
4. Follow the directions in the email or go to <https://www.wiley.com/go/sybextestprep>.
5. Enter the PIN code you received and click the “Activate PIN” button.
6. On the Create an Account or Login page, enter your username and password, and click Login. A “Thank you for activating your PIN!” message will appear. If you don't have an account already, create a new account.
7. Click the “Go to My Account” button to add your new book to the My Products page.