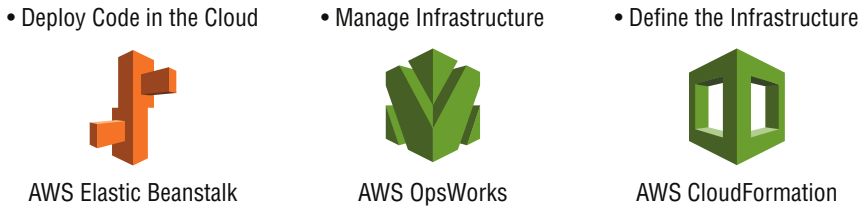
AWS®

Certified Developer

Official Study Guide

Associate (DVA-C01) Exam

FIGURE 6.5 Deployment and maintenance services



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

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

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

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

Automatically Adjust Capacity

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

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

Auto Scaling Groups

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

You can use the Auto Scaling group to scale the number of instances automatically based on criteria that you specify or maintain a fixed number of instances even if an instance becomes unhealthy. This automatic scaling and maintaining the number of instances in an Auto Scaling group make up the core functionality of the EC2 Auto Scaling service.

An Auto Scaling group launches enough Amazon EC2 instances to meet its desired capacity. The Auto Scaling group maintains this number of instances by performing periodic health checks on the instances in the group. If an instance becomes unhealthy, the group terminates the unhealthy instance and launches another instance to replace it.

You can use scaling policies to increase or decrease the number of instances in your group dynamically to meet changing conditions. When the scaling policy is in effect, the Auto Scaling group adjusts the desired capacity of the group and launches or terminates the instances as needed. You can also manually scale or scale on a schedule.

AWS Elastic Beanstalk

AWS Elastic Beanstalk is an AWS service that you can use to deploy applications, services, and architecture. It provides provisioned scalability, load balancing, and high availability. It uses common languages, including Java, .NET, PHP, Node.js, Python, Ruby, Go, and Docker, on common-type web servers, such as Apache, NGINX, Passenger, and IIS.

Elastic Beanstalk charges only for the resources you use to run your application.

Elastic Beanstalk is a solution that enables the automated deployments and management of applications on the AWS Cloud. Elastic Beanstalk can launch AWS resources automatically with Amazon Route 53, AWS Auto Scaling, Elastic Load Balancing, Amazon EC2, and Amazon Relational Database Service (Amazon RDS) instances, and it allows you to customize additional AWS resources.

Deploy applications without worrying about managing the underlying technologies, including the following:

Components

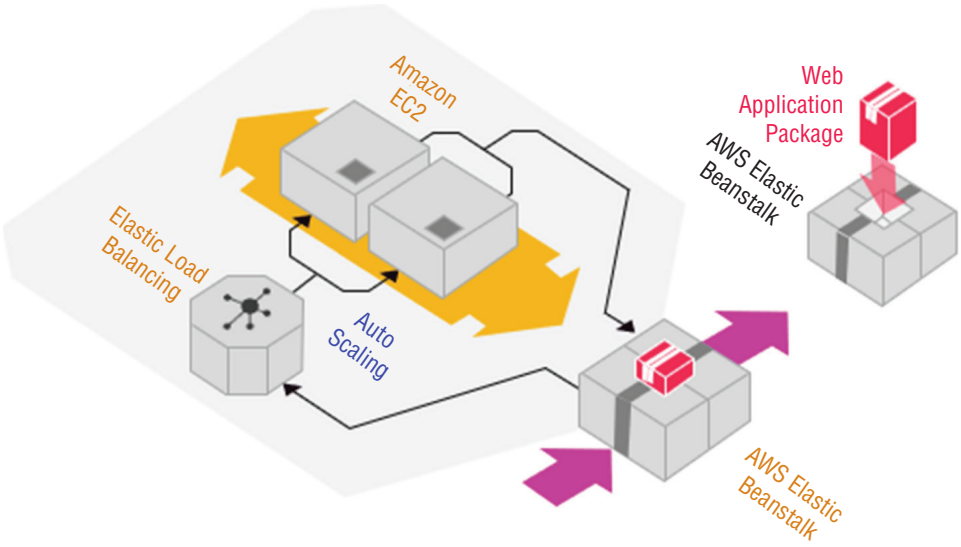
- Environments
- Application versions
- Environment configurations

Permission Model

- Service role
- Instance profile

Figure 6.6 displays the Elastic Beanstalk underlying technologies.

FIGURE 6.6 AWS Elastic Beanstalk underlying technologies



Elastic Beanstalk supports customization and N-tier architectures. It mitigates common manual configurations required in a traditional infrastructure deployment model. With Elastic Beanstalk, you can also create repeatable environments and reduce redundancy, thus rapidly updating environments and facilitating service-managed application stacks. You can deploy multiple environments in minutes and use various automated deployment strategies.

AWS Elastic Beanstalk allows you to focus on building your application.

Implementation Responsibilities

AWS and our customers share responsibility for achieving a high level of software component security and compliance. This shared model reduces your operational burden. The service you select determines the level of your responsibility. For example, Elastic Beanstalk helps you perform your side of the shared responsibility model by providing a managed updates feature. This feature automatically applies patch and minor updates for an Elastic Beanstalk supported platform version.

Developer Teams

Using AWS Elastic Beanstalk, you build full-stack environments for web and worker tiers. The service provides a preconfigured infrastructure.

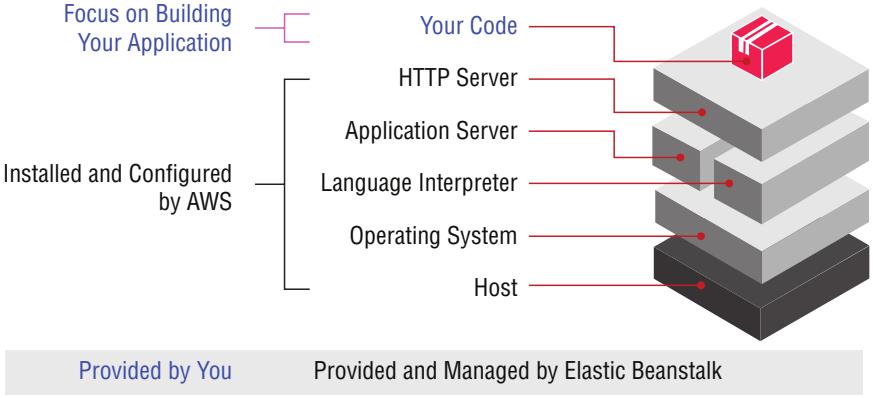
- Single-instance (development, low cost)
- Load balanced, AWS Auto Scaling (production)

Elastic Beanstalk Responsibilities

Elastic Beanstalk provisions the necessary infrastructure resources, such as the load balancer, Auto Scaling group, security groups, and database (optional). It also provides a unique domain name for your application (for example, `yourapp.elasticbeanstalk.com`).

Figure 6.7 displays Elastic Beanstalk responsibilities.

FIGURE 6.7 AWS Elastic Beanstalk responsibilities



Working with Your Source Repository

Developer teams generally begin their SDLC processes by managing their source code in a source repository. Uploading and managing the multiple changes on application source code is a repeated process. With Elastic Beanstalk, you can create an application, upload a version of the application as a source bundle, and provide pertinent information about the application.

The first step is to integrate Elastic Beanstalk with your source code to create your source bundle. As your source repository, you can install Git for your applications or use an existing repository and map your current branch from a local repository in Git to retrieve the source code.

Alternatively, you can use AWS CodeCommit as a source control system to retrieve source code. By using Elastic Beanstalk with the AWS CodeCommit repository, you extract from a current branch on CodeCommit.

To deploy a new application or application version, Elastic Beanstalk works with source bundles or packaged code. Prepare the code package with all of the necessary code dependencies and components.

Elastic Beanstalk can either retrieve the source bundle from a source repository or download the bundle from an Amazon Simple Storage Service (Amazon S3) bucket. You can use the IAM role to grant Elastic Beanstalk access to all services. The service accesses the source bundle from the location you designate, extracts the components from the bundle, deploys new application versions by launching the code, creates and

configures the infrastructure, and allocates the platform on Amazon EC2 instances to run the code.

The application runs on the resources and instances that the service generates. Your configuration for these resources and your application will become your environment settings, supporting the entire configuration of your deployment. Each deployment has an auto-incremented deployment identity (ID), so you are able to manage your multiple running deployments. Think of these as multiple running code releases in the AWS Cloud.

You can also work with different hosting services, such as GitHub or Bitbucket, with your code source.

Concepts

AWS Elastic Beanstalk enables you to manage all the resources that run your application as environments. This section describes some key Elastic Beanstalk concepts.

Application

Elastic Beanstalk focuses on managing your applications as environments and all of the resources to run them. Each application that launches in the service is a logical collection of environment variables and components, application versions, and environment configurations.

Application Versions

Application versions are iterations of the application's deployable code. Application versions in Elastic Beanstalk point to an Amazon S3 object with the code source package. An application can have many versions, with each version being unique. You can deploy and access any application version at any time. For example, you may want to deploy different versions for different types of tests.

Environment

Each Elastic Beanstalk environment is a separate version of the application, and that version's AWS Cloud components deploy onto AWS resources to support that version. Each environment runs one application version at a time, but you can run multiple environments, with the same application on each, along with its own customizations and resources.

Environment Tier

To launch an environment, you must first choose an environment tier. Elastic Beanstalk provisions the required resources to support both the infrastructure and types of requests

the application will support. The environment can launch and access other AWS resources. For example, it may pull tasks from Amazon Simple Queue Service (Amazon SQS) queues or store temporary configuration files in Amazon S3 buckets (according to your customizations). Each environment will then have an environment configuration—a collection of settings and parameters based on your customizations that define associated resources and how the environment will work.

Environment Configuration

You can change your environment to create, modify, delete, or deploy resources and change the settings for each. Your environment configuration saves to a configuration template exclusive to each environment and is accessible by either the Elastic Beanstalk application programming interface (API) calls or the service's command line interface (EB CLI).

In Elastic Beanstalk, you can run either a web server environment or a worker environment. Figure 6.8 displays an example of a web server environment running in Elastic Beanstalk with Amazon Route 53 as the domain name service (DNS) and ELB to route traffic to the web server instances.

FIGURE 6.8 Application running on AWS Elastic Beanstalk

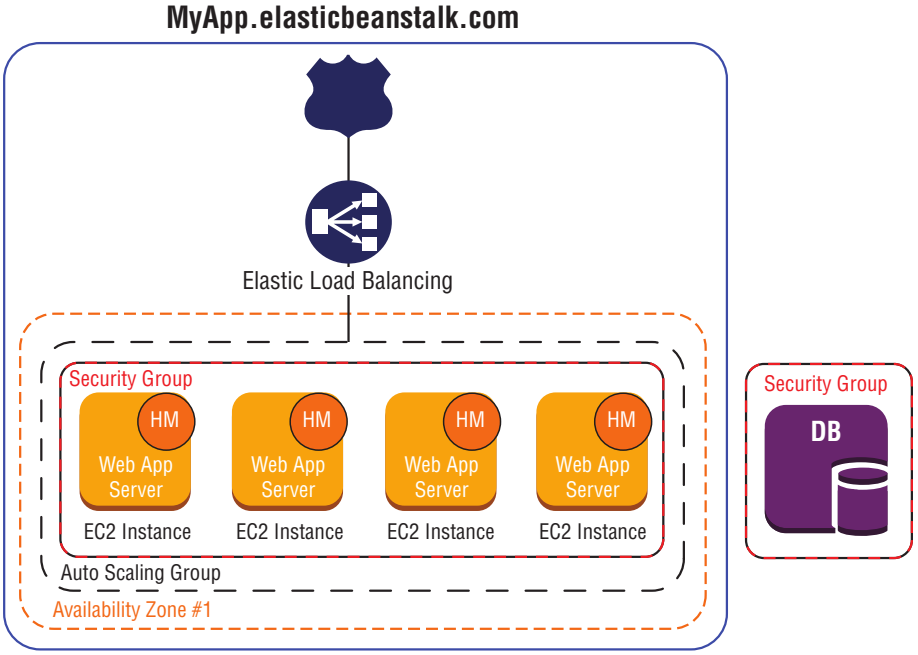
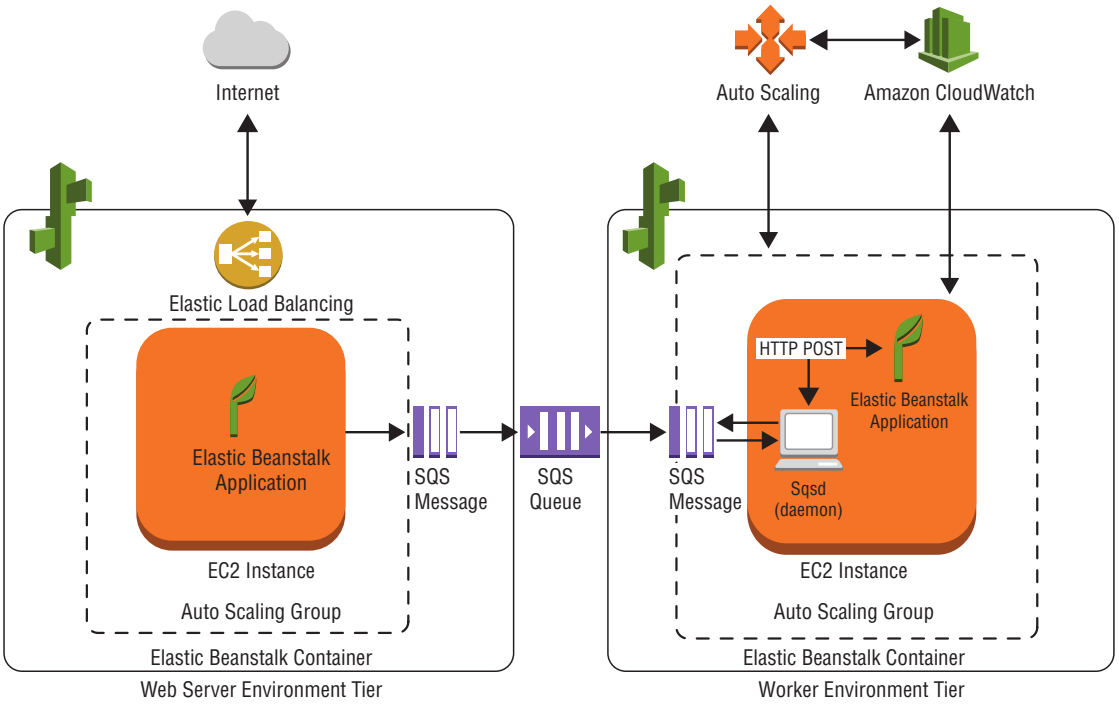


Figure 6.9 shows a worker environment architecture, where AWS resources create configurations, such as Auto Scaling groups, Amazon EC2 instances, and an IAM role, to manage resources for your worker applications.

FIGURE 6.9 Worker tier on AWS Elastic Beanstalk



For the worker environment tier, Elastic Beanstalk creates and provisions additional resources and files to support the tier. This includes services like Amazon SQS queues operating between worker applications, AWS Auto Scaling groups, security groups, and EC2 instances.

The worker environment infrastructure uses all of your customization and provision resources to determine the types of requests it receives.

Docker Containers

You can also use Docker containers with Elastic Beanstalk to run your applications from a container. Install Docker, choose the software you require, and select the Docker images you want to launch. Define your runtime environment, platform, programming language, and application dependencies and tools. Docker containers are self-contained and include configurations and software that you specify for your application to run. Each Docker container restarts automatically if another container crashes. When you choose to deploy your applications with Docker containers, your infrastructure is provisioned with capacity provisioning, load balancing, scaling, and health monitoring, much like a noncontainer environment. You can continue to manage your application and the AWS resources you use.

Docker requires platform configurations that enable you to launch single or multicontainer deployments. A single container deployment launches a single Docker image, and your application uses a single container configuration for a single Amazon EC2 instance.

rolling updates, set a deployment policy either in the AWS Management Console or in the command line (`DeploymentPolicy`) and choose this strategy along with specific options. You can select *Rolling* or *Rolling with additional batch*. By using *Rolling with additional batch*, you can launch a new batch of instances before you begin to take instances out of service for your rolling updates. This option provides an available batch for rollback from a failed update. After the deployment is successfully executed, Elastic Beanstalk terminates the instances from the additional batch. This is helpful for a critical application that must continue running with less downtime than the standard rolling update.

Blue/Green Deployment

When high availability is critical for applications, you may want to choose a *blue/green deployment*, where your newer environment will be separate from your existing environment. The running production environment is considered the *blue environment*, and the newer environment with your update is considered the *green environment*. When your changes are ready and have gone through all tests in your green environment, you can swap the CNAMEs of the environments to redirect traffic to the newer running environment. This strategy provides an instantaneous update with typically zero downtime.

When you deploy to AWS Lambda functions, blue/green deployments publish new versions of each function. Traffic shifting then routes requests to the new functioning versions according to the deployment configuration you define.

If your infrastructure contains Amazon RDS database instances, the data does not automatically transfer to the new environment. Without performing backups, you will experience data loss when you use the blue/green strategy. If you have Amazon RDS instances in your infrastructure, implement a different deployment strategy or a series of steps to create snapshot backups outside of Elastic Beanstalk before you execute this type of deployment.

Immutable Deployment

An *immutable deployment* is best when an environment requires a total replacement of instances, rather than updates to an existing part of an infrastructure. This approach implements a safety feature for updates and rollbacks. Elastic Beanstalk creates a temporary Auto Scaling group behind your environment's load balancer to contain the new instances with the updates you apply. If the update fails, the rollback process terminates the Auto Scaling group. Immutable instances implement a number of health checks. If all instances pass these checks, Elastic Beanstalk transfers the new configurations to the original Auto Scaling group, providing an additional check before you apply your changes to other instances. Enhanced health reports evaluate instance health in the update. After the updates are made, Elastic Beanstalk deletes the temporary Auto Scaling group of the older instances.

During this type of deployment, your capacity doubles for a short duration between the updates and terminations of instances. Before you use this strategy, verify that your instances have a low on-demand limit and enough capacity to support immutable updates.

Deploy to Amazon EC2 Auto Scaling Groups

When you deploy to Amazon EC2 Auto Scaling groups, AWS CodeDeploy will automatically run the latest successful deployment on any new instances created when the group scales out. If the deployment fails on an instance, it updates to maintain the count of healthy instances. For this reason, AWS does not recommend that you associate the same Auto Scaling group with multiple deployment groups (for example, you want to deploy multiple applications to the same Auto Scaling group). If both deployment groups perform a deployment at roughly the same time and the first deployment fails on the new instance, it terminates by AWS CodeDeploy. The second deployment, unaware that the instance terminated, will not fail until the deployment times out (*the default timeout value is 1 hour*). Instead, you should combine your application deployments into one or consider the use of multiple Auto Scaling groups with smaller instance types.

Deployment Configuration

You use *deployment configurations* to drive how quickly Amazon EC2 on-premises instances update by AWS CodeDeploy. You can configure deployments to deploy to all instances in a deployment group at once or subgroups of instances at a time, or you can create an entire new group of instances (*blue/green deployment*). A deployment configuration also specifies the fault tolerance of deployments, so you can roll back changes if a specified number or percentage of instances or functions in your deployment group fail to complete their deployments and signal success back to AWS CodeDeploy.

Amazon EC2 On-Premises Deployment Configurations

When you deploy to Amazon EC2 on-premises instances, you can configure either in-place or blue/green deployments.

In-Place deployments These deployments recycle currently running instances and deploy revisions on existing instances.

Blue/Green deployments These deployments replace currently running instances with sets of newly created instances.

In both scenarios, you can specify wait times between groups of deployed instances (batches). Additionally, if you register the deployment group with an *elastic load balancer*, newly deployed instances also register with the load balancer and are subject to its health checks.

The deployment configuration specifies success criteria for deployments, such as the minimum number of healthy instances that must pass health checks during the deployment process. This is done to maintain required availability during application updates. AWS CodeDeploy provides three built-in deployment configurations.

CodeDeployDefault.AllAtOnce

For in-place deployments, AWS CodeDeploy will attempt to deploy to all instances in the deployment group at the same time. The success criteria for this deployment configuration

Know how to use the credential helper to connect to repositories. It is possible to connect to AWS CodeCommit repositories using IAM credentials. The AWS CodeCommit credential helper translates an IAM access key and secret access key into valid Git credentials. This requires the AWS CLI and a Git configuration file that specifies the credential helper.

Understand the different strategies for migrating to AWS CodeCommit. You can migrate an existing Git repository by cloning to your local workstation and adding a new remote, pointing to the AWS CodeCommit repository you create. You can push the repository contents to the new remote. You can migrate unversioned content in a similar manner; however, you must create a new local Git repository (instead of cloning an existing one). Large repositories can be migrated incrementally because large pushes may fail because of network issues.

Know the basics of AWS CodeBuild. AWS CodeBuild allows you to perform long-running build tasks repeatedly and reliably without having to manage the underlying infrastructure. You are responsible only for specifying the build environment settings and the actual tasks to perform.

Know the basics of AWS CodeDeploy. AWS CodeDeploy standardizes and automates deployments to Amazon EC2 instances, on-premises servers, and AWS Lambda functions. Deployments can include application/static files, configuration tasks, or arbitrary scripts to execute. For Amazon EC2 on-premises deployments, a lightweight agent is required.

Understand how AWS CodeDeploy works with Amazon EC2 Auto Scaling groups. When you deploy to Amazon EC2 Auto Scaling groups, AWS CodeDeploy will automatically run the last successful deployment on any new instances that you add to the group. If the deployment fails on the instance, it will be terminated and replaced (to maintain the desired count of healthy instances). If two deployment groups for separate AWS CodeDeploy applications specify the same Auto Scaling group, issues can occur. If both applications deploy at roughly the same time and one fails, the instance will be terminated before success/failure can be reported for the second application deployment. This will result in AWS CodeDeploy waiting until the timeout period expires before taking any further action.

Resources to Review

What is DevOps?

```
https://aws.amazon.com/devops/what-is-devops/
```

AWS DevOps Blog:

```
https://aws.amazon.com/blogs/devops/
```

Introduction to DevOps on AWS:

```
https://d1.awsstatic.com/whitepapers/AWS_DevOps.pdf
```

Practicing Continuous Integration and Continuous Delivery on AWS:

```
https://d1.awsstatic.com/whitepapers/DevOps/practicing-continuous-
integration-continuous-delivery-on-AWS.pdf
```

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

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

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

Wait Conditions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Stack Create, Update, and Delete Statuses

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

CREATE_COMPLETE

The stack has created successfully.

CREATE_IN_PROGRESS

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

CREATE_FAILED

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

DELETE_COMPLETE

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

DELETE_IN_PROGRESS

The stack is currently deleting.

DELETE_FAILED

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

ROLLBACK_COMPLETE

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

ROLLBACK_IN_PROGRESS

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

ROLLBACK_FAILED

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

UPDATE_COMPLETE

The stack has updated successfully.

UPDATE_IN_PROGRESS

The stack is currently performing an update.

UPDATE_COMPLETE_CLEANUP_IN_PROGRESS

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

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

UPDATE_ROLLBACK_COMPLETE

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

UPDATE_ROLLBACK_IN_PROGRESS

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

UPDATE_ROLLBACK_COMPLETE_CLEANUP_IN_PROGRESS

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

UPDATE_ROLLBACK_FAILED

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

Stack Updates

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

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

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

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

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

TABLE 8.1 AWS CloudFormation Update Types

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

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

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

Update Policies

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

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

TABLE 8.2 AWS Auto Scaling Update Types in AWS CloudFormation

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

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

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

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

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

SuspendProcesses Attribute

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

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

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

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

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

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

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

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

Deletion Policies

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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:

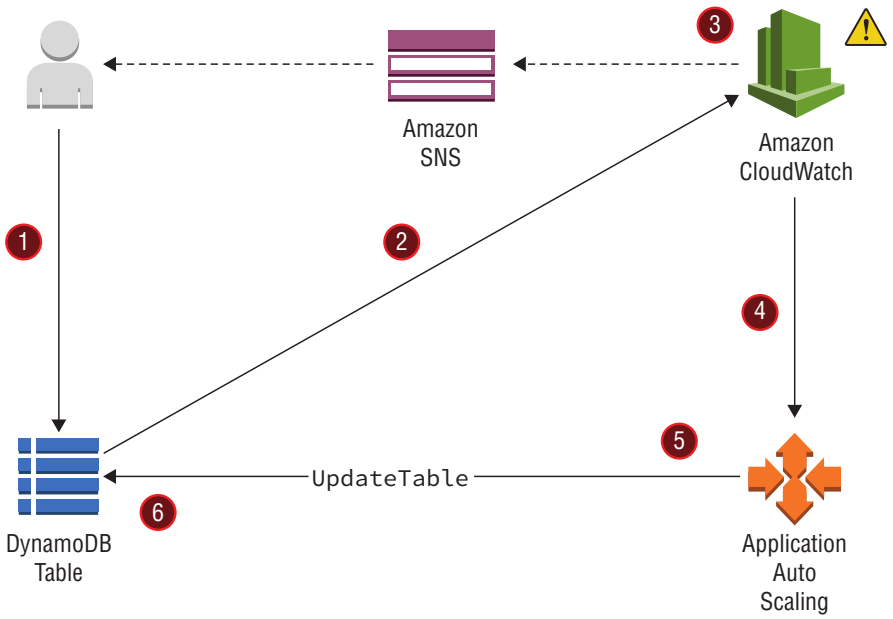
- Send requests to the table until automatic scaling scales down to the minimum capacity.
- Change the policy and reduce the maximum provisioned capacity to the same size as the minimum provisioned capacity.

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.

TABLE 15.2 Amazon EC2 Metrics

Namespace	`AWS/EC2`
Dimensions	▪ `InstanceId`: identifier of a particular Amazon EC2 instance ▪ `InstanceType`: type of Amazon EC2 instance, such as `t2.micro`, `m4.large`
Key metrics	▪ `CPUUtilization`: percentage of vCPU utilization on the instance ▪ `DiskReadOps`, `DiskWriteOps`: number of operations per second on attached disk ▪ `DiskReadBytes`, `DiskWriteBytes`: volume of bytes to transfer on attached disk ▪ `NetworkIn`, `NetworkOut`: number of bytes sent or received by network interfaces ▪ `NetworkPacketsIn`, `NetworkPacketsOut`: number of packets sent or received by network interfaces

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	▪ `GroupMinSize`, `GroupMaxSize`, `GroupDesiredCapacity`: minimum, maximum, and desired size of the Auto Scaling group ▪ `GroupInServiceInstances`: number of instances up and running in the Auto Scaling group ▪ `GroupTotalInstances`: total number of instances in the Auto Scaling group, regardless of state

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 cyclicality 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,