# Chapter 6. Auto Scaling and Elastic Load Balancing

# What Is Auto Scaling?

Most applications have peaks and troughs of user activity. Consumer web applications are a good example. A website that is popular only in the United Kingdom is likely to experience very low levels of user activity at three o’clock in the morning, London time. Business applications also exhibit the same behavior: a company’s internal HR system will see high usage during business hours, and often very little traffic outside these times.

Capacity planning is the process of calculating which resources will be required to ensure that application performance remains at acceptable levels. A traditional datacenter environment needs enough capacity to satisfy peak demand, leading to wasted resources during lulls in activity. If your application requires ten servers to satisfy peak demand and only one server during quiet times, up to nine of those servers are regularly going to waste.

Because of the amount of time it takes to bring physical hardware online, traditional capacity planning must take future growth into consideration; otherwise, services would be subject to outages just as they become popular, and systems administrators would spend more time ordering new hardware than configuring it. Getting these growth predictions wrong presents two risks: first, if your application fails to grow as much as expected, you have wasted a lot of money on hardware. Conversely, if you fail to anticipate explosive growth, you may find the continuation of that growth restrained by the speed in which you can bring new servers online.

In a Public Cloud environment, Auto Scaling allows the number of provisioned instances to more closely match the demands of your application, reducing wasted resources and therefore better managing costs. Auto Scaling also eliminates the procurement aspect of capacity planning, as the lead time required to commission a new server shrinks from multiple weeks to merely a few seconds.

An Auto Scaling group is a collection of one or more EC2 instances. As the level of activity increases, the system will scale up by launching new instances into this group. A subsequent decline in activity will cause the Auto Scaling system to scale down and terminate instances.

The way in which the level of activity is measured centers on the application. In combination with the CloudWatch monitoring service, Auto Scaling can use metrics such as CPU utilization to control scaling activities.

It is possible to submit custom metrics to CloudWatch and use these to trigger Auto Scaling events (i.e., scaling up or down). In the case of a batch processing system, you might wish to launch instances based on the number of items in the queue to be processed. Once the queue is empty, the number of running instances can be reduced to zero.

This is the elastic in Elastic Compute Cloud.

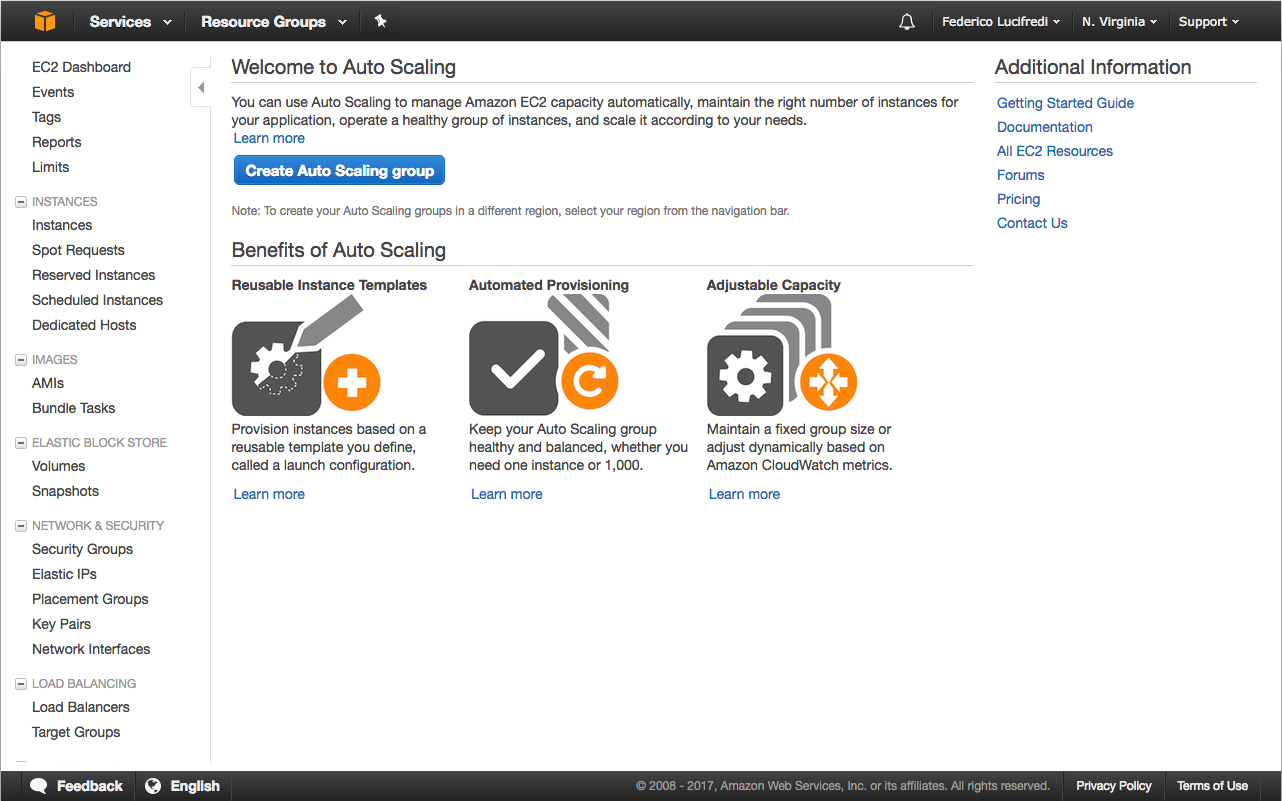
EC2 was not always so elastic. At the time it was launched, Amazon did not yet provide the Auto Scaling service as it exists today. Although EC2 instances could be launched and terminated on demand, performing these tasks was the responsibility of the user. As such, the pioneers who made heavy use of AWS in the early days built their own systems for managing the automatic creation and deletion of instances by interacting with the EC2 APIs.

As AWS continued to grow, Amazon built this logic into the EC2 service under the name of Auto Scaling. Initially requiring the use of command-line tools, the Auto Scaling API, or CloudFormation, Auto Scaling can now be managed entirely from the AWS Management Console.

If you do not use Auto Scaling, you will need to manually recover any failed instances on AWS. Even if you have only a single instance in the group, using Auto Scaling can save a lot of headaches. When an important instance crashes, would you rather scramble to launch a new one manually, or simply read the email from Amazon letting you know that a new instance is being launched?

The benefits of Auto Scaling are not limited to changes in capacity—using Auto Scaling enhances the resilience of your application, and is a required component of any production-grade AWS infrastructure. Remember that availability zones are physically separate data centers in which EC2 instances can be launched. Auto Scaling will distribute your instances equally between AZs. In the event of a failure in one AZ, Auto Scaling will increase capacity in the remaining AZs, ensuring that your application suffers minimal disruption.

The ability to automatically scale your instances comes with its own set of potential drawbacks. For example, consider how your application will handle distributed denial-of-service (DDoS) attacks. With a limited number of physical servers, a concentrated attack would eventually cause your site to crash. With Auto Scaling, your instances might scale up to meet the demand of the DDoS, which can become expensive very quickly. For this reason, you should always impose an upper limit on the number of instances that an Auto Scaling group can spawn, unless you are very sure that a DDoS cannot break the bank, as well as your application.



###### *Figure 6-1. The AWS Console was enhanced to support Auto Scaling in December 2013.*

This chapter introduces the core Auto Scaling concepts and puts them into practice by updating the example application from the previous chapter, adding resilience and scaling capabilities to the infrastructure.

# Static Auto Scaling Groups

Although Auto Scaling at its core revolves around the idea of dynamically increasing or reducing the number of running instances, you can also create a group with a specific number of instances that does not change dynamically. We refer to this as a static Auto Scaling group.

EC2 instances can be terminated without warning for various reasons outside of your control; this is one of the accepted downsides of operating in a public cloud. A manually launched instance—that is, an instance launched outside of an Auto Scaling group—would need to be replaced manually once its failure is noticed. Even if you are using CloudFormation to manage the instance, some manual interaction may still be required to bring the application back up. We already examined in [chapter 3](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#nunnikhoven) the use of Auto Scaling groups to maintain the number of instances required by an application while under security attack.

With Auto Scaling, a failed instance is automatically replaced as soon as the failure is detected by AWS. For this reason, we always use Auto Scaling for every production instance—even if there will always be only a single instance in the group. The small amount of extra work involved in configuring Auto Scaling is well worth the knowledge that if an entire AZ fails, the instance will be replaced immediately and without any effort on our part.

We will begin by modifying the CloudFormation stack from the previous example so that the web and Celery instances are contained within their own static Auto Scaling group. [Example 6-1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch06.html#basic_auto_scaling_stack) shows a CloudFormation stack template fragment describing the resources required to launch a working Auto Scaling group.

##### ***Example 6-1. Auto Scaling groups in CloudFormation***

"MyLaunchConfig" : {

"Type" : "AWS::AutoScaling::LaunchConfiguration",

"Properties" : {

"ImageId" : "*ami-XXXXXXXX*",

"SecurityGroups" : [ { "Ref" : "MySecurityGroup" } ],

"InstanceType" : "t2.micro"

}

},

"MyASGroup" : {

"Type" : "AWS::AutoScaling::AutoScalingGroup",

"Properties" : {

"AvailabilityZones" : ["us-east-1a", "us-east-1b", "us-east-1c", "us-east-1d", "us-east-1e"],

"LaunchConfigurationName" : { "Ref" : "MyLaunchConfig" },

"MinSize" : "1",

"MaxSize" : "1",

"DesiredCapacity" : "1"

}

}

There are two components to Auto Scaling: the Auto Scaling group and a launch configuration. A launch configuration controls which parameters are used when an instance is launched, including the instance type and user data.

###### NOTE

A full description of valid properties for an [Auto Scaling group](https://aws.amazon.com/cloudformation/details/) resource can be found in the CloudFormation section of the AWS documentation.

The size of an Auto Scaling group is controlled by the MinSize and MaxSize parameters, which set lower and upper bounds on the size of the group. The DesiredCapacity parameter specifies the ideal number of instances in the group. CloudFormation will consider the Auto Scaling group to have been created successfully only when this number of instances is running.

An Auto Scaling group must use at least one availability zone. The AvailabilityZonesparameter lets you control which AZs are used for the group—ideally, as many as possible if availability is a concern. Entire availability zones can—and have—become unavailable for extended periods of time. While Amazon AWS has been enjoying the [highest availability](https://www.enterprisetech.com/2015/01/06/aws-rates-highest-cloud-reliability/)among public cloud vendors, it has fallen short of achieving five nines of availability across all its regions. Any application that fails to make use of multiple availability zones in its architecture faces a high likelyhood of a number of hours of outage every year. If this is not acceptable to your users, you need to design accordingly. Amazon gives you the tools to build highly available systems, but it is up to you to use them.

The parameters given to the launch configuration are similar to the ones used with an EC2 instance resource. Group-level attributes—such as the number of instances in the group—are assigned to the Auto Scaling group resource.

To update our example stack to use Auto Scaling groups, we need to perform two steps, repeated for both instance types:

1. Create a launch configuration to replace the EC2 resource, using the same user data and instance type.
2. Create an Auto Scaling group resource using this launch configuration.

[Example 6-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch06.html#web_celery_auto_scale) shows the Auto Scaling groups and launch configurations for the web and Celery instances. Update the stack template (located at cloudformation/myblog.json) by removing the WebInstance and CeleryInstance resources, and adding the text in [Example 6-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch06.html#web_celery_auto_scale) to the Resources section.

##### ***Example 6-2. Auto Scaling web and Celery instances***

"CeleryLaunchConfig" : {

"Type" : "AWS::AutoScaling::LaunchConfiguration",

"Properties" : {

"ImageId" : { "Ref" : "CeleryAMI" },

"SecurityGroups" : [ { "Ref" : "CelerySecurityGroup" } ]

}

},

"CeleryGroup" : {

"Type" : "AWS::AutoScaling::AutoScalingGroup",

"Properties" : {

"AvailabilityZones" : { "Fn::GetAZs" : ""},

"LaunchConfigurationName" : { "Ref" : "CeleryLaunchConfig" },

"MinSize" : "1",

"MaxSize" : "2",

"DesiredCapacity" : "1"

}

},

"WebLaunchConfig" : {

"Type" : "AWS::AutoScaling::LaunchConfiguration",

"Properties" : {

"ImageId" : { "Ref" : "WebAMI" },

"SecurityGroups" : [ { "Ref" : "WebSecurityGroup" } ]

}

},

"WebGroup" : {

"Type" : "AWS::AutoScaling::AutoScalingGroup",

"Properties" : {

"AvailabilityZones" : { "Fn::GetAZs" : ""},

"LaunchConfigurationName" : { "Ref" : "WebLaunchConfig" },

"MinSize" : "1",

"MaxSize" : "2",

"DesiredCapacity" : "1"

}

},

The instance-specific parameters have been moved from the WebInstance resource to the WebLaunchConfig resource. The new Auto Scaling group resource will launch one of each instance type, as set by the DesiredCapacity parameter.

The next step is to update the running CloudFormation stack with the new template. Do this using the Management Console or command-line tools, and wait for the stack to reach the UPDATE\_COMPLETE state.

Because the WebInstance and CeleryInstance resources are no longer in the stack template, these two instances will be terminated by CloudFormation. Once the launch config and Auto Scaling group resources have been created, two new instances will be launched to replace them.

It is worth noting that instances launched as part of an Auto Scaling group are not included in the Resources panel in the Management Console. Instead, you will need to use the AWS CLI tool or the Management Console to list the members of an Auto Scaling group. Instances will also be automatically tagged with the name and ID of the CloudFormation stack to which their parent Auto Scaling group belongs, as well as any optional user-defined tags.

###### NOTE

The fact that EC2 instances are not, technically speaking, part of the CloudFormation stack has some interesting implications when updating running stacks.

Say you want to change the parameters of a launch configuration that is in use by some running instances. When you update the running stack, CloudFormation will create a new launch configuration, update the Auto Scaling group to reference the new launch configuration, and finally delete the old launch configuration.

By default, it will make no changes to the instances that are running at the time the stack is updated. The new launch configuration will apply only to newly launched instances, meaning that currently running instances will still be using the old launch configuration. In some cases, it is acceptable to let the new launch configuration gradually propagate as new instances are launched. In others, it is necessary to immediately replace the instances so they pick up the new configuration.

An update policy can be used to automatically replace instances when their underlying launch configuration is changed. Instances will be terminated in batches and replaced with new instances by the Auto Scaling service.

Now that the web and Celery instances are part of Auto Scaling groups, we can test the resilience of our application by terminating the Celery instance via the Management Console. If you browse the Mezzanine site while the Celery instance is terminated, everything will continue to function as normal, because the web application does not rely on a functioning Celery instance in order to work, because of the decoupled nature of the application. As tasks are received, they are placed in the SQS queue, where they will wait until there is a working Celery instance to process them.

When Amazon’s periodic instance health checks notice that the Celery Auto Scaling group no longer contains a working instance, a replacement will be launched. After a few minutes, the instance will become functional and process any tasks that are waiting in the SQS queue.

With any application, it is important to understand the failure characteristics of each component. How will your application cope when one or more of its components fail is perhaps the most important question to ask when designing an application for hosting in a public cloud environment.

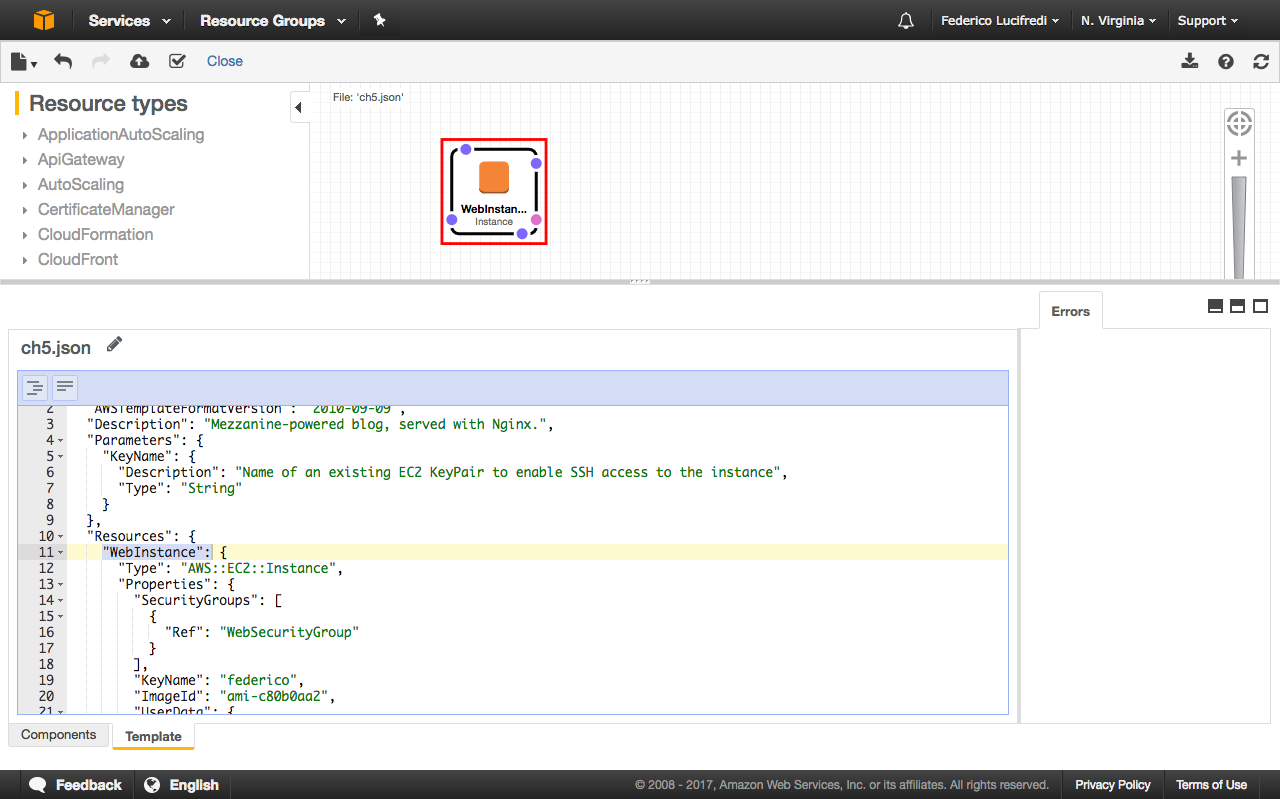
In the case of Celery, the failure characteristics are very good: the application continues working almost entirely as normal from the user’s perspective. Comments posted on the blog will be delayed for a while, which many users may not even notice.

A failed WebInstance, on the other hand, would cause the application to become entirely unavailable, because there is only one web instance in the group. Later in this chapter we will look at using load balancers to distribute traffic between instances.

###### NOTE

The authors feel strongly that CloudFormation will be a defining skill for system administrators in the coming decade. We accordingly expose the reader to copious amounts of its syntax as one of the most complex and valuable aspects of AWS worth mastering-yet there is a simpler way to draft CloudFormation templates.

In the fall of 2015, Amazon introduced [CloudFormation Designer](https://console.aws.amazon.com/cloudformation/designer), a UI offering drag-and-drop manipulation of resources in templates. Federico likes to use it as a convenient, always up-to-date CloudFormation editor with built-in syntax validation: even when handcrafting your templates, reducing trivial errors helps to save time ([Figure 6-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch06.html#cloudformation_designer)).



###### *Figure 6-2. A template from Chapter 5 validated in CloudFormation Designer.*

# Notifications of Scaling Activities

Another element of AWS is the Simple Notification Service (SNS). This is a push-based notification system through which an application can publish messages to topics. Other applications can subscribe to these topics and receive real-time notifications when new messages are available. This can be used to implement the publish/subscribe design pattern in your application.

In addition to notifying other applications when messages are published, SNS can also send notifications to email and SMS recipients, or post the message to an external HTTP web server. Auto Scaling groups can be optionally configured to publish SNS notifications when scaling activities take place, letting you receive an email each time new instances are launched or terminated.

[Example 6-3](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch06.html#auto_scaling_notifications) shows an updated version of our Celery scaling group with SNS notifications enabled. The example shows the four possible types of notifications that Auto Scaling will send. You can choose to subscribe to any combination of these types, but electing to choose all four can result in a lot of email traffic if your application regularly performs scaling activities.

##### ***Example 6-3. Auto Scaling with notifications***

"ScalingSNSTopic" : {

"Type" : "AWS::SNS::Topic",

"Properties" : {

"Subscription" : [ {

"Endpoint" : "*notifications@example.com*",

"Protocol" : "email"

} ]

}

}

"CeleryGroup" : {

"Type" : "AWS::AutoScaling::AutoScalingGroup",

"Properties" : {

"AvailabilityZones" : { "Fn::GetAZs" : ""},

"LaunchConfigurationName" : { "Ref" : "CeleryLaunchConfig" },

"MinSize" : "1",

"MaxSize" : "2",

"DesiredCapacity" : "1",

"NotificationConfiguration" : {

"TopicARN" : { "Ref" : "ScalingSNSTopic" },

"NotificationTypes" : [

"autoscaling:EC2\_INSTANCE\_LAUNCH",

"autoscaling:EC2\_INSTANCE\_LAUNCH\_ERROR",

"autoscaling:EC2\_INSTANCE\_TERMINATE",

"autoscaling:EC2\_INSTANCE\_TERMINATE\_ERROR"

]

}

}

},

##### THE IMPORTANCE OF ERROR NOTIFICATIONS

We strongly recommend subscribing to the INSTANCE\_LAUNCH\_ERROR notification type for any important Auto Scaling groups. This wil alert you to issues with Auto Scaling groups before they turn into real emergencies.

Mike once accidentally deleted an AMI that was still referenced in a production launch configuration, resulting in an Auto Scaling group no longer able to launch new instances.

This particular application—a social media website—had external monitoring that measured the performance of page-load times. Performance started to decrease as the running instances became increasingly overloaded. At the same time, Mike’s inbox began filling up with emails from AWS, letting him know that there was a problem with the scaling group. He quickly realized this was due to deleting the wrong AMI and set about building a new AMI to replace it. Subscribing to these notifications saved valuable time that would otherwise have been spent investigating the problem.

Operator error is not the only time in which these messages can prove useful. For example, if AWS is experiencing problems and cannot provide an instance to satisfy an Auto Scaling request, you will be informed.

Update the cloudformation/myblog.json file, replacing the CeleryScalingGroup resource with the new one. Remember to replace the example email address with your own. You could also add the NotificationConfiguration section to the WebScalingGroup resource if you would like to enable notifications for both scaling groups. After saving the file, update the running stack with the new template.

If you would like to see the notifications in action, terminate the Celery instance and wait for Auto Scaling to replace it. You should start receiving emails for both the termination and launch events, each letting you know which instance is being terminated and the reason for the change.

# Scaling Policies

Static Auto Scaling groups have their uses, but a primary reason to use AWS is its ability to scale compute capacity up and down on demand, shortcutting the need to purchase servers in advance.

There are two ways to configure Auto Scaling to automatically change the number of instances in a group: either at fixed time intervals, or on-demand based on measurements gathered by a monitoring system.

Scaling based on time is useful only when your usage patterns are highly predictable. The implementation process for this is described in detail on Amazon’s [Scaling Based on a Schedule](http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/schedule_time.html) page.

Dynamic scaling is the more interesting and widely used approach. It relies on gathering metrics—such as CPU utilization or requests per second—and using this information to decide when your application needs more or less capacity. This is done by creating scaling policies that describe the conditions under which instances should be launched or terminated. Scaling policies must be triggered in order to perform any action.

A policy that controls when new instances are launched is known as a scale-up policy, and one that controls when instances are terminated is a scale-down policy. Scaling policies can adjust the size of the Auto Scaling group in three ways:

* As an exact capacity. When the policy is triggered, the number of instances will be set to a specific number defined in the policy.
* As a percentage of current capacity.
* As an absolute value. When triggered, n new instances will be launched, where n is defined by the policy.

Scaling policies are triggered as a result of changes in measured metrics, which we will look at in the next section.

## Scaling on CloudWatch Metrics

CloudWatch is a monitoring system provided by Amazon, tightly integrated with most AWS services. It can be used to quickly set up a custom Auto Scaling configuration specific to your application’s needs. Basic metrics gathered at five minute intervals are available free of charge for services including compute (EC2), block storage (EBS), RDS database and ELB load balancing. The same metrics can be gathered at one-minute intervals but in most cases you will incur an additional cost at this higher sampling rate.

Custom metrics from third-party or self-hosted monitoring systems can be published to CloudWatch, allowing you to see this data alongside AWS-provided metrics.

CloudWatch’s Alarms feature can be used to send alerts when these metrics fall outside the levels that you configure. For example, you could receive an email notification when the average CPU load of an instance has been above 80% for at least 10 minutes.

By connecting alarms to scaling policies, CloudFormation metrics can be used to control the size of an Auto Scaling group. Instead of informing you by email that your CPU load is too high, Amazon can launch a new instance automatically.

CloudWatch can aggregate metrics for all instances in an Auto Scaling group and use the aggregated metrics to control scaling actions. If you have an Auto Scaling group consisting of a cluster of instances that are all processing an equal amount of work, the average CPU utilization across the entire group will probably be a good indicator as to how busy the cluster is.

Because there are so many metrics available in CloudWatch, it is worth taking some time to evaluate different scaling strategies to see which is best for your application’s workload. The right choice will depend on which resources are most heavily used by your application. Sometimes, it is not possible to identify a single metric that best identifies when capacity changes are required.

Take our Celery instance as an example. The task that checks a comment for spam merely contacts an external API and returns the result to the Celery broker. This is not a particularly intensive strain on the instance’s CPU because most of the task execution time will be spent waiting for responses to network requests. We could increase the parallelization of Celery by running more processes and making more efficient use of the CPU, but the instance likely will run out of RAM before saturating the CPU.

Unfortunately, it is not possible to measure RAM usage in CloudWatch directly, writing a client-side script to submit this data to the CloudWatch API is required.

##### MONITORING RAM UTILIZATION IN EC2 INSTANCES

Monitoring memory use requires the instance itself to publish the relevant data to CloudWatch. Amazon Web Services maintains (but does not support) a set of sample Perl scripts reporting [memory and disk usage](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/mon-scripts.html) to CloudWatch. The scripts were recently updated to work out of the box on Ubuntu 14.04 and 16.04, as well as a number of older RPM-based distributions.

Standard CloudWatch [pricing](https://aws.amazon.com/cloudwatch/pricing) for custom metrics applies to this approach. Unfortunately, the design of these scripts requires storing a set of AWS credentials on the instance itself, hardly what we would consider a security best practice.

Fortunately, Shahar Evron has devised an [alternative approach](http://arr.gr/blog/2013/08/monitoring-ec2-instance-memory-usage-with-cloudwatch/) making use of IAM roles. You will remember that roles can be used to assign capabilities to AWS instances, in this case granting monitored instances the ability to report metrics to CloudWatch. Shahar’s solution uses a small [boto script](https://gist.github.com/shevron/6204349#file-cw-monitor-memusage-py) to accomplish just this, and can equally support the use of roles or AWS credentials stored in the ~/.boto and ~/.aws/credentials configuration files. No modules are required, boto is the sole dependency.

Following the process we outlined in [Chapter 3](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#chap_aws_access_security), you will need to create a role that EC2 instances can assume, and embed a policy granting access to the cloudwatch:PutMetricData action on the CloudWatch resource. Once this is done, create an IAM instance profile as we have shown and use it to launch a new instance. Log in, and execute the following configuration steps:

sudo apt install python-pip

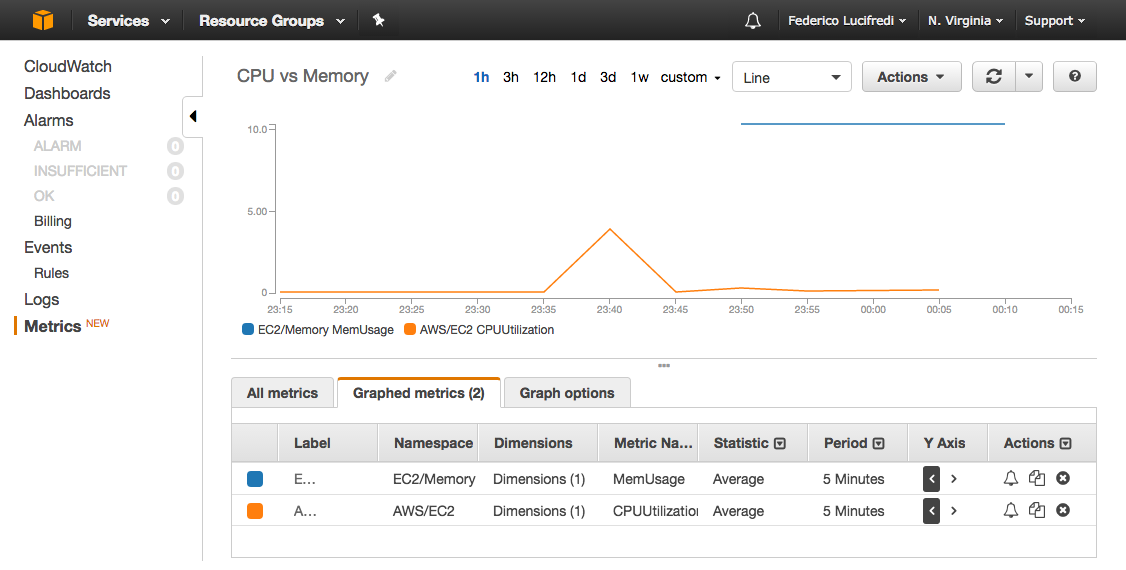
sudo pip install boto

curl https://gist.githubusercontent.com/shevron/6204349/raw/cw-monitor-memusage.py | sudo tee /usr/local/bin/cw-monitor-memusage.py

sudo chmod +x /usr/local/bin/cw-monitor-memusage.py

echo "\* \* \* \* \* nobody /usr/local/bin/cw-monitor-memusage.py" | sudo tee /etc/cron.d/cw-monitor-memusage

This will set up Cron to report every minute the memory usage of this instance to CloudWatch-you will see an EC2/memory namespace populated by the MemUsagemetric appear in your CloudWatch dashboard. This configuration can easily be automated as part of your default instance setup, and you can see how this all comes together in [Figure 6-3](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch06.html#memory_dashboard).



###### *Figure 6-3. Adding RAM metrics to our CloudWatch dashboard.*

Because we are using the SQS service as a Celery broker, we have another option: scaling based on the number of messages in the queue, rather than on instance metrics. This is interesting because we can use one AWS service (SQS) to control another (Auto Scaling groups), even though they are not directly connected to one another.

We will use the number of messages waiting in the SQS queue to control the size of the Celery Auto Scaling group, ensuring there are enough instances in the group to process tasks in a timely manner at all times.

We know that our tasks are usually processed very quickly and the SQS queue is usually empty, so an increase in the queue length indicates either a legitimate increase in tasks or a problem with the Celery instances. Regardless, launching new instances will solve any problems caused by load and force the size of the queue to decrease.

The same metric can be used to terminate instances after the queue has been reduced to an acceptable length. Running too many instances is a waste of money, so we want to keep the Auto Scaling group to be as small as possible.

##### STARTING SERVICES ON AN AS-NEEDED BASIS

Scaling policies respect the minimum and maximum size of your Auto Scaling groups. Because the minimum size of our Celery group is 1, CloudFormation will never terminate the last remaining instance.

By setting the minimum size of the group to 0, you could build a system where instances are launched only when messages are published to the queue. To understand the value of such a policy, imagine using Celery to send out batches of emails at regular intervals. Most of the time the queue will be empty. When you begin publishing messages to the queue, instances will be launched to process the tasks. Once the queue is empty, all instances will be terminated. This is an incredibly cost-effective way to run a task-processing infrastructure.

To implement these changes, we need to make further additions to the stack template, as shown in [Example 6-4](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch06.html#example_auto_scaling_celery_alarms).

##### ***Example 6-4. Auto Scaling with CloudWatch alarms***

"CeleryScaleUpPolicy" : {

"Type" : "AWS::AutoScaling::ScalingPolicy",

"Properties" : {

"AdjustmentType" : "ChangeInCapacity",

"AutoScalingGroupName" : { "Ref" : "CeleryGroup" },

"Cooldown" : "1",

"ScalingAdjustment" : "1"

}

},

"CeleryScaleDownPolicy" : {

"Type" : "AWS::AutoScaling::ScalingPolicy",

"Properties" : {

"AdjustmentType" : "ChangeInCapacity",

"AutoScalingGroupName" : { "Ref" : "CeleryGroup" },

"Cooldown" : "1",

"ScalingAdjustment" : "-1"

}

},

"CelerySQSAlarmHigh": {

"Type": "AWS::CloudWatch::Alarm",

"Properties": {

"EvaluationPeriods": "1",

"Statistic": "Sum",

"Threshold": "100",

"AlarmDescription": "Triggered when SQS queue length >100",

"Period": "60",

"AlarmActions": [ { "Ref": "CeleryScaleUpPolicy" } ],

"Namespace": "AWS/SQS",

"Dimensions": [ {

"Name": "QueueName",

"Value": { "GetAtt": ["CeleryQueue", "QueueName"] }

} ],

"ComparisonOperator": "GreaterThanThreshold",

"MetricName": "ApproximateNumberOfMessagesVisible"

}

},

"CelerySQSAlarmLow": {

"Type": "AWS::CloudWatch::Alarm",

"Properties": {

"EvaluationPeriods": "1",

"Statistic": "Sum",

"Threshold": "20",

"AlarmDescription": "Triggered when SQS queue length <20",

"Period": "60",

"AlarmActions": [ { "Ref": "CeleryScaleDownPolicy" } ],

"Namespace": "AWS/SQS",

"Dimensions": [ {

"Name": "QueueName",

"Value": { "GetAtt": ["CeleryQueue", "QueueName"] }

} ],

"ComparisonOperator": "LessThanThreshold",

"MetricName": "ApproximateNumberOfMessagesVisible"

}

},

Insert this template excerpt into the Resources section of the cloudformation/myblog.jsonfile.

Notice that we do not need to change any aspect of the Celery Auto Scaling group resource in order to enable dynamic scaling. Our scaling policy configuration is entirely separate from the Auto Scaling group to which it applies. The scaling policy could even be in a separate CloudFormation stack.

We have separate policies for scaling up and down. They both use theChangeInCapacity adjustment type to launch or terminate a set number of instances.

The CeleryScaleUpPolicy, when triggered, will launch two new Celery instances. The CeleryScaleDownPolicy will terminate one instance at a time. Why the difference? Launching two new instances at a time lets us quickly respond to changes in demand, springing into action as the work requirements increase. As the queue drops, we want to gradually reduce the number of instances to avoid a yo-yo effect. If we reduce the capacity of the task-processing infrastructure too quickly, it can cause the queue to begin rising again, which might trigger the scale-up policy. At times, the Elastic Compute Cloud can be a little too elastic and introducing some hysteresis through our scale planning is necessary.

The Cooldown property gives us a further means of controlling the elasticity of our Auto Scaling policy. This value, specified in seconds, imposes a delay between scaling activities to make sure the size of the group is not adjusted too frequently.

CelerySQSAlarmHigh is a CloudWatch Alarm resource that monitors the length of the SQS queue used for Celery tasks. When there are more than 100 messages in the queue, this alarm is activated, triggering the CeleryScaleUpPolicy. Conversely, CelerySQSAlarmLow triggers the CeleryScaleDownPolicy when the queue length drops below 20. In practice, it is unlikely that the queue length thresholds will be so low. However, these values make it much easier to test and demonstrate that Auto Scaling is working as planned.

After saving the updated file, update the running stack with the new template. Because the DesiredCapacity of the group is still set to 1 and none of the relevant CloudWatch alarms have been triggered, nothing will actually happen yet.

To demonstrate that Auto Scaling is working, stop the Celery process on the running instance and post some test comments, causing the number of queued messages to increase until Celery is scaled up.

Using the Management Console or command-line tools, find the public DNS name of the instance in the Celery group. Remember that it will be tagged with the name of the CloudFormation stack and role=celery. Log in to the instance and stop Celery with the following command:

supervisorctl celery stop

Visit the Mezzanine page in your web browser and post example comments. In another tab, open the CloudWatch Alarms page and watch the status of the CelerySQSHighAlarm. Once enough messages have been published to the queue, it will enter the ALARM state and trigger the CeleryScaleUpPolicy, launching two new Celery instances.

Because we configured notifications for this scaling group, you will receive a few email messages as the Auto Scaling activities are performed. After a brief period, you should see there are now three running Celery instances.

Notice that they are probably all running in different availability zones within your region. Amazon will attempt to evenly distribute an Auto Scaling group across an EC2 region to enhance resilience.

The two new instances will quickly process the tasks in the queue and take the queue length below the scale-down threshold. Once the CelerySQSLowAlarm is triggered, two of the instances will be terminated.

###### TIP

When terminating instances, the default behavior is to terminate the instance that has the oldest launch configuration. If more than one instance is running the old configuration, or all instances are running the same configuration, AWS will terminate the instance that is closest to the next instance hour. This is the most cost-effective strategy, as it maximizes the useful lifetime of instances.

If instances were launched together—as is likely in an Auto Scaling group—more than one instance will be “closest” to a full instance hour. In this case, a random instance from this subset is terminated.

This logic can be further controlled by assigning a termination policy to the Auto Scaling group, as is described in Amazon’s [Auto Scaling documentation](http://docs.aws.amazon.com/autoscaling/latest/userguide/as-instance-termination.html).

Now that you know Auto Scaling is working, you can resume the Celery process on the original instance, assuming it was not terminated when scaling down. Do this with the following:

supervisorctl celery start

The Celery part of the infrastructure will now grow and shrink dynamically, according to the number of tasks to be processed. Tasks will be processed as quickly as possible, while ensuring that we are not wasting money by running too many instances.

# Elastic Load Balancing

Whether in the cloud or on your own hardware, system failures are an inevitable part of a system administrator’s life. If your application is hosted on a single server, the eventual system failure will render your application unavailable until a replacement server or virtual instance can be provisioned.

One way to improve the reliability of your application is to host it on multiple instances and distribute the traffic between them. When individual instances fail, your application will continue to run smoothly as long as the remaining instances have enough capacity to shoulder the burden of the additional requests they must now serve. This is known as load balancing, and the server that distributes traffic is a load balancer.

We have seen how Auto Scaling can be used to help solve this problem in AWS by automatically launching new instances to replace failed ones, or dynamically increasing capacity to respond to demand. So far, we have added dynamic Auto Scaling only to the Celery part of the infrastructure.

Converting the web application instance into an Auto Scaling group requires solving a problem not present with Celery: how do we distribute HTTP requests between multiple instances? For testing purposes, we have been using the public DNS name of each individual instance so far, and newly launched instances would launch with different names. Once the infrastructure is in production, users will visit it at http://blog.example.com. So how can we connect a single, public DNS name to a group of instances?

Elastic Load Balancing is Amazon’s solution to this problem. An Elastic Load Balancer (ELB) is a virtual device built specifically to provide dynamic load-balancing capabilities to applications hosted on AWS. An ELB effectively sits in front of a group of EC2 instances and distributes traffic between them. A round robin approach is commonly used, but alternative, application-aware strategies such as least outstanding requests are also popular.

Instead of pointing the blog.example.com DNS record toward a specific EC2 instance, we will point it at the ELB using a CNAME DNS record. All requests will be sent by the ELB to the instances behind the ELB. You can use the Management Console and API to manually add and remove instances from the ELB. The ELB will regularly perform health checks on these instances, and any instances that are deemed unhealthy will be automatically removed from the ELB.

###### NOTE

Mastery of elastic load balancers requires a solid understanding of the DNS system. DNS itself is the broadest of subjects, covered by books much heftier than ours. We recommend [DNS an Bind](http://shop.oreilly.com/product/9780596100575.do) by Cricket Liu and Paul Albitz (O’Reilly) as the ultimate source on this topic. Federico’s now dog-eared copy saw an impressive amount of use during his graduate student years!

# Elastic Load Balancer and Auto Scaling Groups

ELBs are designed to work in conjunction with Auto Scaling Groups. When you create a scaling group, you can specify the name of the associated ELB. New instances launched in this scaling group will be automatically placed behind the ELB, at which point they will begin receiving traffic and serving requests.

We will use this feature to convert the web application component of the example stack into an Auto Scaling group behind an ELB. Once finished, we will be able to access our example blog via the public DNS name of the ELB.

[Example 6-5](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch06.html#example_autoscaling_elb) shows the updated section of the example application stack.

##### ***Example 6-5. Auto Scaling group with Elastic Load Balancer***

"WebELB" : {

"Type" : "AWS::ElasticLoadBalancing::LoadBalancer",

"Properties" : {

"AvailabilityZones" : { "Fn::GetAZs" : "" },

"Listeners" : [ {

"LoadBalancerPort" : "80",

"InstancePort" : "80",

"Protocol" : "HTTP"

} ],

"HealthCheck" : {

"Target" : { "Fn::Join" : [ "", ["HTTP:80/"]]},

"HealthyThreshold" : "3",

"UnhealthyThreshold" : "5",

"Interval" : "30",

"Timeout" : "5"

}

}

},

"WebGroup" : {

"Type" : "AWS::AutoScaling::AutoScalingGroup",

"Properties" : {

"AvailabilityZones" : { "Fn::GetAZs" : ""},

"LaunchConfigurationName" : { "Ref" : "WebLaunchConfig" },

"MinSize" : "1",

"MaxSize" : "2",

"DesiredCapacity" : "2",

"LoadBalancerNames" : [ { "Ref" : "WebELB" } ]

}

},

Update cloudformation/myblog.json to replace the WebInstance resource with the new code. These are almost exactly the same changes required to convert Celery into a scaling group, with two main exceptions: an Elastic Load Balancer resource has been added, and the Auto Scaling group has been updated to include the LoadBalancerNames parameter.

The DesiredCapacity for the WebGroup has been changed to 2, which means an additional instance will be launched when the CloudFormation stack is updated.

The ELB performs a health check that verifies that the application is responding to HTTP requests.

# ELB Health Checks

An ELB can be configured with a custom health check that is performed on any instance in the group. This is done by making an HTTP request to the specified target at regular intervals. If the target server does not respond to this request with a 200 status code within the given time-out, the check fails. Once the number of failed checks reaches theUnhealthyThreshold value, the instance is considered unhealthy and removed from the ELB. As a result, an instance could be replaced if it becomes very slow or unreliable, even if it doesn’t completely fail.

The health check will still be performed on unhealthy instances. Should they recover and resume operating normally, they will be automatically returned to the ELB after enough successful checks have occurred, as specified in HealthyThreshold.

Another option for the custom health check is to initiate a TCP connection. If the TCP handshake can be completed, the check is considered successful.

HTTP-based health checks are useful for more than web applications. A really useful practice is to build a custom HTTP server that represents the health of your application. This can consist of a number of tests specific to your environment, which verify that every component of the instance is working as expected. If any of these tests fail, have the server return an HTTP 500 error code in response to the health check, which results in the instance being removed from the ELB.

In addition to the custom health check just described, Amazon performs an instance-level health check automatically. This checks for problems at the virtualization layer that might not be recognized by an application-level health check.

While health checks are useful, it is important to understand the implications of allowing health checks to terminate and launch new instances. A misconfigured health check will cause ELB to believe all of your instances are unhealthy, and unceremoniously remove them from the group. It is imperative that your health check return a 500 error only if the problem can be solved by removing the instance from the ELB. That is the only remedial action the ELB can take, so there is no point in alerting it about problems that require another solution, and doing so may be actually counter-productive.

Consider the example of a web application that requires a database to function properly. If the database fails, each web instance will begin returning 500 error codes for every request. Removing an individual instance of the web server from the ELB will do nothing to bring the database back to life.

If the health check page also returns 500 error codes because the database is not working, all of the instances will be removed from the ELB. At this point, visitors to your website would see the standard ELB error page, rather than the beautifully designed error page of your application.

Carefully monitor the number of unhealthy instances in each ELB by setting up a CloudWatch alarm to alert you if it remains above zero for a significant length of time.

Update the running CloudFormation stack with the modified template. The old web instance, which no longer exists in the template, will be deleted and replaced with an Auto Scaling group with a single member.

Once the stack has finished updating, use the CloudFormation Management Console or command-line tools to list the stack’s resources and find out the public DNS name of the ELB. Alternatively, you could look in the Elastic Load Balancers section of the Management Console.

Visit the ELB’s address in your web browser, and you should once more see the Mezzanine welcome page. This page has been served by one of the instances in the scaling group. Because they are identical, it does not matter which one actually served the request.

###### TIP

The number of requests received per second is one of the Elastic Load Balancing metrics in CloudWatch. As such, it can be used to scale the number of web instances in a manner directly related to current levels of traffic.

Terminate one of the running web application instances with the Management Console. You can find it by searching for one of the values contained in the instance’s tags, such as the scaling group name (WebInstanceScalingGroup).

After this instance terminates, refresh the Mezzanine page in your browser. The site will continue to function as normal even though one of the instances is no longer running.

Within a few minutes, AWS will notice the terminated instance and launch a replacement for it. Once this instance has finished the startup process, it will be marked as healthy by the ELB, and traffic will once again be split between the two instances.

With this step complete, the example application is now ready to scale at both the Celery and web levels. It is resilient to failures of individual instances and entire availability zones, and will run at the optimal capacity to ensure happy users and a low AWS bill for your organization.

# Managing Outages

Amazon Web Services has come a long way from its early days, when it was routine for any medium-scale deployment to lose a few EC2 instances at least once every week. Instances no longer “disappear” as often, thanks to the dedication of the AWS team to increasing reliability for their customers, but this remains a distinctive feature of spot priced instances, purchased by users bidding for spare excess capacity. Any application running on EC2 must be designed to account for the possibility of an instance being terminated without any prior warning. Production datacenters present similar risks, including the loss of a hypervisor node or a network failure, but in the AWS model that opaque risk cannot be managed by the administrator in even the smallest degree, as she is unaware of Amazon’s daily operational plans and needs-or the presence of any backhoes near their fiber-optic lines.

###### TIP

The [spot instance market](https://aws.amazon.com/ec2/spot/pricing/) allows AWS customers to bid for spare excess capacity by specifying what maximum instance-hour price they are willing to pay. EC2 launches (or terminates) instances as the spot price fluctuates based on the supply and demand of unused EC2 capacity. Operating in this model can result in massive savings over on-demand instance pricing and is very suitable to applications that can be safely interrupted.

The spot market is becoming increasingly sophisticated over time, introducing new choices for the user. Spot priced instances now receive a two-minute notification of impending termination once the price exceeds the user-specificed limit, and partially-used hours are not charged. It is now also possible to define a minimum duration requirement, in which case a running instance will not be terminated even if the spot price exceeds the specified constraint.

Spot instance termination notices are provided through the termination-timeitem in the instance’s metadata. If a spot instance is marked for termination, the item will be defined, its value specifying at which time the instance will receive the ACPI shutdown event. Amazon recommends polling this field at five second intervals, which is easily accomplished as follows:

if curl -s http://169.254.169.254/latest/meta-data/spot/termination-time | grep -q .\*T.\*Z

then

echo terminated;

fi

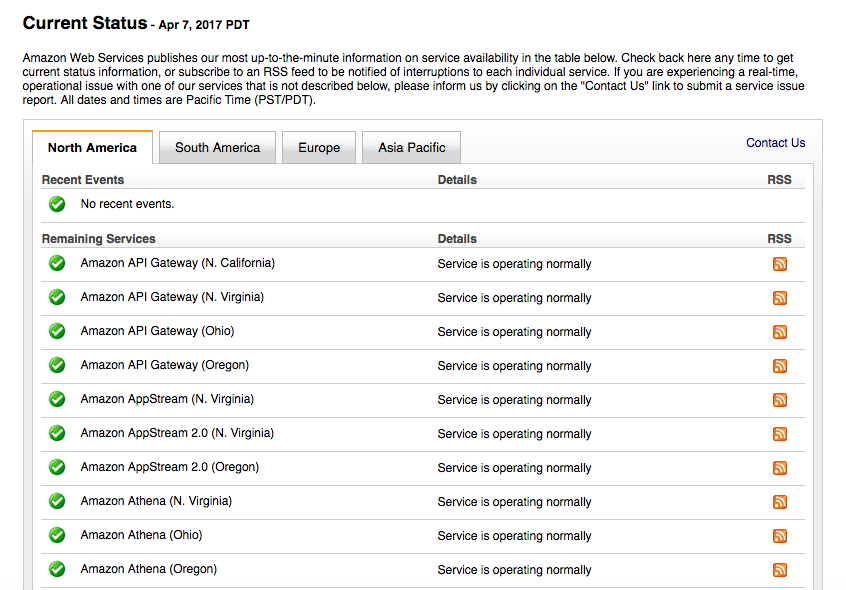
A properly executed [Mode 2 IT](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch02.html#pets_vs_cattle) transition requires architectural design effort: we should not fall prey to the classic mistake of trying to make a new technology perform exactly like the old did. We should instead use the valuable opportunity created by the infrastructure renewal cycle to modernize applications and reduce technical debt as part of the transition. Doing so comfortably uses existing legacy systems to continue to take care of what they do best: running our applications as originally intended, while we design their next implementation making the best use of new AWS infrastructure. There is no reason to migrate everything to new infrastructure once: we can deliberately choose our timeline and application migration strategy to accomplish these goals, while remining mindful of our budget constraints. Start by migrating those applications best suited to run in the cloud, while you make your way through the redesign of ill-fitting workloads.

Netflix’s [Chaos Monkey](https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey) is the tool of choice for validating an application’s response to EC2 instance outages. Chaos Monkey is a service that may be configured to randomly select and terminate instances in your AWS deployment, thereby testing the resiliency of your design and the automated responses set up to handle such contingencies. Failures happen, and unleashing Chaos Monkey on your infrastructure insures that the flaws in its design are found during business hours and at the best time to investigate their cause, instead of being discovered at the worst possible moment in production.

AWS Infrastructure outages remain rare, but are by no means unheard of. We live through the experience of a significant AWS outage every other year, and can observe its impact ripple through the most disparate services of the global internet. Failures cascade as increasingly more web sites are built around core infrastructure supplied by Amazon Web Services-An AWS outage can disrupt Twitter, which uses S3 to host files, which in turn can disrupt another service relying on Twitter to deliver its user notifications. And so on.

Some of AWS’ most notorious service events to date have included a storage component. On [October 22, 2012](https://aws.amazon.com/message/680342/) an unlikely combination of minor bugs in a monitoring system for the Elastic Block Storage service initiated a massive concurrent failover of volumes hosted in the us-east-1 region. Operations in the region ground to a halt for many users, who also encountered difficulty in controlling their resources through the management API. Similarly, on [February 28, 2017](https://aws.amazon.com/message/41926/) an operator on the AWS team fat-fingered a routine command intended to remove a small number of servers supporting the S3 billing process. The additional servers removed supported the metadata and placement subsystems for a whole S3 region. While these systems were being restarted, S3 was unable to service requests. Services that rely on its API, like EC2 and EBS, were severely affected. Both of these events were less than a day in duration yet they were front-page news in the technical press because of the scale of their impact-the 2017 outage severly affected major sites like Twitter, Slack, and GitHub and was reported on CNN.

We discuss these incidents here to illustrate the complexity of an AWS outage-they typically arise from a low-probability event that becomes possible (or even eventually inevitable) with AWS’s massive scale. It can be difficult for any administrator to determine if the unlikely circumstance of an AWS outage is underway at a given moment, which is why Amazon provides users with the [AWS Service Health Dashboard](http://status.aws.amazon.com/) ([Figure 6-4](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch06.html#service_health)), publishing the most up-to-date status information available for every AWS service. A status history covering all services for the last 365 days is also provided, as well as an RSS feed to subscribe to the health status of each service.



###### *Figure 6-4. A small section of the many pages of the AWS Service Health Dashboard.*

While it is standard practice to rely on multiple availability zones to guarantee services are highly availabile, some of the worst outages have affected multiple AZs within the same region. Even in the direst of circumstances, no AWS outage has ever brought down an AWS service across multiple regions concurrently. If your organization cannot afford a few hours of downtime annually as the worst-case scenario, you should analyze the cost and complexity required to operate your service from multiple AWS regions concurrently, as opposed to merely multiple availability zones.

# Mastering Scale

Auto Scaling policies are tricky to get right the first time, and it is likely that you will need to tweak these as your application workload changes. Err on the side of caution—in most cases, it is best to run with a little extra capacity, rather than having too little capacity and offering users a poor experience.

Auto Scaling enables some creative thinking. Do you have an internal HR system that is used only during office hours? Use scheduled Auto Scaling to automatically launch and terminate the instance so it is running only when needed.

What about building stacks that automatically launch when needed and self-destruct once their work is complete? The minimum size of an Auto Scaling group is zero. The Celery stack could be configured to launch instances when there are more than 50 messages in the queue.

Now you’re thinking with Auto Scaling groups.