# Chapter 11. Monitoring

Monitoring dynamic instances can be a challenge. The classic monitoring tools expect your instances to be around for a long time, and can have difficulty recognizing the difference between an instance that has failed and an instance that has been terminated as part of an Auto Scaling event or other planned termination in response to changes in capacity requirements.

AWS provides its own monitoring service, CloudWatch, which has been designed from the ground up to work in such an environment. Additionally, with some planning and custom scripts, most traditional monitoring tools can be used to monitor dynamic instances without spamming operators with false alarms when instances are terminated. This chapter shows you some of these methods, as well as Amazon’s CloudWatch service, and how the two can be integrated.

###### NOTE

A cottage industry of cloud-based monitoring tools has sprung up around AWS and other cloud providers. There are too many tools to mention in this book, and each has its own strengths and weaknesses.

The tighter these tools integrate with AWS, the more useful they are. The more advanced tools automatically query the EC2 tags associated with instances and use them to aggregate metrics. This allows you, for example, to either view a high-level overview of your application across all EC2 regions, or drill down to view the performances of instances in a particular availability zone.

# Why Are You Monitoring?

There are many reasons for setting up a monitoring system for your application and infrastructure. The most obvious reason is that when things break, operators should be alerted by automated systems, rather than phone calls from upset customers or C-level executives.

Another reason is to allow systems administrators to identify trends in application usage so they can make informed decisions about capacity requirements. Knowing how your application performed last month is critical when it comes to planning your requirements for next month.

Yet another reason is to allow administrators and developers to accurately measure the effects of infrastructure changes and new application features. It is difficult to improve what you do not measure. If your application is running slowly, a well-planned monitoring system will allow you to quickly identify and remove bottlenecks. You need metrics demonstrating that your changes are having a positive effect in order to make continual, incremental improvements, to keep the aforementioned C-level executives happy. The opposite is also true—it is easy to accidentally reduce the performance of your application when deploying new code on a frequent basis. In this case, it is imperative to recognize which changes are negatively affecting performance so that they can be reverted or fixed.

# CloudWatch

Amazon’s own CloudWatch service is the starting point for many administrators when it comes to monitoring AWS services. In fact, many AWS services, such as Auto Scaling, rely on CloudWatch to perform scaling operations, making it an essential part of the infrastructure. For example, CloudWatch is responsible for monitoring metrics, such as the CPU loads of EC2 instances. When these metrics cross certain thresholds, the Auto Scaling system is alerted so that it can take the relevant action of spawning or terminating instances according to an Auto Scaling policy.

CloudWatch is also an integral part of Amazon’s Health Check feature, used by Elastic Load Balancers to identify instances that have failed or are otherwise “unhealthy.”

In addition to tracking built-in metrics, such as disk usage and CPU load, CloudWatch can monitor [custom metrics](http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/publishingMetrics.html) provided by external sources. If the built-in metrics do not provide the detail required to inform your Auto Scaling requirements, these custom metrics can provide more granular control when scaling Auto Scaling groups up or down. The sky’s the limit when using custom metrics. Typically, administrators monitor values such as requests per second, although more outlandish metrics, such as solar flare activity or the phase of the moon could be used if it makes sense for your application.

Although powerful, CloudWatch is not perfect and has some drawbacks. The largest, in my opinion, is its web-based interface. When monitoring a large number of metrics, the interface can become slow and cumbersome, taking a long time to display the results you are looking for. This is because graphs are generated on demand each time they are viewed. So if you have many metrics, or are viewing data across a large time range, generating these graphs can take a long time and become quite frustrating.

Furthermore, a cost is associated with submitting custom metrics to CloudWatch. The cost is based on the number of metrics submitted, and the number of API requests required to submit these requests. At the time of writing, the cost is $0.50 per metric per month, plus $0.01 per 1,000 API requests. In most cases, each metric submission will require one API PUT request. Assuming you are submitting your custom metric every minute, this will result in a cost of around $0.43 per month. While not overly expensive, these costs can quickly add up and must be taken into consideration when building your monitoring system. Of course, building a custom monitoring system will have other costs: the time taken to implement it, licensing costs for third-part monitoring services, and so on.

## Auto Scaling and Custom Metrics

One of the most useful features of CloudWatch is its integration with Auto Scaling. This is commonly used to increase or decrease capacity in an Auto Scaling group according to metrics such as CPU utilization. When your instances are becoming too busy to cope with demand, more instances are launched. As demand decreases, the surplus instances are gradually terminated.

Auto Scaling is not limited to metrics that are built into CloudWatch: it is also possible to scale up or down based on the values of custom metrics that you provide to CloudWatch. As an example, consider a task-processing application in which tasks are queued in a messaging system and processed by EC2 instances. When running such an application, you might want the number of EC2 instances to scale dynamically according to the number of tasks waiting in the queue. If your message processing system is based on Amazon’s Simple Queue Service, you are able to use CloudWatch’s built-in metrics (such as the ApproximateNumberOfMessagesVisible SQS metric, which shows the number of messages available for retrieval in the queue) to control the size of your Auto Scaling group.

If you are using something other than SQS to store your queued messages for your task-processing application, you will need to provide CloudWatch with the data yourself so that Auto Scaling can make decisions based on these metrics.

Custom metrics do not need to be predefined: simply send the metric to CloudWatch, and it will begin storing and graphing it for you. This can be done in a number of ways. The Amazon API can be used from language-specific libraries (such as Boto for Python or Fog for Ruby), or by sending requests using the RESTful API. The most straightforward method is to use the AWS CLI tool.

In the following example, we will create a WaitingTasks custom metric, which performs the same function as the ApproximateNumberOfMessagesVisible metric for SQS-based systems. Once CloudFormation has some data on your custom metric, it can be used in the same way as built-in metrics to control Auto Scaling processes.

To begin sending our custom metric to CloudWatch, execute the following command:

aws cloudwatch put-metric-data --namespace "MyAppMetrics" --metric-name WaitingTasks --value 20 --unit Count

This example creates a WaitingTasks metric and provides an initial value of 20. After a few moments, this metric will become visible in the CloudWatch Management Console.

Instead of providing values on the command line, you can achieve the same result by sending a JSON file containing the metric data. For example, you could create a file named metric\_data.json containing the following contents:

[

{

"MetricName": "WaitingTasks",

"Value": 20,

"Unit": "Count"

}

]

This file can be uploaded with the following command:

aws cloudwatch put-metric-data --namespace "MyAppMetrics" --metric-data file://metric\_data.json

This is equivalent to the first command in this section and is most useful when providing more complex or detailed metrics.

[Chapter 6](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch06.html#aws_sysadmin_autoscaling) describes how to create and manage Auto Scaling groups. In this section, we will create a CloudFormation stack that describes an Auto Scaling group that dynamically shrinks and grows based on the value of our WaitingTasks metric. An example of such a stack is as follows:

{

"AWSTemplateFormatVersion" : "2010-09-09",

"Description" : "Auto Scaling on Custom Metrics",

"Resources" : {

"CustomMetricLaunchConfig" : {

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

"Properties" : {

"ImageId" : "ami-12345678",

"InstanceType" : "m1.small"

}

},

"CustomMetricScalingGroup" : {

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

"Properties" : {

"AvailabilityZones" : [ "eu-west-1" ],

"Cooldown" : "300",

"DesiredCapacity" : "1",

"LaunchConfigurationName" : "CustomMetricLaunchConfig",

"MaxSize" : "10",

"MinSize" : "1"

}

},

"ScaleUpPolicy" : {

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

"Properties" : {

"AdjustmentType" : "ChangeInCapacity",

"AutoScalingGroupName" : { "Ref" : "CustomMetricScalingGroup" },

"ScalingAdjustment" : "1"

}

},

"ScaleDownPolicy" : {

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

"Properties" : {

"AdjustmentType" : "ChangeInCapacity",

"AutoScalingGroupName" : { "Ref" : "CustomMetricScalingGroup" },

"ScalingAdjustment" : "-1"

}

},

"WaitingTasksAlarm" : {

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

"Properties" : {

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

"ComparisonOperator" : "GreaterThanThreshold",

"EvaluationPeriods" : "1",

"MetricName" : "WaitingTasks",

"Namespace" : "MyAppMetrics",

"OKActions" : [ { "Ref" : "ScaleDownPolicy" } ],

"Statistic" : "Maximum",

"Threshold" : "10",

"Unit" : "Count"

}

}

}

}

This stack consists of several components, which are all required to make everything work correctly.

CustomMetricLaunchConfig and CustomMetricScalingGroup should be familiar from [Chapter 6](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch06.html#aws_sysadmin_autoscaling). These are a required part of any Auto Scaling group.

Next, we have ScaleUpPolicy and ScaleDownPolicy. These scaling policy resources control how the capacity of an Auto Scaling group should be changed. The scale-up policy has a ScalingAdjustment parameter of 1, which means a single additional EC2 instance should be launched into the Auto Scaling group when this policy is triggered. The scale-down policy’s ScalingAdjustment parameter is −1, meaning a single instance should be removed from the group when this is triggered.

The ScalingAdjustment parameter controls what changes will be made to the size of the Auto Scaling group when this policy is triggered. The AutoScalingGroupName parameter associates the scaling policy with a particular Auto Scaling group.

The final component is WaitingTasksAlarm, which ties everything together and controls when the capacity should be changed. The important parts of this resource are theAlarmActions and OKActions parameters, which state what should happen when it enters and leaves the Alarm state. It enters the Alarm state when the specified metric—in this case, the WaitingTasks metric in the MyAppMetrics namespace—is over the threshold of 10 for a single evaluation period. That is, as soon as CloudWatch notices this value is above 10, it will put this alarm into the Alarm state, triggering the ScaleUpPolicy.

This will cause an additional instance to be launched into the Auto Scaling group. Once this instance begins processing tasks, the WaitingTasks value will drop below 10, which puts the WaitingTasksAlarm in the OK state. This causes the ScaleDownPolicy to be triggered, resulting in a single instance in the scaling group being terminated.

The CustomMetricScalingGroup has its Cooldown parameter set to 300. This value, measured in seconds, controls how frequently Auto Scaling events occur for this group. By setting it to five minutes (300 seconds), we ensure that there is a gap between instances being created and deleted.

To see this in action, create the CloudFormation stack using the template just shown. Because we previously set the value of the WaitingTasks metric to 20, a new instance will be launched. Wait for the two instances to finish launching and then issue the following command:

aws cloudwatch put-metric-data --namespace "MyAppMetrics" --metric-name WaitingTasks --value 5 --unit Count

This makes CloudWatch think there are only five messages remaining in the queue, which will trigger the ScaleDownPolicy. After a few moments, one of the instances in the scaling group will be terminated.

In the real world, we would be periodically executing a command that checks the size of the waiting tasks queue and submits the value to CloudWatch, making this an entirely automated process.

If no data is available for the requested metric, Auto Scaling will, by default, take no action. This behavior can be controlled by adding an InsufficientDataActions parameter to the WaitingTasksAlarm resource. For more information on controlling this behavior, see the documentation for the [AWS::CloudWatch::Alarm resource type](http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-cw-alarm.html).

# Old Tools, New Tricks

Many common monitoring tools predate AWS significantly. Nagios, a popular open source monitoring tool, has been around since 1999, seven years before the introduction of EC2. One of the advantages of Nagios (and other mature monitoring tools) is the ecosystem surrounding it, providing features such as graphing, reporting, and integration with third-party services. Using such tools, you can build a replacement for CloudWatch that might be more suited for your specific needs, as well as work around some of the limitations mentioned in the previous section.

###### TIP

The [Nagios Exchange](http://exchange.nagios.org/) contains plug-ins that can be used to integrate Nagios and AWS. In addition to plug-ins that directly monitor the status of your EC2 instances, there are plug-ins that query CloudWatch and other AWS components, allowing you to monitor your AWS infrastructure in a variety of ways. Remember that any tools that pull data from CloudWatch will do so by querying the CloudWatch API. Pulling these metrics too frequently will result in a higher AWS bill.

As Robert Heinlein pointed out in The Moon is a Harsh Mistress, there’s no such thing as a free lunch. The time taken to implement a custom monitoring solution is time taken away from building your core application infrastructure, so this undertaking should be carefully considered against off-the-shelf tools, or indeed simply using CloudWatch and learning to love it despite its limitations.

At a high level, there are two main ways of dynamically configuring tools like Nagios within an AWS infrastructure.

The first is to use a configuration management tool such as Puppet. Puppet provides a feature known as [exported resources](http://docs.puppetlabs.com/puppet/2.7/reference/lang_exported.html), which allows the configuration of one node to influence the configuration of another node. For example, when Puppet runs on one of your web application instances, it can use the data collected on this node to dynamically configure your monitoring instances. If Puppet recognizes that Nginx is running on the web application node, the Nagios instance will be automatically configured to run Nginx-specific checks against the web application node, such as making sure HTTP requests to port 80 respond with an HTTP 200 status code.

This feature relies on [PuppetDB](http://docs.puppetlabs.com/puppetdb/1.5/), which means it will work only when Puppet is running in the traditional master/client mode.

Implementing this system requires two entries in your Puppet manifest files. The first is the declaration stage, where you declare the virtual resource type. This stanza is placed in the Puppet manifest for the host to be monitored. For example, to monitor Nginx as just described, the following virtual resource declaration can be used:

@@nagios\_service { "check\_http${hostname}":

use => 'http-service',

host\_name => "$fqdn",

check\_command => 'check\_http',

service\_description => "check\_http${hostname}",

target => '/etc/nagios/conf.d/dynamic\_${fqdn}.cfg',

notify => Service[$nagios::params::nagios\_service],

}

Notice that the resource type (in this case, nagios\_service) is prefixed with @@. This lets Puppet know that this is a virtual resource that will be realized on another node. Declaring this virtual resource will not cause any changes to be made on the monitored instance itself. Rather, it causes the relevant data to be stored in PuppetDB for later use by another node.

This declaration will configure Nagios to perform a simple HTTP service check, using the check\_http command. The target parameter writes this configuration to a file whose name contains the fully qualified domain name (FQDN) of the monitored instance. If your monitored instance has an FQDN of web01.example.com, this configuration segment would be written to a file named /etc/nagios/conf.d/dynamic\_web01.example.com.cfg on the Nagios host. By default, Nagios will include all .cfg files contained within the /etc/nagios/conf.d directory when the configuration is read. The notify parameter causes Nagios to be restarted whenever this file changes, so that the new monitoring configuration is picked up automatically.

The second component is the collection stage, which affects the Nagios instance. The following line should be added to the node definition for the Nagios instance in your Puppet manifest file:

Nagios\_service <<| |>>

When Puppet is run on the Nagios instance, any previously declared virtual resources describing Nagios resources will be realized on the Nagios instance. This is the point at which the /etc/nagios/conf.d/dynamic\_web01.example.com.cfg file is created, and the Nagios service restarted.

Although support for Nagios is explicitly built into Puppet’s exported resources feature, there is nothing to stop it from being used for other packages. In fact, it can be used to configure any service that relies on text-based configuration files, making it a flexible tool for dynamic configuration.

Another method of achieving this goal is to use a custom script to query the AWS API and write Nagios configuration files based on the retrieved data. This is a good option if you do not have an existing Puppet master/client infrastructure in place. Implementing Puppet merely to take advantage of exported resources could be considered overkill, making the custom script route a more attractive option.

One potential downside of this system relates to instances that get terminated as part of expected Auto Scaling operations. Nagios must be informed that the instance no longer exists and shouldn’t be monitored. For this reason, I typically recommend putting a separate configuration file in the Nagios configuration directory for each node, hence the use of dynamic\_${fqdn).cfg in the example.

Auto Scaling can be configured to send a notification to Amazon’s Simple Notification Service when instances are launched or terminated. Subscribing to these notifications makes it simple to delete all the Nagios configuration files for a particular host after it is terminated. After Nagios is restarted, the host will no longer be monitored, and will not cause any false alarms after Nagios notices that the instance is no longer accessible to its checks.

##### SENSU AND OTHER MONITORING OPTIONS

The preceding section described how to use Nagios to monitor EC2 instances. Of course, Nagios is only one of the wide range of tools that can be used for system monitoring, and I chose it because it still enjoys a high level of popularity among system administrators. This is, in part, due to the number of plug-ins available for Nagios, allowing almost any service or application to be monitored.

Many other highly regarded packages can be used to monitor your services. One example is [Icinga](https://www.icinga.org/), which is a fork of the open source version of Nagios. Icinga aims to provide compatibility with all existing Nagios plug-ins while making enhancements to the web interface and core software.

Another package in this space is [Sensu](http://sensuapp.org/). Sensu has features that make it an excellent choice for monitoring cloud-based infrastructure. Chief among these is the architecture of Sensu itself. Rather than operating in a client/server mode (as do Nagios and Icinga), it uses RabbitMQ, an AMQP-based messaging system. This makes it inherently more scalable than software that relies on a single central master server.

When a service is checked, the Sensu client writes the check result to a RabbitMQ queue, where it is read by the Sensu server process. Decoupling the submission of a check from the reading process in this way enables a much higher throughput than an architecture in which clients submit their check results directly to the master. Because RabbitMQ can operate in a highly available cluster, it is also more resilient to failure. As long as both the master and client processes know the address of the RabbitMQ server, check results can be submitted. If the master server is briefly unavailable, messages will wait in the queue until the master is available again.

When combined with Auto Scaling groups, this configuration of Sensu makes the entire monitoring system more reliable. Should the EC2 instance running the master server be terminated due to failure, a new one can be brought into service, and it will begin processing all of the waiting check results automatically.