AMAZON CLOUDWATCH MONITORING GUIDE

bluematador

aws

How to Use This Ebook

Welcome to a deep dive on Amazon CloudWatch and how to use it to monitor your AWS infrastructure! This book mostly focuses on individual AWS services and how to monitor them, but the first couple of sections are a primer on CloudWatch and CloudWatch alarms. Even if you've used CloudWatch before, we encourage you to read the intro sections, as they are the foundation of the topics discussed throughout the rest of the ebook.

Table of Content

What is CloudWatch?

Amazon CloudWatch monitors your Amazon Web Services (AWS) resources. Essentially, CloudWatch is an archive built to store AWS metrics' time series data. CloudWatch converts raw data feeds into digestible, actionable information. Amazon provides a set of pre-defined metrics in CloudWatch for free. The free tier also lets you view these metrics.

Their paid service allows you to access, graph, create dashboards, and send alerts for all metrics—including your own custom metrics—through the console, command line, or API.

Important CloudWatch Concepts

This section explains some of the basic CloudWatch concepts that will be critical to understand before continuing with the rest of this guide.

Namespaces

A namespace is a container for metrics that belong to a service. For example, all EC2 metrics are grouped into the *AWS/EC2* namespace.

Dimensions

A dimension is a set of metadata key-value pairs that help identify a metric. Examples include identifying an SQS queue with the QueueName dimension, or identifying a Lambda function by FunctionName.

Metrics

Metrics are the core of CloudWatch. Each metric is a time series data set published from an AWS service to CloudWatch. When querying for a metric, you supply a namespace, metric name, and any dimensions necessary to identify the resource that published the metric. CloudWatch then responds with a set of data points that are a timestamp mapped to the value of the metric at that time. Each metric has a period, or an amount of time over which the metric is aggregated.

CloudWatch retains metric data as follows:

- Data points with a period of less than 60 seconds are available for 3 hours
- Data points with a period of 1 minute are available for 15 days
- Data points with a period of 5 minutes are available for 63 days
- Data points with a period of 1 hour are available for 455 days (15 months)

Statistics

Statistics are aggregated values of metrics. When querying CloudWatch, you will have to choose which statistic you would like. CloudWatch supports the following statistics:

- Minimum - the smallest value during the aggregation period
- Maximum - the highest value during the aggregation period
- Average - the average value during the aggregation period
- Sum - the sum of all reported values during the aggregation period
- SampleCount - the number of data points published to CloudWatch during the aggregation period
- pNN.NN - the value of the specified percentile (with up to 2 decimal points of precision), for example p90 is the value of the 90th percentile for the data points within the aggregation period

Not all statistics are supported for every metric.

Period

When specifying the statistic, you will also have to choose the aggregation period, which will usually be the metric granularity provided by the service publishing data to CloudWatch. Valid values for period are 1, 5, 10, 30, and any multiple of 60 up to 86,400 seconds (1 day). The default value is 60.

Alarms

CloudWatch allows you to create an alarm when the value of a metric's statistic crosses a threshold. These alarms can then notify your on-call team via Amazon SNS.

CloudWatch alarms allow you to get notified of events in your AWS infrastructure by monitoring a particular metric for when it crosses a configured threshold. CloudWatch alarms are AWS's built-in way to alert you when your infrastructure is unhealthy.

How to Approximate Anomaly Detection in CloudWatch

Infrastructure resources differ from each other based on their purpose. For example, one ELB may have different load than another, making it difficult to find a single threshold to alert on for all ELBs. This problem is further compounded as they also fluctuate in their usage, which means a threshold that might be healthy on a Monday morning could be unhealthy on a Saturday evening. To combat this issue, many monitoring tools use anomaly detection to find what looks healthy for your resource and then alert only when it deviates from this norm.

However, CloudWatch doesn't support true anomaly detection, so you'll have to use averages to find issues. To do so, you'll want to look at a week of data for a metric and then determine the average high and low. Create a 10% window above the average high and below the average low and you have a good threshold for what looks healthy for your application. It should be noted that this method will not account for any time based variation or seasonality in data.

CloudWatch also supports the **Percentile** statistic on some metrics, but we don't use this for a number of reasons:

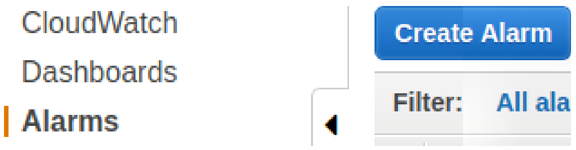
1. It isn't supported across all services, so it's difficult to build a monitoring strategy around it.
2. It works by finding the percentile within the values inside of a metric's period. So it would find the nth percentile across a minute of metrics, or five minutes of metrics, which can be spikey.

For these reasons, we find it more effective to use the method described above to create thresholds for CloudWatch alarms. Throughout the rest of this ebook, when we refer to the "approximate anomaly detection method," we mean this method.

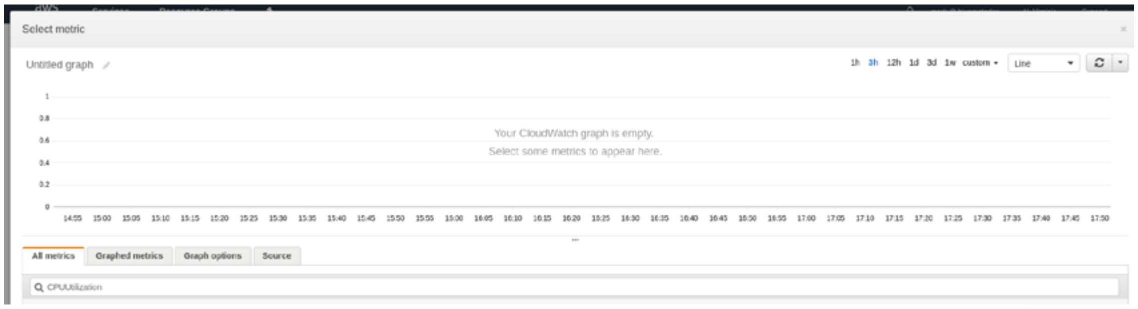
How to Create a CloudWatch Alarm

CloudWatch alarms are created from the CloudWatch console. To do so:

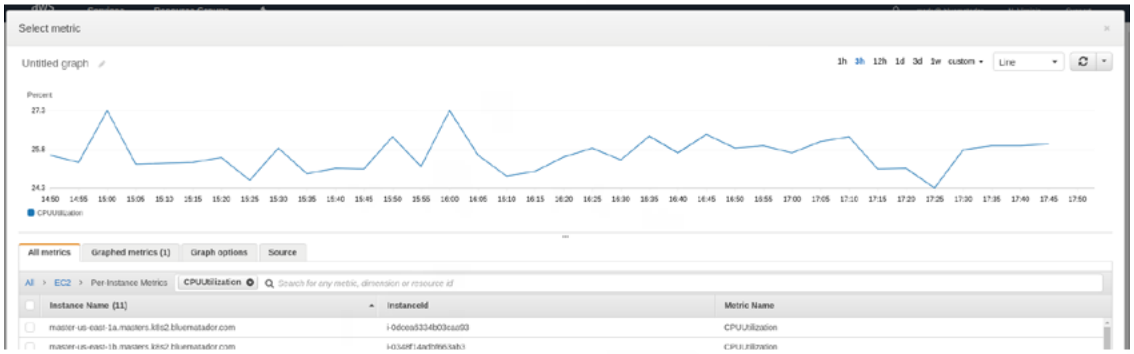
1. Navigate to the CloudWatch console.
2. Click on **Create Alarm**.



3. Click on **Select Metric** and type the name of the metric you'd like to monitor into the search box.

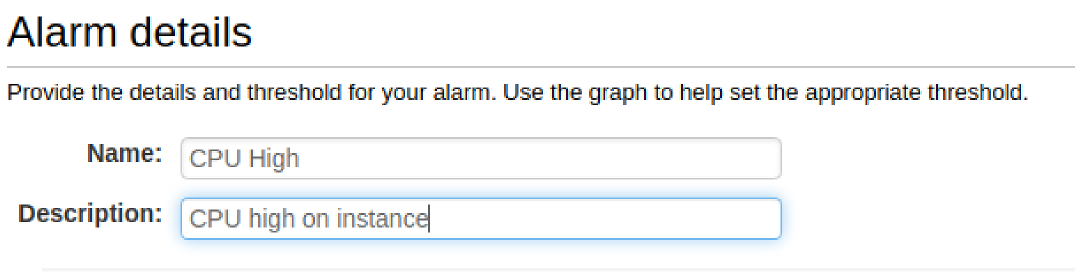

Select metric
Untitled graph
1h 3h 12h 1d 3d 1w custom
Line
Your CloudWatch graph is empty.
Select some metrics to appear here.
All metrics
Graphed metrics
Graph options
Source
CPUUtilization

4. Choose the metric for the resource you're going to monitor. This will plot the selected metric on the graph. CloudWatch only allows you to create an alarm on a single metric. However, if you want to add another metric to use for reference, you can click on the **All Metrics** tab and follow **Step 3** again to find another metric, or use the bread crumbs to move up a category. In the next section, we'll discuss how to combine multiple metrics into a single value through **Metric Math Expressions**. For now, look at a 1 week graph to determine what threshold to use. Click **Select Metric**.



Instance Name (11)	InstanceId	Metric Name
master-us-east-1a.masters.k8s2.bluematador.com	i-0dcea5334b03caa93	CPUUtilization
master-us-east-1b.masters.k8s2.bluematador.com	i-0348f14adb863ab3	CPUUtilization

5. Name and describe your alarm. Choose something that will help you know what's going on if you were to receive the alarm without any context.

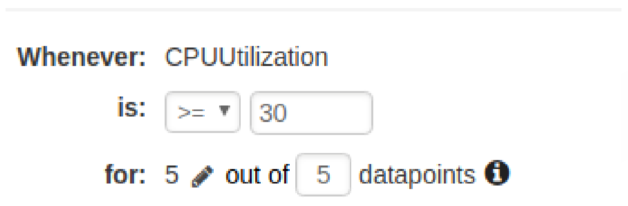

Alarm details

Provide the details and threshold for your alarm. Use the graph to help set the appropriate threshold.

Name: CPU High

Description: CPU high on instance

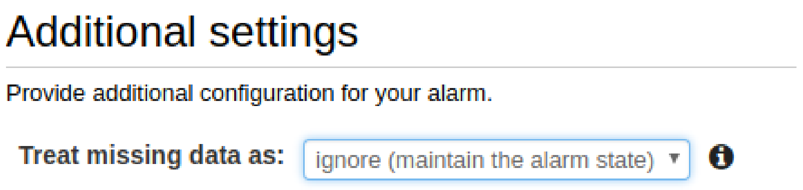
6. Configure your thresholds. You can use the threshold you picked at the end of **Step 4**. You will also need to pick how many data points need to cross the threshold before alerting. If the metric you'd like to monitor is reported on a 1 minute period, choose 3 or 4 out of 5 data points. This means it will take up to 5 minutes for the alarm to trigger, but also helps you avoid false positives or spikes that lead to alert fatigue. If you find that the alert still triggers too frequently, try raising the threshold or requiring more unhealthy data points (i.e. 5/5 or 8/10 data points). With metrics that have a 5 minute period, you will need to balance how long you can wait to be alerted with how sensitive you want the alarm to be, but a good starting point is to require 2 out of 5 data points to be unhealthy.


Whenever: CPUUtilization

is: >= 30

for: 5 out of 5 datapoints

7. Choose what to do when data is missing. You can choose to treat missing data as follows:
    a. Good (resolve any existing alarm)
    b. Bad (create an alarm)
    c. Missing (get more data points and evaluate the most recent existing data)
    d. Ignore (do not change alarm state)

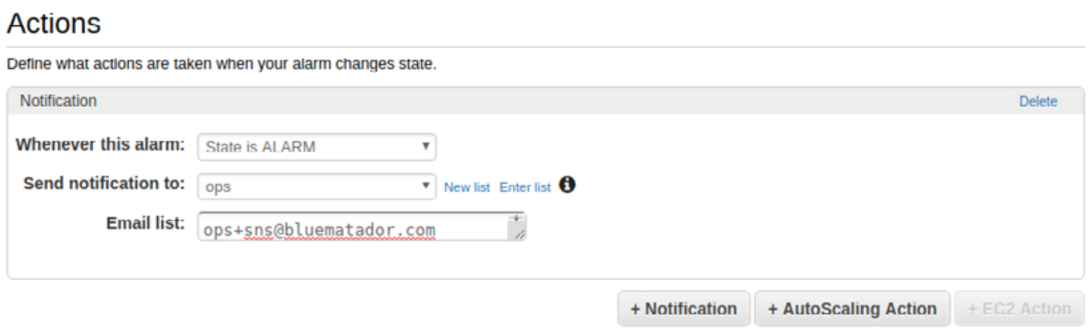

Additional settings

Provide additional configuration for your alarm.

Treat missing data as: ignore (maintain the alarm state)

The best option depends on the metric, but in general, we advise choosing *Ignore* and maintaining alarm states.

8. Set up notifications for the alarm. Notifications can be sent through SNS, and from there, to any number of services


Actions

Define what actions are taken when your alarm changes state.

Notification

Whenever this alarm: State is ALARM

Send notification to: ops

Email list: ops+sns@bluematador.com

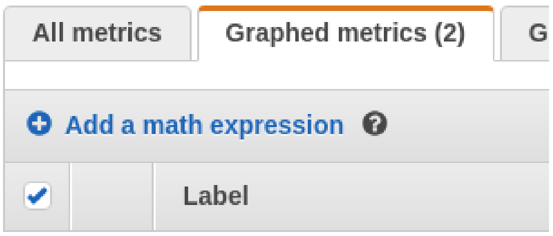
9. Click **Create Alarm**.

At this point, your CloudWatch alarm is now active. Unfortunately, since CloudWatch only supports creating an alarm on a single metric and threshold, you'll have to repeat the whole process for each resource and metric combination. Additionally, if you want to set an alarm on both an upper and lower bound for a metric, you'll have to create two alarms for the metric.

How to Create Alarms on Computed Values

Sometimes you will want to monitor a value that is not directly reported by CloudWatch. It might be the case that multiple metrics can be combined to produce the necessary value. For these situations, you can use **Metric Math Expressions** in your CloudWatch alarms. To create a **Metric Math Expression** do the following:

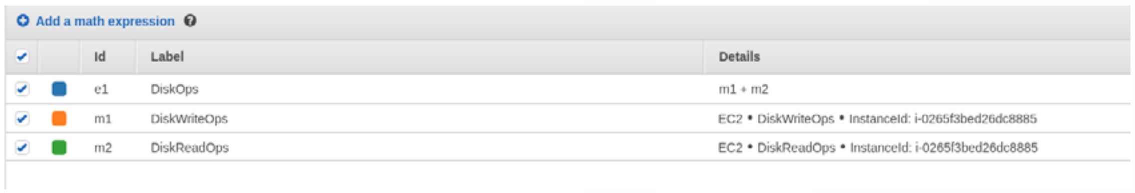
1. Choose the metrics you need in the **All Metrics** tab.
2. Navigate to the **Graphed Metrics** tab.
3. Click on the link that says **Add a math expression**.



CloudWatch will then add an additional value to your **Graphed Metrics** tab. To rename the expression so you have a more readable name in the graph legend, double click on its label and type in the desired name.

Next, edit the math expression in the **Details** column. For the most part, the math expression language is pretty straight forward. You perform arithmetic on the metrics you've selected by referencing their value in the **Id** column of the table. For example, the expression `m1 * 5` would result in a graph of the original **m1** metric with each data point value multiplied by 5. Things get more interesting when you start combining metrics:

In this example, we add the *DiskReadOps* and *DiskWriteOps* metrics for an EC2 instance together to get a timeseries view of the total number of disk operations. CloudWatch comes with a big list of functions you can use to experiment with metric math.


Add a math expression

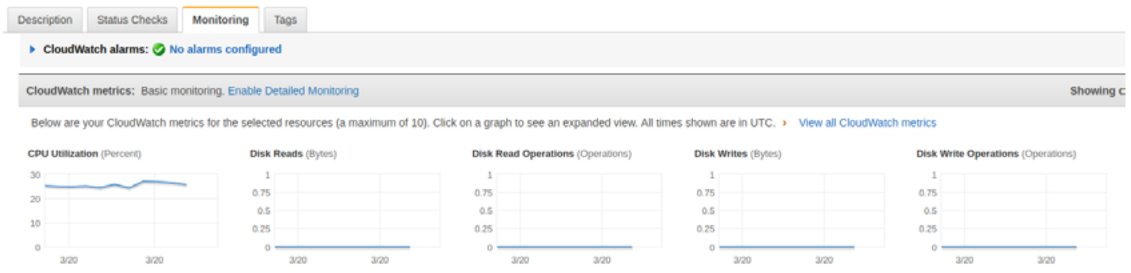
✓		Id	Label	Details
✓	🟦	e1	DiskOps	m1 + m2
✓	🟧	m1	DiskWriteOps	EC2 • DiskWriteOps • InstanceId: i-0265f3bed26dc8885
✓	🟩	m2	DiskReadOps	EC2 • DiskReadOps • InstanceId: i-0265f3bed26dc8885

CloudWatch for EC2

Amazon Elastic Cloud Compute (EC2) allows you to spin up servers for your application without having to actually manage physical hardware. In this section, we'll explain how to use CloudWatch to monitor EC2 and which metrics are important to watch.

How to View CloudWatch Metrics for EC2

CloudWatch metrics for EC2 can be viewed through the **Metrics** portion of CloudWatch, but it is also possible to use the **Monitoring** tab in the Instances section of the EC2 console. This tab shows several metric graphs for each instance.



Description
Status Checks
Monitoring
Tags
CPU Utilization (Percent)
Disk Reads (Bytes)
Disk Read Operations (Operations)
Disk Writes (Bytes)
Disk Write Operations (Operations)
3/20

CloudWatch alarms: ✓ No alarms configured

CloudWatch metrics: Basic monitoring. Enable Detailed Monitoring

Showing c

Below are your CloudWatch metrics for the selected resources (a maximum of 10). Click on a graph to see an expanded view. All times shown are in UTC. › View all CloudWatch metrics

Metrics to Monitor

A Note on Metric Granularity

By default, EC2 reports metrics to CloudWatch in 5 minute intervals. However, if you enable enhanced monitoring for an instance, you'll be able to get metrics in 1 minute intervals (though there is an additional cost). If your application has higher performance requirements, it may be worth the money to get more data so you can react more quickly to changes in your application.

NetworkIn, NetworkOut, NetworkPacketsIn, & NetworkPacketsOut

A sharp increase or decrease in your instance's network traffic is typically a good indication that the instance is unhealthy or about to become unhealthy. It's one of the best ways to detect abnormal application behavior. The *NetworkIn* and *NetworkOut* metrics measure network traffic in bytes, while *NetworkPacketsIn* and *NetworkPacketsOut* measure traffic in number of packets. You should create CloudWatch alarms for each of these metrics using the approximate anomaly detection method with the **Average** statistic, looking for at least 2 data points of anomalous values (unless you've enabled enhanced metrics, in which case, you can look for 5 data points).

CPUCreditBalance

Some EC2 instance types support CPU Credit Balance, which allows them to temporarily burst above the baseline CPU threshold for the instance type. The *CPUCreditBalance* metric measures the current CPU Credit Balance for an instance. This metric is a good one to monitor because when the credit balance is depleted, it is a sure sign that your instance is using too much CPU. To monitor it, create a CloudWatch alarm for when the **Average** statistic goes below 25% of the instance type's maximum CPU Credit Balance.

CPUUtilization

In CloudWatch, *CPUUtilization* measures the percent of total CPU being used by the instance. While anomalous *CPUUtilization* can signal issues in your instance, the metric also has a tendency to fluctuate and should be used more for correlation than detection of critical issues (*CPUCreditBalance* is a good way to monitor CPU issues for instance types with CPU credits). However, it can still be useful to create a CloudWatch alarm on the **Average** statistic using the approximate anomaly detection method and have notifications sent to a lower priority notification method.

StatusCheckFailed

The *StatusCheckFailed* metric measures if your instance has failed its Instance or System status checks in the last minute (it is available at minutely granularity even if enhanced monitoring is not enabled on the instance). If your instance is failing its status checks for more than one datapoint, it needs to be troubleshooted immediately. To monitor *SystemCheckFailed*, create a CloudWatch alarm for the **Sum** statistic for values greater than 0 for 2 data points.

DiskReadOps, DiskWriteOps, DiskReadBytes, & DiskWriteBytes

Like network traffic, disk IO is a good indication of the health of your instance. Unfortunately, EC2 measures disk metrics only for instance types that have instance storage. If your instance type has instance storage (in the table it will list the amount, and not just "EBS only"), you can follow the approximate anomaly detection method to create CloudWatch alarms on the **Average** statistic of each metric. Otherwise, use the metrics in the EBS section of this book to monitor the attached volumes.

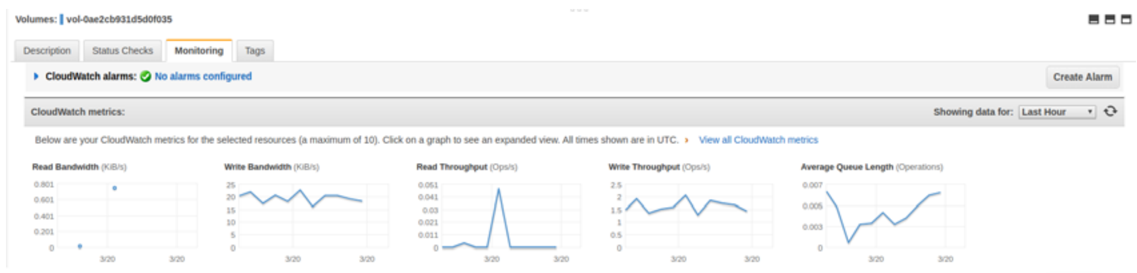
Keep in mind that these metrics only measure disk performance for instance storage, so even if your instance has instance storage, you will still have to monitor any additional EBS volumes you've attached to the instance. You will also need to use another solution to monitor actual disk space used on your file systems, as this metric is not available in CloudWatch.

CloudWatch for EBS

Amazon Elastic Block Store (EBS) allows you to provision storage volumes for your EC2 instances without having to actually manage physical drives. In this section, we'll explain how to use CloudWatch to monitor EBS and which metrics are important to watch. EBS reports metrics at 5 minute granularity.

How to View CloudWatch Metrics for EBS

CloudWatch metrics for EBS can be viewed through the **Metrics** portion of CloudWatch, but it is also possible to use the **Monitoring** tab in the Volumes section of the EC2 console. This tab shows several metric graphs for each volume.



Volumes: vol-0ae2cb931d5d0f035

CloudWatch alarms: No alarms configured

CloudWatch metrics:

Showing data for: Last Hour

Below are your CloudWatch metrics for the selected resources (a maximum of 10). Click on a graph to see an expanded view. All times shown are in UTC. › View all CloudWatch metrics

Metrics to Watch

A Note about Volume Types

EBS has the following volume types:

- General Purpose SSD (gp2)
- Provisioned IOPS SSD (io1)
- Throughput Optimized HDD (st1)
- Cold HDD (sc1)

Some metrics are not available based on volume type. We will indicate that this is the case if applicable in each metric's section.

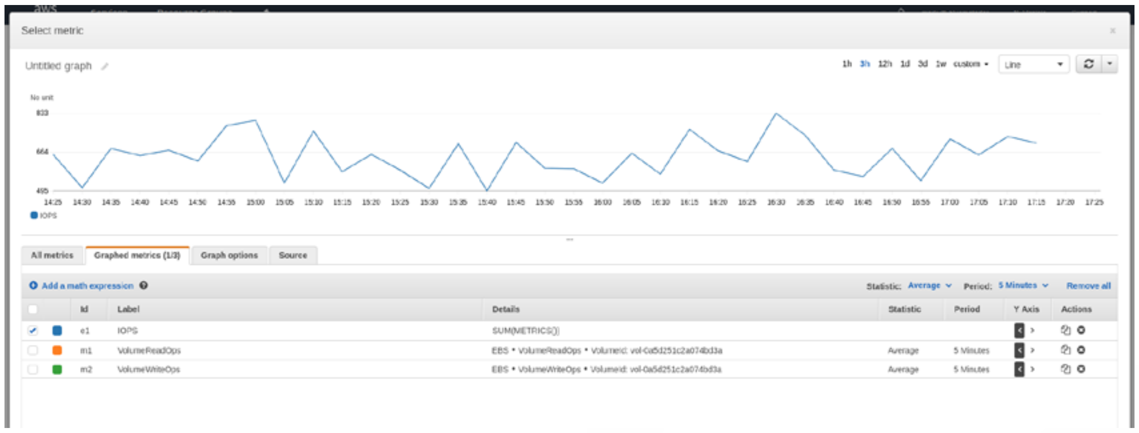
VolumeReadOps & VolumeWriteOps

VolumeReadOps and *VolumeWriteOps* measure the number of read and write operations on a volume. In most cases, it's probably not valuable to monitor these metrics for anomalies, as disk reads can be a little spikey. However, if the volume is attached to a cache, disk access should be infrequent, so you should create a CloudWatch alarm on the **Sum** statistic using the approximate anomaly detection method.

Otherwise, you'll want to create a CloudWatch alarm to alert you if you are approaching the IOPS limit for your volume type. To do so, first calculate the IOPS limit for your volume based on volume type:

- io1: the amount you've set
- gp2: the minimum of 3 * volume capacity (in GB) and 16,000
- sc1: 300
- st1: 500

Next, when choosing the metric for your alarm, create a **Metric Math Expression** that sums the *VolumeReadOps* and *VolumeWriteOps* and then select only the result. Then set your threshold for when that combination is greater than 90% of your limit.



Statistic: Average Period: 5 Minutes

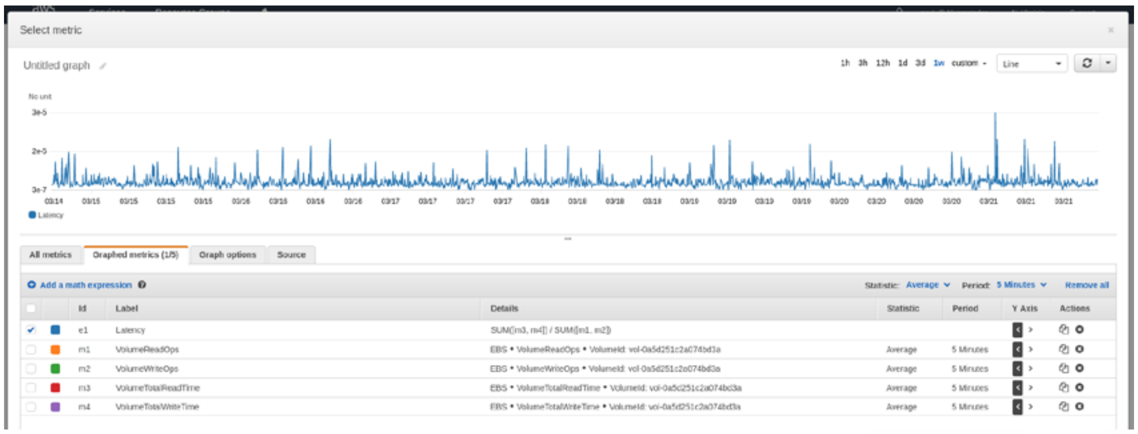
		Id	Label	Details	Statistic	Period
✓	🟦	e1	IOPS	SUM(METRICS())		
	🟧	m1	VolumeReadOps	EBS • VolumeReadOps • VolumeId: vol-0a5d251c2a074bd3a	Average	5 Minutes
	🟩	m2	VolumeWriteOps	EBS • VolumeWriteOps • VolumeId: vol-0a5d251c2a074bd3a	Average	5 Minutes

VolumeTotalReadTime & VolumeTotalWriteTime

VolumeTotalReadTime and *VolumeTotalWriteTime* measure the total amount of time spent in read or write operations on the volume. By themselves, they are not particularly valuable, but can be used to calculate the disk latency for your volume. Latency is calculated with the following formula:

(*VolumeTotalReadTime* + *VolumeTotalWriteTime*) / (*VolumeReadOps* + *VolumeWriteOps*)

To monitor latency, you should create a CloudWatch alarm by using the approximate anomaly detection method on your calculated latency metric, using the **Sum** statistic for all metrics in the formula.



Statistic: Average Period: 5 Minutes

		Id	Label	Details	Statistic	Period	Y Axis	Actions
✓	🟦	e1	Latency	SUM([m3, m4]) / SUM([m1, m2])				
	🟧	m1	VolumeReadOps	EBS • VolumeReadOps • VolumeId: vol-0a5d251c2a074bd3a	Average	5 Minutes		
	🟩	m2	VolumeWriteOps	EBS • VolumeWriteOps • VolumeId: vol-0a5d251c2a074bd3a	Average	5 Minutes		
	🟥	m3	VolumeTotalReadTime	EBS • VolumeTotalReadTime • VolumeId: vol-0a5d251c2a074bd3a	Average	5 Minutes		
	🟪	m4	VolumeTotalWriteTime	EBS • VolumeTotalWriteTime • VolumeId: vol-0a5d251c2a074bd3a	Average	5 Minutes		

VolumeQueueLength

VolumeQueueLength measures the number of disk operations queued. When this metric spikes, access to your disk will slow and your application performance may suffer. However, monitoring your volume for a queue size greater than 0 is not desirable, as it suggests you expect your volume to sit idle all of the time. Instead, create a CloudWatch alarm on *VolumeQueueLength* with the **Average** statistic using the approximate anomaly detection method.

VolumeThroughputPercentage

VolumeThroughputPercentage measures the percent of the provisioned IOPS for your volume that your volume is actually getting. It does *not* measure the amount of IOPS actually being consumed on the volume. AWS expects volumes to be within 10% of their provisioned limit for 99.9% of the year, but it can be helpful to monitor this metric for correlation with other issues you are seeing in your application. To do so, create a CloudWatch alarm on *VolumeThroughputPercentage* on the **Average** statistic for values less than 90%. Send this notification to a lower priority notification method.

This metric only applies to io1 volume types.

BurstBalance

Because gp2 volumes don't have provisioned IOPS, they have a Burst Balance that allows them to temporarily perform more operations. The *BurstBalance* metric measures the remaining percentage of the Burst Balance for your volume. Consistently using the Burst Balance is a sign that you need to upgrade your volume and you should monitor for this condition. Create a CloudWatch alarm on *BurstBalance* for the **Average** statistic for values less than 25% that happen for more than 3 data points (5 minutes each).

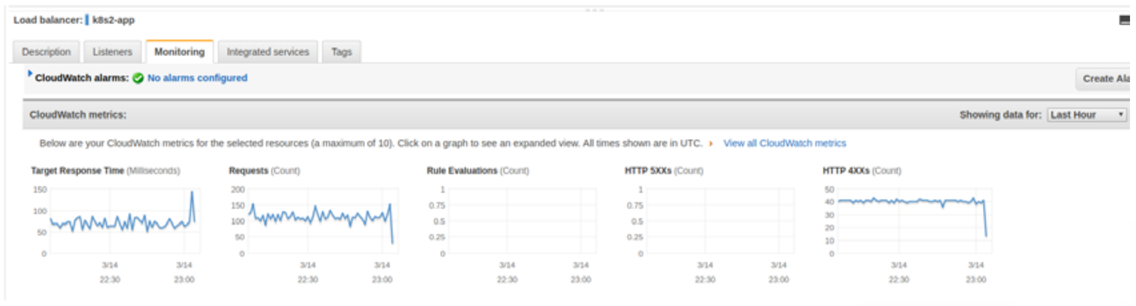
This metric applies to gp2, st1, and sc1 volume types.

CloudWatch for ELB

Amazon Elastic Load Balancing (ELB) allows you to create load balancers for your application without having to actually manage the servers that do the load balancing. In this section, we'll explain how to use CloudWatch to monitor Elastic Load Balancing and what metrics are important to watch. ELB metrics are reported on 1 minute intervals.

How to View CloudWatch Metrics for Elastic Load Balancing

CloudWatch metrics for Elastic Load Balancing can be viewed through the **Metrics** portion of CloudWatch, but it is also possible to use the **Monitoring** tab in the Load Balancer section of the EC2 console. This tab shows several metric graphs for each ELB.


Load balancer: k8s2-app
Description
Listeners
Monitoring
Integrated services
Tags
CloudWatch alarms: No alarms configured
Create Ala
CloudWatch metrics:
Showing data for: Last Hour
Below are your CloudWatch metrics for the selected resources (a maximum of 10). Click on a graph to see an expanded view. All times shown are in UTC.
View all CloudWatch metrics
Target Response Time (Milliseconds)
Requests (Count)
Rule Evaluations (Count)
HTTP 5XXs (Count)
HTTP 4XXs (Count)
3/14 22:30
3/14 23:00

Metrics to Watch

A Note about Load Balancer Types

Elastic Load Balancing has the following load balancer types:

- Classic Load Balancer
- Application Load Balancer
- Network Load Balancer

Some metrics are not available based on load balancer type, or may have different names. We will indicate which types are applicable in the headers of each metric.

UnHealthyHostCount (Classic, Application, and Network)

One of the most obvious metrics to monitor is the number of hosts in a load balancer that are failing their health checks. If an instance is failing health checks it is unable to serve traffic.

To monitor *UnHealthyHostCount*, you should create a CloudWatch alarm on the **Average** statistic. However, the threshold depends on the number of hosts in your load balancer. If you have 5 or less, you should probably be alerted whenever *UnHealthyHostCount* is nonzero. Otherwise, you will have to determine what is reasonable for your application, but a good rule of thumb is no more than 20% of your hosts should be unhealthy. Unfortunately, this is further complicated if you have set up autoscaling, as CloudWatch has no notion of time based alarms. You will just have to use the scaled down threshold.

If you're using application load balancers, keep in mind that CloudWatch reports the metric for each load balancer/target group combination, so you'll have to create a CloudWatch alarm for each target group.

RequestCount (Classic and Application)

RequestCount measures the number of requests made to the ELB. A surge or drop the number of requests could signal an issue in clients that are calling your load balancer. It could also mean your ELB is returning many errors and clients are retrying.

To monitor *RequestCount*, you'll be looking for anomalies. Create a CloudWatch alarm on the **Sum** statistic using the approximate anomaly detection method for whenever *RequestCount* exceeds the determined threshold for more than 5 data points.

HTTPCode_Backend_5XX (Classic) & HTTPCode_Target_5XX_Count (Application)

Both Classic Load Balancers and Application Load Balancers have a metric for 5xx errors returned by the hosts behind the load balancers. If you see a sudden spike in this metric, it is a clear indication that something is wrong in your application.

However, because some number of 5xx errors are expected in any distributed system, you'll need to use the approximate anomaly detection method to detect issues in your system. Again, you'll use the **Sum** statistic to create the CloudWatch alarm.

HTTPCode_Backend_4XX (Classic) & HTTPCode_Target_4XX_Count (Application)

Both Classic Load Balancers and Application Load Balancers have a metric for 4xx errors returned by the hosts behind the load balancers. If you see a sudden spike in this metric, you likely have an issue in clients making requests to your load balancer.

However, because some number of 4xx errors are expected in any distributed system, you'll need to use the approximate anomaly detection method to detect issues in your system. Again, you'll use the **Sum** statistic to create the CloudWatch alarm.

Latency (Classic) & TargetResponseTime (Application)

CloudWatch measures the amount of time it takes for your hosts to return a response through the load balancer with the *Latency* or *TargetResponseTime* metrics. If response time increases drastically, it almost certainly means there are issues in your application. These types of errors are especially important to detect because they cascade through services as your clients spend longer waiting on resources they request from the load balancer.

Once again, anomaly detection is the best way to monitor this metric. Use the approximate anomaly detection method to detect spikes in latency. This time, use the **Average** statistic for CloudWatch alarm creation.

BackendConnectionErrors (Classic) & TargetConnectionErrorCount (Application)

The *BackendConnectionErrors* metric is incremented whenever your ELB is unable to connect to the hosts backing the load balancer. If this is happening consistently, your hosts are likely overloaded and unable to accept connections, or traffic may be routed to a port that is not open.
While random connection errors can occur, if this value is consistently nonzero, you should know about it. Create a CloudWatch alarm on the **Sum** statistic when this metric is nonzero for 5 consecutive data points.

SurgeQueueLength (Classic)

The surge queue length is the number of pending requests to a healthy instance in a classic load balancer. When the number of requests exceeds the maximum of 1,024, they will be rejected. As such, it's a good idea to keep an eye on your surge queue length and ensure that you have enough instances backing your load balancer to handle your load.

To monitor *SurgeQueueLength*, create a CloudWatch alarm on the **Maximum** statistic to alert you when *SurgeQueueLength* exceeds 768 (75% of the max) for 5 consecutive data points. If your application is very performance constrained, you'll probably want to have a stricter alarm, either by lowering the threshold or by checking for nonzero values over a longer timespan like 15 minutes.

ProcessedBytes (Application and Network)

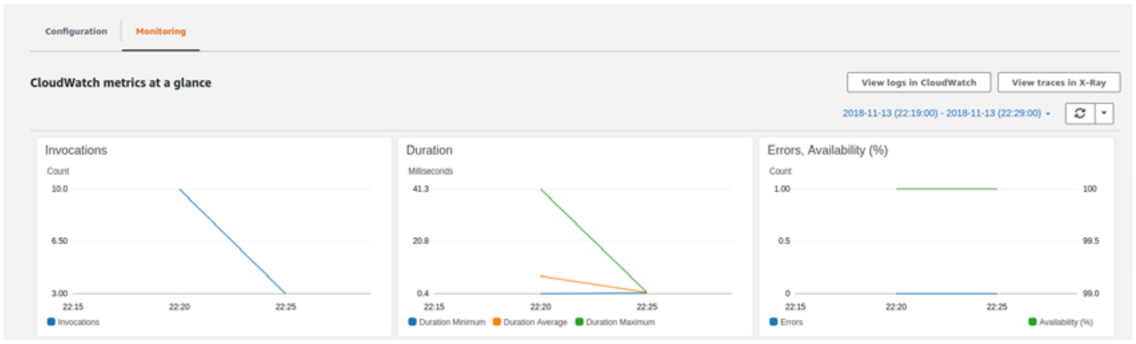
ProcessedBytes measures the number of bytes processed by the ELB. An anomalous amount can signal issues in your application, but is best used in correlation with other metrics. As such, you probably shouldn't send alarms for this metric to your on-call team. However, it can be useful to still create a CloudWatch alarm for the **Sum** statistic using the approximate anomaly detection method and have the notifications sent to an informational notification method.

CloudWatch for Lambda

Amazon Lambda is a serverless execution environment that allows you to run code based on triggers. It integrates with a large number of other AWS services. CloudWatch Metrics and CloudWatch Logs are most useful for monitoring the service. We'll explain how to use them and what metrics are important to watch. Lambda metrics are reported on 1 minute intervals.

How to View CloudWatch Metrics for Lambda

CloudWatch metrics for Lambda can be viewed through the **Metrics** portion of CloudWatch, but it is also possible to use the **Monitoring** tab in the Lambda function UI. This tab shows several metric graphs for each function.



How to Use CloudWatch Logs with Lambda

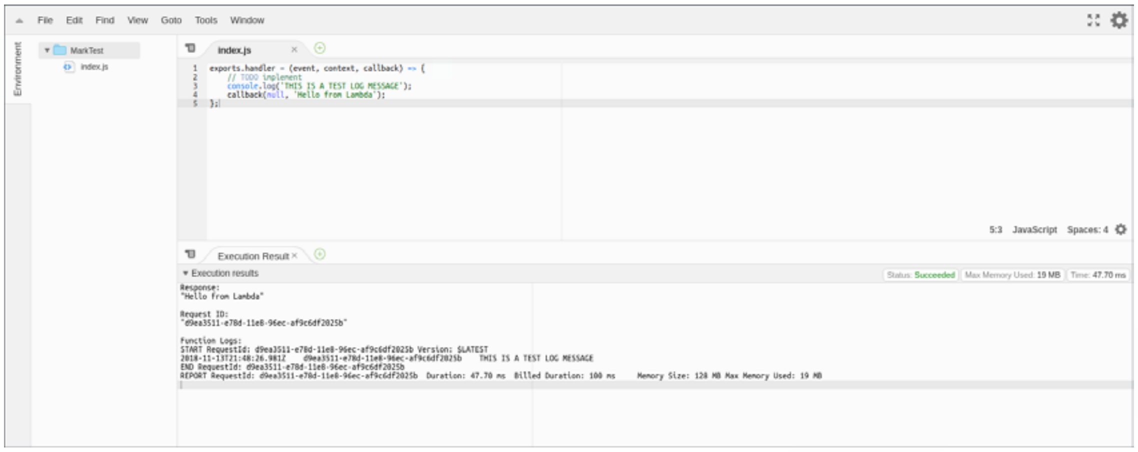
When you spot an anomaly in a metric, it may not always be immediately apparent what caused the problem. When this is the case, logging is helpful to gain insight on the application level of what might be the root cause. You should add log statements to your function that will help you find where things are going wrong.

For this purpose, CloudWatch Logs allow you to emit log messages in your functions that are then stored in CloudWatch.

Each of the programming languages supported by Lambda have some sort of logging functionality, whether that's Node.js's `console.log` or Java's `LambdaLogger.log` . Here is a list of documentation on how to send logs in each of the supported languages:

- Node.js
- Python
- Java
- Go
- C#
- Powershell

Once you've started writing log messages in your function, there are a number of ways to check them. The easiest is probably just to run your function from the AWS web console. Log messages are printed in the execution results.



```
exports.handler = (event, context, callback) => {
    // TODO implement
    console.log('THIS IS A TEST LOG MESSAGE');
    callback(null, 'Hello from Lambda');
};
```

```
Response:
"Hello from Lambda"

Request ID:
"d9ea3511-e78d-11e8-96ec-af9c6df2025b"

Function Logs:
START RequestId: d9ea3511-e78d-11e8-96ec-af9c6df2025b Version: $LATEST
2018-11-13T21:48:26.981Z    d9ea3511-e78d-11e8-96ec-af9c6df2025b    THIS IS A TEST LOG MESSAGE
END RequestId: d9ea3511-e78d-11e8-96ec-af9c6df2025b
REPORT RequestId: d9ea3511-e78d-11e8-96ec-af9c6df2025b  Duration: 47.70 ms  Billed Duration: 100 ms     Memory Size: 128 MB Max Memory Used: 19 MB
```

You can also view all logs for your function in CloudWatch. Select **Logs** out of the menu on the left and then choose the Log Group that contains the name of your function. Within the log group, you'll be able to access log messages emitted by your function.

Metrics to Watch

Invocations

The *Invocations* metric measures the number of times your function is invoked. This metric is particularly important to watch as it has a direct effect on your AWS costs. It's not uncommon to hear about Lambda users who run up big bills when a bug causes their invocations to get out of control. Therefore, it's important to be watching this metric for spikes so you can shut it down before things get out of hand. Set a CloudWatch alarm on the **Sum** statistic using the approximate anomaly detection method.

Errors

The *Errors* metric is incremented any time your function fails to run successfully. Obviously, it's important to know if your function is failing due to errors. To calculate the error rate of your function, divide *Errors* by *Invocation*.

The most useful statistic for this metric is **Sum**. You might be tempted to set your CloudWatch alarm to alert when there are any errors at all (ie, setting the threshold to 0), but it's often the case that with distributed systems a small number of errors is normal. Use the approximate anomaly detection method to determine a good threshold.

If you want to monitor the error rate, you can use a **Metric Math Expression**. To do so, add both *Errors* and *Invocations* to your CloudWatch alarm, and create a **Metric Math Expression** that divides *Errors* by *Invocations*. Name that expression *Error Rate* and select that metric.

Duration

Duration measures how long it takes to run your function in milliseconds. Functions can timeout, and the longer your functions run, the more you'll be charged, so you'll want this metric to be as low as possible. To set a CloudWatch alarm for this metric, use the approximate anomaly detection method on the **Average** statistic. For functions with a relatively low timeout, you may also want to monitor when the average *Duration* gets within 90% of the timeout.

Throttles

Finally, the *Throttles* metric measures the number of times your function is throttled. This happens when you hit the concurrency limit for your account. If you're having issues with throttling, check out our blog post on the subject. It contains all the information you'll need to fix the problem.

The most useful statistic for this metric is **Sum**. Set a CloudWatch alarm for any time this value is above 0. If your Lambda function is being throttled, you should take action to avoid using up your concurrency allocation.

CloudWatch for DynamoDB

Amazon DynamoDB is a key-value and document database that allows you to easily scale to huge numbers of records with single digit millisecond performance. In this section, we'll explain how to use CloudWatch to monitor DynamoDB and what is important to watch.

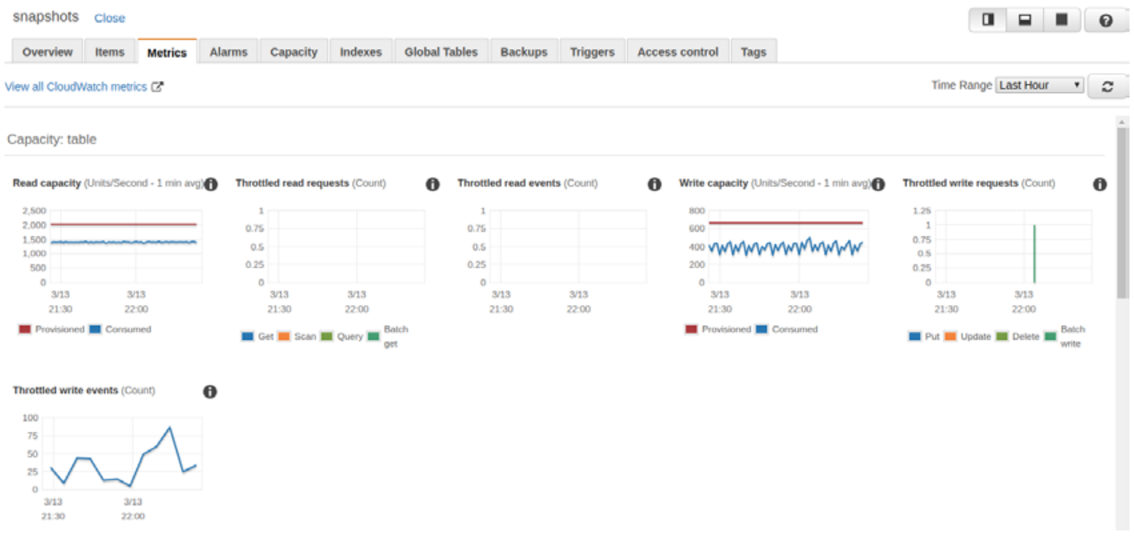
CloudWatch aggregates the following DynamoDB metrics at1 minute intervals:

- ConditionalCheckFailedRequests
- ConsumedReadCapacityUnits
- ConsumedWriteCapacityUnits
- ReadThrottleEvents
- ReturnedBytes
- ReturnedItemCount
- ReturnedRecordsCount
- SuccessfulRequestLatency
- SystemErrors
- TimeToLiveDeletedItemCount
- ThrottledRequests
- UserErrors
- WriteThrottleEvents

For all other DynamoDB metrics, the aggregation granularity is 5 minutes.

How to View CloudWatch Metrics for DynamoDB

CloudWatch metrics for DynamoDB can be viewed through the **Metrics** portion of CloudWatch, but it is also possible to use the **Metrics** tab in the DynamoDB Table console. This tab shows several metric graphs for each table.


snapshots
Close
Overview
Items
Metrics
Alarms
Capacity
Indexes
Global Tables
Backups
Triggers
Access control
Tags
View all CloudWatch metrics
Time Range
Last Hour
Capacity: table
Read capacity (Units/Second - 1 min avg)
Throttled read requests (Count)
Throttled read events (Count)
Write capacity (Units/Second - 1 min avg)
Throttled write requests (Count)
Throttled write events (Count)
Provisioned
Consumed
Get
Scan
Query
Batch get
Put
Update
Delete
Batch write

Metrics to Watch

UserErrors

The *UserErrors* metric is incremented each time DynamoDB responds with a 400 HTTP error. Causes for this include invalid query parameters, trying to access a table or index that does not exist, or permissions errors. As such, any nonzero value for this metric represents an actionable issue. When you detect *UserErrors* are nonzero, check for any code releases or config changes that might have broken your queries.

To monitor *UserErrors*, create a CloudWatch Alarm to alert you whenever the **Sum** statistic of this metric is greater than 0. Be aware that this metric is found under the **Account Metrics** section of DynamoDB metrics as they are not associated with a particular table.

SystemErrors

The *SystemErrors* metric is incremented each time DynamoDB responds with a 500 HTTP error. This metric means that the DynamoDB service is experiencing internal errors, which can be correlated with issues you are seeing in the rest of your application. Your application should be designed to retry requests to DynamoDB with exponential backoff to handle this situation.

When monitoring *SystemErrors*, you should create a CloudWatch alarm whenever *SystemErrors* is nonzero, but these alarms should be more informational, and not sent to your on-call team. That's because consistent *SystemErrors* are useful to know about so you can handle the results in the rest of your application, but are not directly actionable. Instead, your application should be designed to retry calls to DynamoDB. Be aware that this metric is found under the **Account Metrics** section of DynamoDB metrics as they are not associated with a particular table.

ConsumedReadCapacityUnits & ConsumedWriteCapacityUnits

DynamoDB tables can be configured with a provisioned amount of Read Capacity Units (RCU) and Write Capacity Units (WCU) that are consumed whenever you read from or write to your table. When you consistently exceed either amount, your request will be throttled. As such, it's important to monitor the amount of RCU and WCU you are using to minimize throttling.

To do so, set CloudWatch alarms for *ConsumedReadCapacityUnits* and *ConsumedWriteCapacityUnits* when their **Average** statistic is greater than than 80% of your provisioned limit for more than 10 data points. This will allow you to determine that you are consistently close to the limit and give you time to provision more RCU or WCU as needed. Additionally, CloudWatch is unable to detect when you add more provisioned RCUs or WCUs, so you'll have to update your alarms if you ever change your capacity.

If you have set up Auto Scaling with DynamoDB, you will have already created CloudWatch alarms that result in scaling RCU or WCU, but may want to create additional alarms to detect when your DynamoDB tables are scaled to the highest amount you've set with Auto Scaling.

ThrottledRequests

ThrottledRequests is incremented any time any part of a request to a table is throttled. This metric is a good catchall for throttling, as it encompasses both reads and writes. This metric is particularly important to monitor, as any throttle potentially represents a failure in your application or an inability to save data. For a detailed explanation of throttling in DynamoDB and how to troubleshoot it, refer to our guide on the subject.

Despite its importance, monitoring *ThrottledRequests* is not as simple as creating an alert for any nonzero value. Because of the way DynamoDB partitions your data, a small number of throttle events (where part of a batch request fails, but not the whole request) are normal, and your application should be able to simply retry the events that failed.

Therefore, to monitor *ThrottledRequests*, you'll be looking for anomalies. Create a CloudWatch alarm on the **Sum** statistic using the approximate anomaly detection method for the metric whenever *ThrottledRequests* exceeds the determined threshold for more than 5 data points. Because CloudWatch separates *ThrottledRequests* by operation, you'll need to use a **Metric Math Expression** to create an alert on all throttles in aggregate. To do so, select the following operations when creating the alarm:

- PutItem
- DeleteItem
- UpdateItem
- GetItem
- BatchGetItem
- Query
- BatchWriteItem

Add them all together by using the expression *SUM(METRICS())*.

What about ReadThrottleEvents or WriteThrottleEvents?

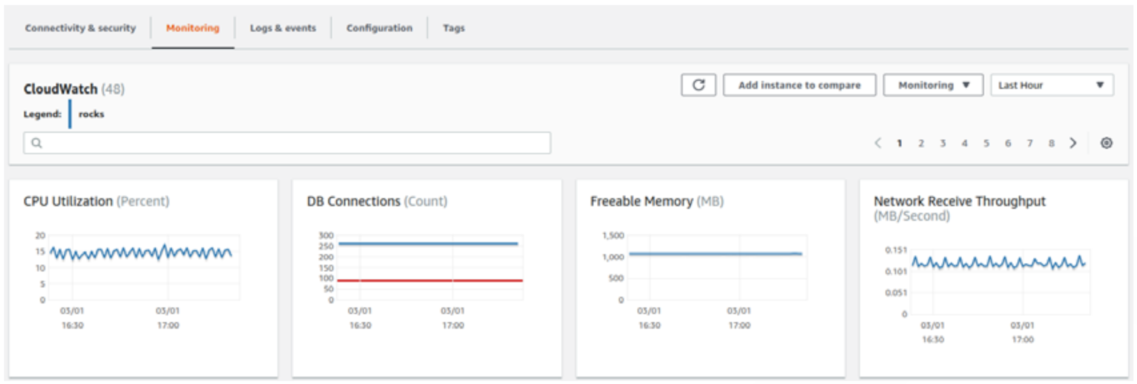
ReadThrottleEvents and *WriteThrottleEvents* are normal occurrences in DynamoDB if you are using batch operations. Your application needs to be able to handle retries for individual events that fail in your batch operations. It seems like these would be good metrics to monitor, but it turns out that if either of these two metrics occur for a request, *ThrottledRequests* will also be incremented, but only once, making it a better indication of failing requests.

CloudWatch for RDS

Amazon Relational Database Service (RDS) allows you to store your application data in databases without having to actually manage the servers the databases are hosted on. It also allows you to easily set up read replicas and take snapshots of your database. In this section, we'll explain how to use CloudWatch to monitor RDS and what metrics are important to watch. RDS metrics are reported on 1 minute intervals.

How to View CloudWatch Metrics

CloudWatch metrics for RDS can be viewed through the **Metrics** portion of CloudWatch, but it is also possible to use the **Monitoring** tab in the RDS console. This tab shows several metric graphs for each database.



Metrics to Watch

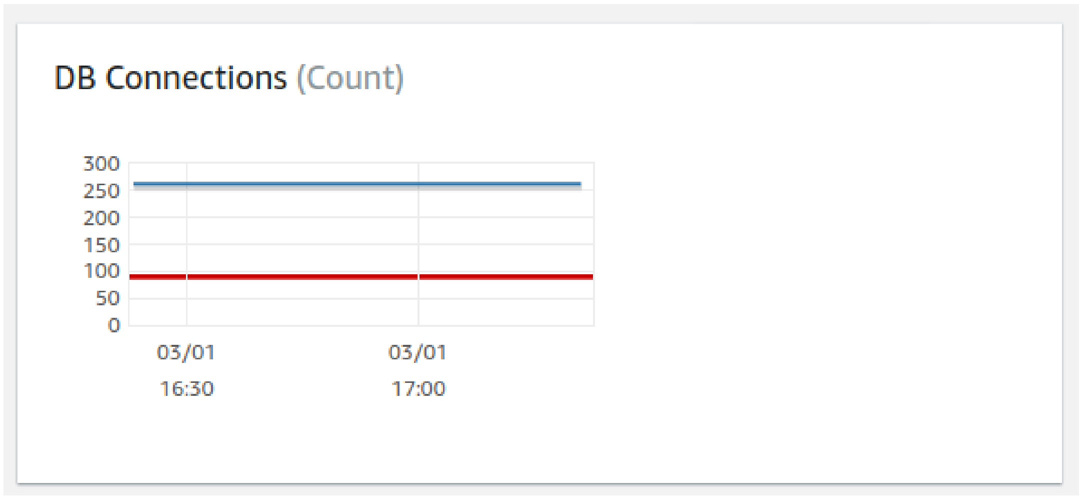
FreeableMemory

RDS's *FreeableMemory* metric refers to the amount of unused memory on a database instance. When this metric gets low, the OS of the database instance may begin to start swapping memory in and out of swap space. This will result in significantly slower reads and make your database unable to respond to requests.

To monitor *FreeableMemory* you should create a CloudWatch alarm on the metric's **Average** statistic that fires when you go below 100MB of free memory.

DatabaseConnections

RDS's *DatabaseConnections* metric measures the number of connections to your database instance. An anomalous number of connections can hint at unexpected behavior in your application. More importantly, reaching the maximum number of connections for your database can also cause new connections to be rejected.



Because the number of collections allowed depends on the size of your db instance type, to monitor *DatabaseConnections* you should first determine the maximum number of connections for your database. Then create a CloudWatch alarm to alert you when you go over 95% of that value for the **Average** statistic.

Deadlocks (Aurora Only)

If you are using the Amazon Aurora database engine, you'll have access to the *Deadlocks* metric. A deadlock occurs when two or more transactions hold locks that each other require. Deadlocks are resolved by aborting one of the transactions and allowing the others to complete, which may have an adverse effect on your application. If you have consistent deadlocks, you will need to examine the queries you're making to find the source of deadlocks.

To monitor *Deadlocks*, just create a CloudWatch alarm on the **Sum** statistic that alerts you when there are any deadlocks in your database. To avoid being too spammy, you'll want to configure the alert to only fire when at least 5 consecutive data points are greater than 0 (that is, it's happening consistently for 5 minutes).

ReplicaLag

RDS's *ReplicaLag* metric measures the number of seconds a replica is behind the primary instance. If your replica gets too far behind the primary and the primary experiences a failure, your replica will be missing data that was in the primary instance.

To monitor *ReplicaLag*, create a CloudWatch alarm on the **Maximum** statistic to alert you when your replica gets too far behind. You'll need to decide how much lag is acceptable for your application, but we recommend no more than 30 seconds. Like *Deadlocks*, you'll want this alert to only fire after at least 5 consecutive data points are over the threshold to avoid alert fatigue for temporary spikes.

NetworkReceiveThroughput & NetworkTransmitThroughput

NetworkReceiveThroughput and *NetworkTransmitThroughput* refer to the number of bytes sent to and from your database, respectively. A sharp spike or drop in either metric could signal that your application is querying your database in an unexpected way, or no longer is querying the database. To monitor *NetworkReceiveThroughput* and *NetworkTransmitThroughput*, create CloudWatch alarms on the **Average** statistic using the approximate anomaly detection method for when 5 consecutive data points are anomalous.

ReadThroughput & WriteThroughput

ReadThroughput and *WriteThroughput* are similar to the network IO metrics except for disk IO. A spike in reads could signal a RDS taking a snapshot, and a spike in writes could hint at expensive table modifications. To monitor *ReadThroughput* and *WriteThroughput*, create CloudWatch alarms on the **Average** statistic using the approximate anomaly detection method for when 5 consecutive data points are anomalous.

CPUUtilization

CPUUtilization tracks the percent of CPU the database instance is using. While an excellent indicator of an overworked database, it tends to fluctuate a lot and can lead to noisy alerts. The key to *CPUUtilization* is to look for sustained high CPU. To monitor *CPUUtilization*, create a CloudWatch alarm on the **Average** statistic using the approximate anomaly detection method for when 15 consecutive data points are anomalous.

SelectLatency, SelectThroughput, CommitLatency, & CommitThroughput

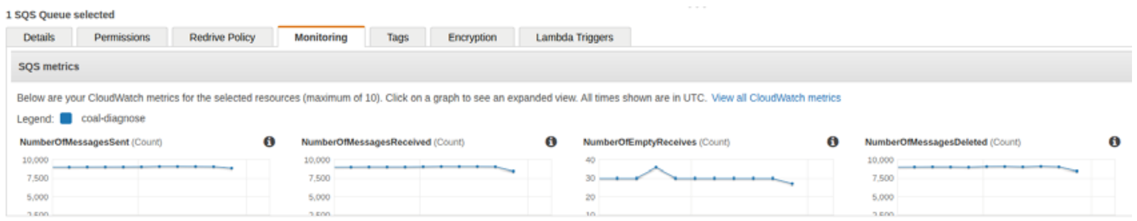
These Aurora only metrics measure the latency for your queries as well as the actual operation counts. Keeping an eye on these metrics can help you correlate issues you may see in your application. To monitor them, create CloudWatch alarms on the **Average** statistic using the approximate anomaly detection method for when 5 consecutive data points are anomalous.

CloudWatch for SQS

Amazon Simple Queue Service (SQS) allows you to send and receive huge numbers of messages from a queue using a simple API. Without setting up any infrastructure, you can have a distributed and fault tolerant queuing system. In this section, we'll explain how to use CloudWatch to monitor SQS and what metrics are important to watch. SQS metrics are reported on 5 minute intervals.

How to View CloudWatch Metrics

CloudWatch metrics for SQS can be viewed through the **Metrics** portion of CloudWatch, but it is also possible to use the **Monitoring** tab in the SQS console. This tab shows several metric graphs for each queue.



Below are your CloudWatch metrics for the selected resources (maximum of 10). Click on a graph to see an expanded view. All times shown are in UTC. View all CloudWatch metrics

Metrics to Watch

The Metric Delay Problem

CloudWatch metrics for SQS are only available at a 5 minute granularity. What's worse, these metrics often have 10-15 minutes of latency, which means that you will not be able to detect an issue in SQS when it actually happens. Because this is the case, if you need to know about issues immediately, you should monitor the producers and consumers of SQS messages in addition to SQS itself.

NumberOfMessagesSent

One of the simplest metrics to watch in CloudWatch is *NumberOfMessagesSent*. This metric measures the number of messages enqueued in a 5 minute interval. It can be useful for determining the health of the systems sending data to SQS. Watch this metric to make sure your producer doesn't suddenly start sending more messages, or stop sending messages completely.

To monitor *NumberOfMessagesSent*, you should look at its graph in CloudWatch for the **Sum** statistic and use the approximate anomaly detection method to determine a healthy baseline. Then create a CloudWatch alarm on the threshold.

Why Not Use NumberOfMessagesDeleted or NumberOfMessagesReceived?

At first glance, it would seem that you want to monitor *NumberOfMessagesReceived* to make sure that all messages are being read from the queue, and *NumberOfMessagesDeleted* to ensure that they are being successfully processed. However, doing so would create CloudWatch alarms that are identical to the ones for *NumberOfMessagesSent*. To illustrate why this is a problem, consider the case when your producer unexpectedly stops. You immediately get an alert for *NumberOfMessagesSent*. Shortly after, when your consumers have cleared out your queue, you will get an alerts for *NumberOfMessagesReceived* and *NumberOfMessagesDeleted*. You already knew about the root problem, but were still notified two more times. Instead, we recommend using ApproximateAgeOfOldestMessage.

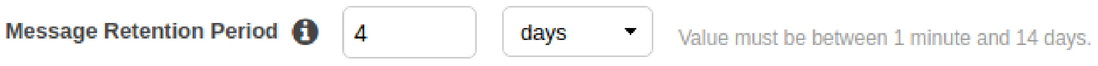
Additionally, we've found that when deleting a message from a queue, CloudWatch records the NumberOfMessagesDeleted at the creation time of the message, rather than the time you deleted the message, which is not useful for real-time monitoring.

ApproximateAgeOfOldestMessage

ApproximateAgeOfOldestMessage measures the number of seconds since the creation of the oldest message in the queue. This metric is effective because if it creeps up, it means that messages are not being processed quickly enough. If you don't have a redrive policy set for your queue, it also alerts you to messages that your consumers can't handle and that are stuck in your queue. For young messages (that is, the oldest message in the queue was added recently), this metric is not guaranteed to be very accurate.

To monitor *ApproximateAgeOfOldestMessage*, view the CloudWatch graph for the metric (using the **Maximum** statistic) to determine a healthy baseline for your queue. If you typically read messages as soon as they come in, the threshold should be close to zero. If you have a more bursty workload, find the average time it takes to clear out the queue and then set your threshold 10% above that. Then create a CloudWatch alarm for when it goes over the threshold you determine.

Additionally, you'll want to set an alarm when *ApproximateAgeOfOldestMessage* gets close to the retention period you set when configuring the queue. If a message gets too old, it will be discarded from the queue and you will lose that data.


Message Retention Period: 4 days
Value must be between 1 minute and 14 days.

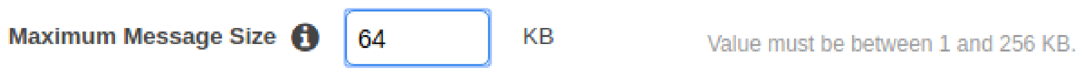
ApproximateNumberOfMessagesNotVisible

Inflight messages are the messages that have been received by a consumer, but have not been deleted or failed. In other words, they are actively being processed. For a standard SQS queue, there is a limit of 120,000 inflight messages, and 20,000 is the limit for FIFO queues. It's important to keep an eye on this limit because if you exceed it, you will be unable to process more messages until you reduce the number of inflight messages.

To monitor for this situation, watch the *ApproximateNumberOfMessagesNotVisible* metric by creating a CloudWatch alarm that alerts when the **Maximum** statistic exceeds 110,000 messages for a standard queue or 18,000 for FIFO queues.

SentMessageSize

When you create your queue, you configure a maximum message size that ranges from 1 to 256 KB. If you exceed that, your message will be rejected. As such, it's a good idea to watch *SentMessageSize* to check for messages that approach the maximum message size.


Maximum Message Size 64 KB

Value must be between 1 and 256 KB.

When monitoring *SentMessageSize*, there are two strategies. If your application cannot tolerate any rejected messages, you'll want to set a CloudWatch alarm for when the **Maximum** statistic approaches your configured maximum message size. This won't actually catch the messages that go over the max message size (since those would be rejected and would not be recorded), but will catch messages that areclose to the limit, giving you an indication that there may be something wrong. If you just need to keep messages from failing generally, do the same with the **Average** statistic.

ApproximateNumberOfMessagesVisible for Dead Letter Queues

When a message repeatedly fails to be processed, it can be sent to a configured Dead Letter Queue (DLQ). Unfortunately, in many cases DLQs are forgotten about and messages sent there disappear into the void. To avoid this problem, you'll need to watch the *ApproximateNumberOfMessagesVisible* metric for the DLQ. Just set a CloudWatch alarm for when the **Sum** statistic exceeds 0 on the queue for 2 data points.

Why Not Use NumberOfMessagesSent?

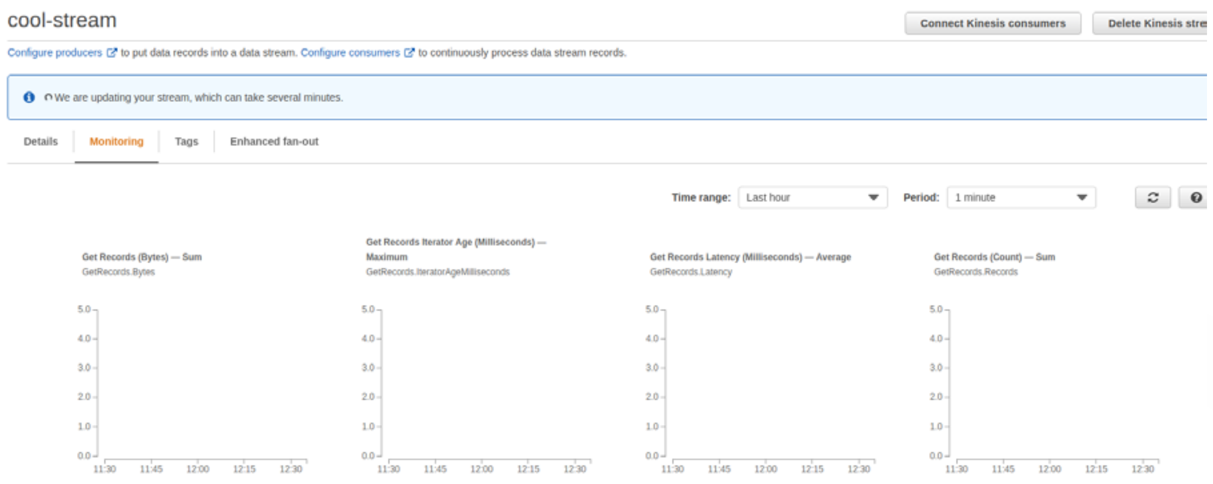
You might be wondering why we can't just use the *NumberOfMessagesSent* metric that we used before to detect messages in the DLQ. Unfortunately, in a somewhat counterintuitive way, messages being sent to the DLQ as a result of failing in the original queue do not increment the *NumberOfMessagesSent* metric for the DLQ.

CloudWatch for Kinesis

Amazon Kinesis is a service that allows you to process streaming data at scale. It lets you maintain multiple iterators and shards your data for you. In this section, we'll describe how to monitor Kinesis with CloudWatch and what metrics are important to watch. Kinesis metrics are reported on 1 minute intervals.

How to View CloudWatch Metrics

CloudWatch metrics for SQS can be viewed through the **Metrics** portion of CloudWatch, but it is also possible to use the **Monitoring** tab in the Kinesis console. This tab shows several metric graphs for each stream.


cool-stream
Connect Kinesis consumers
Delete Kinesis stre
Configure producers to put data records into a data stream. Configure consumers to continuously process data stream records.
We are updating your stream, which can take several minutes.
Details
Monitoring
Tags
Enhanced fan-out
Time range: Last hour
Period: 1 minute
Get Records (Bytes) — Sum
GetRecords.Bytes
Get Records Iterator Age (Milliseconds) — Maximum
GetRecords.IteratorAgeMilliseconds
Get Records Latency (Milliseconds) — Average
GetRecords.Latency
Get Records (Count) — Sum
GetRecords.Records

Basic vs Enhanced Level Monitoring

CloudWatch can provide two levels of metric granularity, basic and enhanced. By default, a Kinesis stream has basic level granularity enabled, which means CloudWatch will collect metrics for the stream as a whole, and not for individual shards that make up the stream. To get shard level metrics, you will need to use the EnableEnhancedMonitoring API to turn on enhanced granularity for a stream. Using this API, you can select which metrics you'd like to enable enhanced monitoring for your stream.

You can enable enhanced monitoring for the following metrics:

- IncomingBytes
- IncomingRecords
- OutgoingBytes
- OutgoingRecords
- WriteProvisionedThroughputExceeded
- ReadProvisionedThroughputExceeded
- IteratorAgeMilliseconds

You can also access enhanced monitoring through the Kinesis stream UI. To do so, select your stream from the list of streams in the Kinesis console and expand the section labeled **Shard level metrics**.

Why Enable Enhanced Monitoring?

Enhanced monitoring is useful for determining when your shards are not being evenly utilized. If any shards deviate significantly from the average, you should assess the evenness of your partition key's distribution. Even if your partition keys are designed to be evenly distributed, there are times when you can have "hot" or "cold" shards. In these cases, you should split over-utilized shards and merge under-utilized shards.

Why Shouldn't I Always Enable Enhanced Monitoring?

Basic level monitoring is included for free with Kinesis, but each metric you enable will be charged as a custom metric in CloudWatch. Each stream has 7 metrics that can be enabled, meaning each stream could cost $2.10/mo to monitor with enhanced monitoring. If you enabled enhanced monitoring on all metrics for all your streams it can get expensive quickly, especially if you aren't using the data.

Metrics to Watch

Gotchas to Look out for

When using the Kinesis metrics in CloudWatch, you'll need to be aware that some statistics do not behave in an intuitive way. In some cases, the **Average**, **Minimum**, and **Maximum** statistics for a metric will apply to only individual API calls, rather than values across the stream at that point in time. Take, for example, the **Average** statistic for *IncomingRecords*. Rather than getting the average number of records put to the stream for a time period, you would get the average size of the batches of records sent to the Kinesis stream. Similarly, **Minimum** would return the smallest batch size, and **Maximum** would return the largest.

This applies to the following list of metrics:

- GetRecords.Bytes
- GetRecords.Records
- IncomingBytes
- IncomingRecords
- SubscribeToShardEvent.Bytes
- SubscribeToShardEvent.Records

GetRecords.IteratorAgeMilliseconds

GetRecords.IteratorAgeMilliseconds measures the difference between the age of the last record consumed and the latest record put to the stream. This metric is particularly important to monitor because having too high of an iterator age in relation to your stream's retention period can cause you to lose data as records expire from the stream. AWS recommends that this value should never exceed 50% of your stream retention; when you get to 100% of your stream retention, data will be lost. Monitor the **Maximum** statistic to make sure none of your shards ever approach this limit.

If you are getting behind, a temporary stopgap is to increase the retention time of your stream; the real solution is to add more consumers to keep up with the rate at which data is being put to your stream.

ReadProvisionedThroughputExceeded

When your consumers exceed your provisioned read throughput (determined by the number of shards you have), they will be throttled and you won't be able to read from the stream. This can start backing up your stream.

If you find that you are being consistently throttled, you will have to add more shards to your stream to increase your provisioned read throughput. If adding more shards doesn't lower the number of throttles, you may have a "hot" shard that is being read from more than others. Enable enhanced monitoring, find the "hot" shard, and split it.

To monitor *ReadProvisionedThroughputExceeded,* use the approximate anomaly detection method to create a CloudWatch alarm for the **Average** statistic. Ideally, you should try to get this value as close to 0 as possible.

WriteProvisionedThroughputExceeded

When your producers exceed your provisioned write throughput (determined by the number of shards you have), they will be throttled and you won't be able to put records to the stream. To fix consistent throttling, you will have to add shards to your stream. This will raise your provisioned write throughput and keep you from being throttled in the future.

To monitor *WriteProvisionedThroughputExceeded,* create a CloudWatch alarm using the approximate anomaly detection method on the **Average** statistic to help you to determine if your producers are healthy.

PutRecord.Success, PutRecords.Success

PutRecord.Success and *PutRecords.Success* are incremented whenever your producers succeed to send data to your stream. Monitoring for spikes or drops can help you monitor the health of your producers and help you catch problems early. You'll want to create a CloudWatch alarm on the **Average** statistic using the approximate anomaly detection method for whichever of the two API calls you use (because CloudWatch splits the two APIs into two different metrics).

GetRecords.Success

GetRecords.Success is the consumer-side corollary to *PutRecords.Success*. As such, looking for spikes or drops in this metric will allow you to ensure your consumers are healthy and let you catch problems early. Create a CloudWatch alarm using the approximate anomaly detection method with the **Average** statistic for this purpose.

Next Steps

Now that you know how to use CloudWatch to monitor your AWS infrastructure, take a look at Blue Matador.

As you've seen from reading this book, it can be error prone and tedious to set up CloudWatch monitoring for all your resources (and remembering to do it for newly created resources is tough!). Even after you get your alarms set up, your resource utilization will continue to change, so expect to spend time tweaking thresholds.

Instead, take the manual work and toil out of monitoring and use Blue Matador.

After a fast onboarding, Blue Matador:

- **Quickly identifies every resource, every service, and every server**
- **Automatically creates hundreds of alarms out-of-the-box without any configuration**
- **Proactively notifies of any potential issues**
- **Dynamically updates alarms as your AWS environments scales and evolves**

RUN APPLICATIONS WITH CONFIDENCE
WITH BLUE MATADOR, YOU WON'T MISS ANYTHING.

START YOUR FREE TRIAL

bluematador