signal fx

BEST PRACTICES FOR CREATING ALERT DETECTORS

Table of Contents

Introduction 3

A Refresher: Simple Detectors 4

Monitoring All The Things 5

Flapping Detectors 5

Aperiodic Data 10

Delayed Data 12

Cyclical Patterns In Your Data 15

Compound Conditions 16

Large Numbers Of Sources 18

Ephemeral Infrastructure 19

Conclusion 20

Introduction

Applications today are built on cloud infrastructure and containers, as well as third-party and custom services. As you are undoubtedly too well aware, monitoring cloud applications requires ingesting and analyzing huge amounts of data from hundreds and even thousands of web services—many of which employ dynamic, scale-out architectures—and confronting fluctuating workloads.

Businesses must respond quickly to a range of demands or risk losing customers; yet cloud operations teams must create effective alerts without the luxury of prolonged trial-and-error processes. Determining the best alert conditions and the impact they will have on cloud operations professionals is complex and challenging. Arguably your most critical concern is avoiding alert fatigue, a condition that can cause a team to lose confidence in the alerting process and ignore what they shouldn't.

Traditional approaches to alerting are inflexible and inadequate for the cloud environment in which you operate. Health checks lack the context to assess the nature of a supposed issue, and understanding what is normal and what is not can be difficult. Currently available platforms deliver false positives and meaningless alert storms. The noise and confusion are bad for operational efficiency—and bad for business.

As with most time series-centric systems, SignalFx makes it easy for you to alert on well-behaved (periodic, long-lived, on-time) raw metrics, using simple thresholds (e.g., "greater than 80") and with a large time tolerances (reporting within 2 minutes of the detected event). However, real-world data and real-world scenarios are rarely this well-behaved, simple or forgiving. In many cases, you not only need to know as soon as possible when you have an actual issue (preferably within seconds), you also need to make sure your alerting is able to cover cases where data is:

- *Aperiodic:* Not all metrics report regularly. For example, metrics sent upon the occurrence of events are unlikely to have a regular cadence.
- *Ephemeral:* Elements that are emitting metrics may be short-lived, or there may be gaps between retiring one set of sources and introducing their replacements.
- *Delayed:* For any number of reasons—a busy machine, insufficient bandwidth, buggy instrumentation—data may be sent or received with a wall clock time that is significantly later than the actual time of the measurement.

Alerts should also be able to account for:

- *Composite or otherwise processed signals:* The raw data you send may not be the same as the data you want to alert on (e.g., you may want to use aggregates of some kind such as percentiles or sums or formulas such as ratios to represent rate of change).
- *Cyclical or seasonal patterns:* Monday afternoons, for example, are normally busier than Tuesday mornings, thus, static thresholds for triggering alerts are ineffective.
- *Outliers:* This refers to data that shouldn't be included in a baseline (e.g., the average over the last few weeks except when there was an outage).
- *Scale:* If the number of elements you are monitoring changes, you shouldn't have to revisit your alerts.
- *The potential for "flapping" behavior:* If the metric values are hovering around a static threshold, they may cause alerts to fire and clear repeatedly

Unlike other monitoring systems, SignalFx provides a number of capabilities that allow you to alert on these real-world scenarios accurately and in a timely manner.

A Refresher: Simple Detectors

Terminology

SignalFx detectors evaluate metric time series against a specified condition, and optionally for a duration. When a condition has been met, detectors generate events of a specified severity. In the SignalFx application, events generated when detector conditions are met are referred to as alerts. Among other things, alerts can be used to trigger notifications in incident management platforms such as PagerDuty or messaging systems such as Slack or email.

Using Static Thresholds

The most basic kinds of alerts trigger immediately when a simple metric crosses a static threshold, for example any time CPU utilization goes above 70 percent. Fixed thresholds are easy to implement and interpret when there are absolute goals to measure against. If we know the typical memory/CPU profile of a certain application, say, we can define bounds that encode normal state. If we have a business requirement to serve requests within a certain time period, we know what unacceptable latency for that function is.

Consistent Signal Types

For a detector to work properly, the signal evaluated should represent a consistent type of measurement. This is the normal state of affairs when you choose a metric; for example, `cpu.utilization` as reported by the SignalFx collectd agent is a value between 0 and 100 and represents average utilization across all CPU cores for a single Linux instance or host. If you use wildcards in your metric name, you should make sure they do not mistakenly include metrics of different types. For example, if you enter jvm.* as the metric name, this could cause your detector to evaluate `jvm.heap, jvm.uptime` and `jvm.cpu.load`—assuming they are metric names in use in your SignalFx organization—against the same threshold, which may lead to unexpected results.

Viewing at Native Data Resolution

A common and easy way to create a detector is to first create a chart, which lets you visualize the behavior of the signal you want to alert on. You then convert the signal to a detector by selecting New Detector from Chart from the chart builder actions menu.

If you do this, make sure you are visualizing the data at its native resolution, because this gives you the most accurate picture of the data your detector will evaluate. For example, if you create a detector using a metric that reports every 10 seconds, you should make sure the time range for your chart is small enough—say, 15 minutes—so you can see individual measurements every 10 seconds.

Doing so is important because, by default, SignalFx chooses a chart display resolution that fits within the time range you've chosen and summarizes the data to match that resolution. For example, if you are using a metric that reports every 10 seconds but look at a one-day window, then by default, the data you see on the chart will represent 30-minute intervals. Depending on the rollup or summarization method chosen, this can mean that any peaks or dips are averaged out, which gives you an inaccurate understanding of your signal and what constitutes an appropriate detector threshold.

Monitoring All The Things

Monitoring Individual Members of a Population

SignalFx offers a simple and concise way of defining detectors that monitor a large number of similar items, such as CPU utilization of all the hosts in a given cluster. This is done through the metadata associated with metric time series and is analogous to how that metadata—dimensions, properties or tags—is used in creating charts.

For example, if you have a group of 30 hosts that provide a clustered service such as Kafka, you will normally include a dimension like `service:kafka` with all the metrics coming from those hosts.

In this case, if you want to track whether CPU utilization remains below 70 percent for each of those hosts, you can create a single detector for the `cpu.utilization` metric that is filtered using the `service:kafka` dimension and evaluates them against the static threshold of 70. This detector will trigger individual alerts for each host whose CPU utilization exceeds the threshold—just as if you had 30 separate detectors.

In addition, if the population changes—say, because the cluster has grown to 40 hosts—you do not need to make any changes to your detector. Provided you have included the `service:kafka` dimension for metrics coming from the newly added hosts, the existing detector will find them and automatically include them in the threshold evaluation.

This kind of detector works best when all members of the population have the same threshold and the same notification policy (e.g., publish alerts into the same Slack channel). If you have different thresholds or notification policies, you need to create multiple detectors—one for each permutation of threshold and notification—although the likely number of such detectors will still be fewer than the count of individual members.

Monitoring Sub-Groups in a Population

You can also use detectors to look at aggregated sub-groups in the population. Let's say you have 100 hosts, divided among 10 services, and you want to make sure the 95th percentile of CPU utilization across the cluster of hosts that provides each of those services remains below 70 percent. In this case, you create a single detector for `cpu.utilization`, then apply an analytics function of `P95`, grouped by `service`—assuming `service` is a dimension or property. This approach will not work with tags.

Similar to the previous detector example, this detector triggers alerts for each service, just as if you had 10 separate detectors. And as before, if you add services, they will automatically be included in what this detector monitors, as long as you have included a `service` dimension or property for the newly added service's metrics.

Flapping Detectors

Detectors using static thresholds are easy to create and understand. During the initial setup it can be tempting to have them alert immediately every time a signal crosses a threshold. This offers the benefit of timeliness: if elevated signal values reveal problems to come, you will be informed as soon as possible. But even if you choose relevant signals and reasonable thresholds, the elevated value may only reveal transient system stress. Ultimately this simplest form of detectors may lead to alerts that fire and clear repeatedly in a short period of time ("flapping").

There are several ways to handle this phenomenon in SignalFx, as discussed in the following sections.

Duration

The most straightforward method is to set not only a threshold but also a duration for the alert. For example, instead of triggering an alert immediately when CPU utilization exceeds 70 percent, you can require that utilization be above 70 percent for two minutes. In SignalFx the severity of the alert and associated notification are also configurable. That is, you can have a minor alert—above 70 percent for two minutes, say—send a message to a Slack channel, while a critical alert—above 90 percent for five minutes, say—pages the on-call engineer.

Percent of Duration

One issue with duration conditions is that they may be too stringent: a single reading below the threshold means the detector will not fire an alert. To handle this, you can configure SignalFx detectors with refined duration conditions. For example, you can require that CPU utilization be above 70 percent for 80 percent of a five-minute period.

Note that using percent of duration conditions requires that you know the resolution of your detector, which is normally equivalent to the resolution of your data, to be able to accurately interpret its behavior. The reason is that the denominator for calculating percent of duration is the number of data points ***expected***, not the number of data points actually ***received***.

Let's say you are sending CPU utilization every 10 seconds. This means that in a five-minute window you should have 5 x 6 = 30 measurements of CPU utilization, and that 80 percent of the five-minute window means that 24 values must be above 70 percent for your detector to fire. However, if only 20 out of 30 expected data points arrive, and even if 16 of them, or 80 percent, are above 70 percent, the detector will ***not*** fire. The rationale for this behavior is that the detector (a) expects periodic data (i.e., every 10 seconds in this case), and (b) makes no assumptions about the possible "goodness" or "badness" of missing data. As such, missing data can leave the detector in an ambiguous state, and the detector defaults to retaining its previous state rather than raising or clearing alerts based on assumptions.

One important consequence of this behavior is that you cannot always use percent of duration conditions for data that is aperiodic. See the two following sections for ways to handle this data.

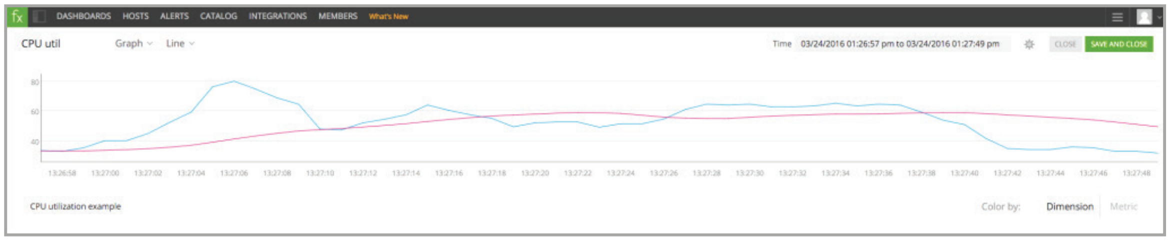
Smoothing Transformations

Another way to address flapping is to use SignalFx's analytics functionality: instead of firing an alert on the raw metric, you can first apply a smoothing transformation to obtain a better signal which experiences less fluctuation. The goal of smoothing transformations is to reduce the impact of a single, possibly spurious, extreme observation.

One such transformation is the rolling mean, which replaces the original signal with the average of its last several values, where "several" is a parameter—the window length—that you can specify. In effect, you replace the signal with a summary of its behavior during the last window.

One motivation for applying the rolling mean is to think of the observed values as obtained by sampling the true signal at regular intervals. Then, averaging the last few sample values is a method of approximating the true signal.

In practice, setting a threshold on the rolling mean is similar to creating a static threshold on a raw metric for some duration. For example, if a metric is above 60 percent for a two-minute duration, the two-minute rolling mean will be at least 60 percent. Setting a 60 percent threshold against the two-minute rolling mean will trigger in a variety of scenarios, corresponding roughly to several static threshold and percent of duration combinations.



Custom Clearing Conditions

The final method for handling flappiness is to establish a separate **clearing** condition. By default, an alert clears when the original trigger condition is no longer true. For example, if your alert is triggered immediately by CPU utilization that rises above 70 percent, it will clear when CPU utilization sinks below 70 percent (again, immediately). This kind of clearing condition works well if your signal does not encounter the threshold frequently or in quick succession. However, if your signal hovers in the vicinity of the threshold, the reciprocal clearing contributes directly to flappiness.

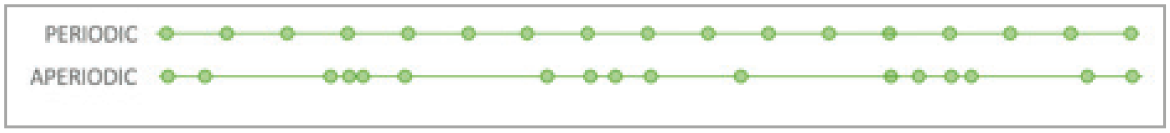
As noted, it's possible to avoid flappiness by using durations or smoothing transformations, but those approaches trigger alerts only after a period of time—the duration of the detector or the window of the transformation—has elapsed. In some cases, you want to be notified immediately after a threshold has been crossed, but you don't need or want it to clear so quickly.

Let's say you have a detector that monitors your heap usage for a Java application. Garbage collection will ensure that the heap metric resets periodically, so if your detector triggers on static threshold and clears reciprocally, then it may not only be noisy, but also not particularly helpful. Instead, what you want is to be told when heap exceeds that threshold and stays below some other threshold longer than a single period of garbage collection.

To build a detector that accounts for this class of scenarios, you can establish a separate clearing condition. See our blog "Reducing Alert Noise: Ranges for Firing and Clearing" for an example.

Aperiodic Data

Some of the metrics used in applications and infrastructure monitoring are aperiodic in nature. In other words, successive data points from a given emitter for that metric are not regularly spaced in time.



It is not uncommon to encounter aperiodic data if you are generating your metric based on the occurrence of an event. Let's say you're measuring the latency of transactions for your application, and your instrumentation captures values only when a transaction has occurred and then sends them once a minute to SignalFx. Further, your transactions occur more often during the day than at night, such that there are minutes-long stretches at night when there are only one or two transactions. For the time where there are no transactions, you don't send any data points to SignalFx, and SignalFx records Null values.

In this case, if you want to be alerted when latency is trending above a certain threshold, your detector must account for the greater regularity of data during the day. For example, you can create a static threshold detector with a percent of duration (e.g., above 5s for 80 percent of the last 15 minutes), and it would work when the metrics are received regularly by SignalFx. However, at night, such a detector would likely never fire, because you would rarely have enough full data sets—15 successive values at one-minute intervals. For an explanation, see the section "Percent of Duration."

To compensate for aperiodicity, you can use SignalFx analytics to express a percentage of data points received during some window above (or below) a static threshold, and alert on that. Let's use the same example as earlier—that we want to be alerted when 80 percent of the data points received in the last 15 minutes are above the value 5

When a detector is based on actual metrics data, you can sometimes simply use an extrapolation policy—typically Last Value—along with a percent of duration condition. For cases where your detector involves some computation, however, this may not work, because extrapolation can cause an inaccurate computation. For example, if you are calculating an average or some other aggregation, you typically do not want the extra values inserted by extrapolation, because they throw the weighting off. In those cases, you must take the approach in the following paragraphs.

The key ingredient is the Count transformation—in this case, with a window of 15 minutes. This analytics function is available in the SignalFx API, but not the UI. You can mimic this in the UI by dividing the 15-minute Sum by the 15-minute Mean. Let's call this derived signal "S - Count (15m)." Next, count the data points received in the last 15 minutes that are above 5. To calculate this, divide Signal [Exclude < 5] [Sum (15m)] by Signal [Exclude < 5] [Mean (15m)]. Let's call this derived signal "S above 5 - Count (15m)."

Now the quotient [S above 5 - Count (15m)] / [S - Count (15m)] is the proportion of data points received in the last 15 minutes that are above the value 5. Finally, alert immediately when this signal is above 0.8.

Note that the quotient—the signal on which we alert—publishes Null if all data points received in the last 15 minutes are below the threshold of 5. In particular, it publishes Null if no data points have arrived in the last 15 minutes. In the scenario where only values below 5 are observed, the quotient always publishes Null and the detector never fires, as desired.

However, if the detector fires, no data arrives for 15 minutes, and then data points below the threshold of 5 begin to arrive, the detector will not clear, because a Null value can induce neither an alert nor its clearing. (Note that this problem can be resolved using the SignalFx API. Also, this workflow is packaged in the aperiodic module in the SignalFlow library, which can be invoked in a SignalFlow program.)

In this example, the 15-minute duration is expressed in the analytics. In choosing a duration, you must make a tradeoff: long durations give more accurate alerts, because you are calculating the proportion on more data points; shorter durations deliver more timely alerts, which may be very important during business hours.

The Exclude analytics function provides an elegant way to handle this tradeoff. Suppose, for example, you want to calculate with a 15-minute duration during business hours—and be alerted when at least 80 percent of the data points received are above 5—but you do not want to be alerted overnight during a 15-minute window in which exactly one data point is received, and its value is above 5. Let's say you want an alert to trigger when 80 percent of the data points received in the last 15 minutes are above 5 and you have received at least 10 data points above 5 in the last 15 minutes. Create the signal "S above 5 - Count (15m)" as in the earlier example and apply Exclude < 10. This publishes the number of points above 5 in the last 15 minutes, provided at least 10 such points have been received; otherwise it publishes Null. Then form the denominator shown earlier and use the quotient:

[S above 5 - Count (15m) - Exclude < 10] / [S - Count (15m)].

This is exactly the proportion of data points above 5 received in the last 15 minutes, if at least 10 points above 5 have been received. If fewer than 10 such have been received, this signal is Null. Note the same caveat about failing to clear applies to this signal.

Delayed Data

Metrics sent to the SignalFx service can sometimes be delayed. In other words, the time at which an actual measurement was taken may differ significantly—on the order of minutes—from the time when the data actually arrives at SignalFx. The reasons for delay are many and varied: there could be network congestion; the host from which the metric is being sent may be busy; the process or agent responsible for sending data may be improperly configured; the API you access to gather the data may provide you with data in batches every five minutes, even though it is taking measurements every 10 seconds; and so on.

The lack of timeliness has an impact on SignalFx detectors, because they are optimized to use streaming real-time data, sent with monotonically increasing timestamps. Consistently delayed data affects how quickly a detector can react to the data but has no impact on accuracy. Inconsistently delayed data, on the other hand, may impact the accuracy of detector computations.

Let's say your detector condition depends on calculating the average value of CPU utilization from 10 servers, and the metrics from all the servers are consistently 15 seconds behind. Because the SignalFx system knows about that delay, it knows to calculate the average value after waiting 15 seconds for the data to arrive. Your alerts may only fire 15 seconds after the event, but they fire accurately based on the average CPU utilization values for all 10 servers.

Further, let's say that one of the servers begins to arrive a full minute later than is expected (i.e., a total of 1 minute and 15 seconds). In that case, the detector must decide whether it should wait that extra minute to calculate that average (i.e., wait for all the data to arrive), or if it should simply perform its calculation with the data it has at a given point in time (i.e., potentially without the data from the 10th, "late" server).

Of course, it would be better to not have to make the choice: timely data yields timely and accurate alerts. For that reason, if some of the causes of delay—especially those that are volatile—are within your control, you should find ways to address them.

However, for situations where the data is delayed and not easily correctable, you will want to make sure that your detectors still perform with the level of accuracy that you need. SignalFx provides several mechanisms to help account for delayed data points, as discussed in the following two sections.

The `MaxDelay` Parameter

Every detector in SignalFx includes a parameter, MaxDelay, that sets a deadline for incoming data to be included in its evaluation set. This parameter is expressed as a number of minutes or seconds up to a maximum value of 15 minutes. A detector will wait up to the MaxDelay limit for expected data points to arrive. If they arrive after that, they are not considered, even though they are persisted in the backend data store and are available to other charts or detectors.

MaxDelay is set automatically and dynamically based on observed latencies of incoming data points, but you can also set it manually to a fixed value if you have a good understanding of how your data arrives at SignalFx.

It is generally a good idea to leave MaxDelay in auto mode. However, if emitter lag is erratic, even dynamic max delay may exclude some data.

A manual MaxDelay should be used in the following circumstances:

- *As an upper limit to lag:* If chart or detector timeliness is critical and you don't want the computation to lag by more than a specified limit
- *As a lower limit to lag:* If emitter lag is erratic and you want to specify a minimum wait time to accommodate the emitter tardiness

In practice, there is only one MaxDelay value per detector, and how you use it—and how you interpret it—depends on the problem you are trying to solve.

Extrapolation

If a data point is sufficiently delayed as to be excluded because it has exceeded MaxDelay, or if the data point does not exist because a measurement was never made or reported, it is considered a Null value for purposes of the detector's computations. Depending on the nature of your data, it may be appropriate to configure an **extrapolation policy** to compensate for potential Null values.

An extrapolation policy creates synthetic data points for missing data. You should choose the specific policy to complement the metric and rollup type. For example, a counter metric with a sum rollup is probably best served with a zero extrapolation, whereas a last-value extrapolation may be better for a gauge with a mean rollup.

The Max Extrapolations parameter indicates the number of consecutive data points for which the selected policy applies. By default, extrapolation is set for infinity, but there are cases where this is inappropriate. For example, if your detector uses the `count` analytics function, you will want to limit the number of times an extrapolated value is used, because that can create an inaccurate result.

How can this affect your data? A `count` analytics function on a metric can be used to create host up or down detectors. If the extrapolation policy is set to Zero or Last Value the synthetic value created may hide whether the host is up or down, leading to an inaccurate detector.

Note that a last-value extrapolation will not yield a synthetic value prior to the first real value and may therefore not synthesize any values for inactive or dead time series. In other words, a time series that never reports data cannot be made to report a value using extrapolation.

Policy	Behavior
Auto	Applies the default policy for selected metric type and rollup
Null	Inserts a NULL value for missing datapoints
Zero	Inserts a zero (0) value for missing datapoints
Last value	Uses the last reported value until the next datapoint arrives

In the following table, we are looking at a metric that reports values every five seconds, and it skips two intervals. The values output after applying different extrapolation policies are as follows:

Value of metric time series at time t =					
Time t =	**10:01:05 AM**	**10:01:10 AM**	**10:01:15 AM**	**10:01:20 AM**	**10:01:25 AM**
Received data point value	10	15	null	null	5
Null extrapolation	10	15	null	null	5
Zero extrapolation	10	15	0	0	5
Last-value extrapolation	10	15	15	15	5
Linear extrapolation	10	15	20	25	5

You can see that:

- Null extrapolation policy does not alter the values at all.
- Zero extrapolation policy inserts zeros in place of null values. This is often used for counters that only report when there is a value and where a null value is properly interpreted as a zero.
- Last-value extrapolation policy uses the last value it received. This is most often used for cumulative counters and gauges, where a null value is usually interpreted as no change in value.
- Linear extrapolation policy uses the last two data points received to determine what the following values would have been.

Cyclical Patterns In Your Data

It is relatively common for the values of a metric to exhibit a weekly periodicity. That is, if most of an applications' users live in a certain time zone, and the metric is influenced by user activity, normal Monday afternoons tend to look similar to each other and different from, say, Thursday evenings or Sunday mornings. For such metrics, sensible static thresholds may be difficult to define, because the notion of normal behavior itself changes with time.

Time shifting is the basic strategy for dealing with cyclicality in metric values. For example, we can divide the 30-minute rolling mean of a metric by the 30-minute rolling mean of the metric shifted by one week. We then expect the value of the quotient to be relatively close to 1, and we alert when the value is very different from 1 (with the exact discrepancy determined by our tolerance for deviation).

While a dynamic threshold is generally better than a static threshold, comparing current values to a single time-shifted window can be quite noisy. In particular, alerts are likely to echo one week after they happen. For example, we might see a flurry of activity because a large customer has been onboarded or a change in metrics due to a code deployment, and our alerts (correctly) fire. Unfortunately, unless similar events happen precisely one week later, we receive alerts again the following week because normal activity does not match the behavior observed during the previous week's events.

To deal with this phenomenon, we suggest using multiple time-shifted windows and discarding outliers, to calculate the dynamic threshold against which to compare. For example, we can calculate 30-minute rolling means one, two, three, and four weeks ago; discard the highest and lowest values; alert when the current value is greater than 20 percent (say) above the larger remaining value, or less than 20 percent (say) below the smaller remaining value. With more data, we can more confidently construct a band of normal values; unless two past outliers align perfectly—in this example, are spaced exactly one week apart—this strategy eliminates their influence on the dynamic threshold. We can obtain variations on this strategy by summarizing past data in different ways. For example, instead of constructing a +/- 20 percent band around a rolling mean, we can construct a band using standard deviations. This replaces the static value of 20 percent with a dynamic estimate of the typical spread of values.

See the "Dealing with Weekly Periodicity" section in our blog Analytics in Action: Finding The Right Signal To Alert On for an example.

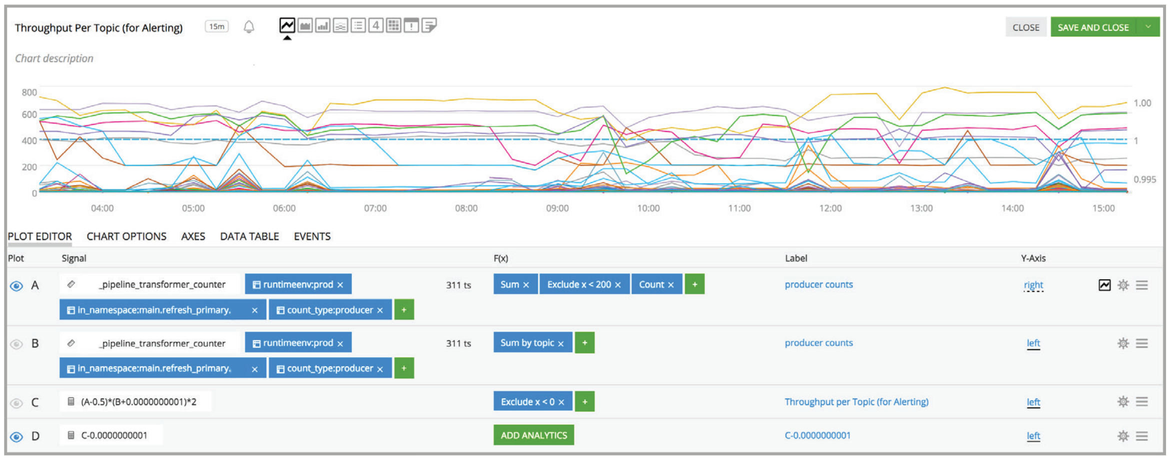
Compound Conditions

Although you can generate many alerts accurately with a single signal and threshold, especially with the proper analytics, there are cases when you need compound conditions. For example, alert on condition A only when condition B is also not true. Common examples include detectors on services with +1/-1 dependencies—service B is downstream of service A, so if service A is having problems, there's no need to trigger alerts on service B because the root cause is upstream—or service component detectors—component services A, B and C are all part of service D, so if an alert is firing on D, you know that A, B and C will all be affected, and there's no need to alert on them. Conversely, if D is not firing, any individual alerts on A, B and C should fire as needed.

While SignalFx API supports Booleans for detector conditions, direct UI access can accommodate such detector requirements with the following two approaches: toggle on dependent metrics and service component detectors.

Toggle on Dependent Metric

First, take the two signals you are using and decide which one you want to receive alerts on. For example, if you are the owner of service B, you care about the signal that applies to B. Second, apply analytics functions to the service A metric, such that the output is 0 when you don't want an alert on B, and 1 when you do. Third, create a plot line C that is the product of those two metrics, and alert on C.



Plot	Signal		F(x)	Label	Y-Axis
A	_pipeline_transformer_counter, runtimeenv:prod, in_namespace:main.refresh_primary., count_type:producer	311 ts	Sum, Exclude x < 200, Count	producer counts	right
B	_pipeline_transformer_counter, runtimeenv:prod, in_namespace:main.refresh_primary., count_type:producer	311 ts	Sum by topic	producer counts	left
C	(A-0.5)*(B+0.0000000001)*2		Exclude x < 0	Throughput per Topic (for Alerting)	left
D	C-0.0000000001			C-0.0000000001	left

We have documented a more sophisticated approach for ANDing two conditions, with no metadata subtleties in "Creating and Testing the Alert" in our blog "Analytics in Action: Finding The Right Signal To Alert On".

Service Component Detectors

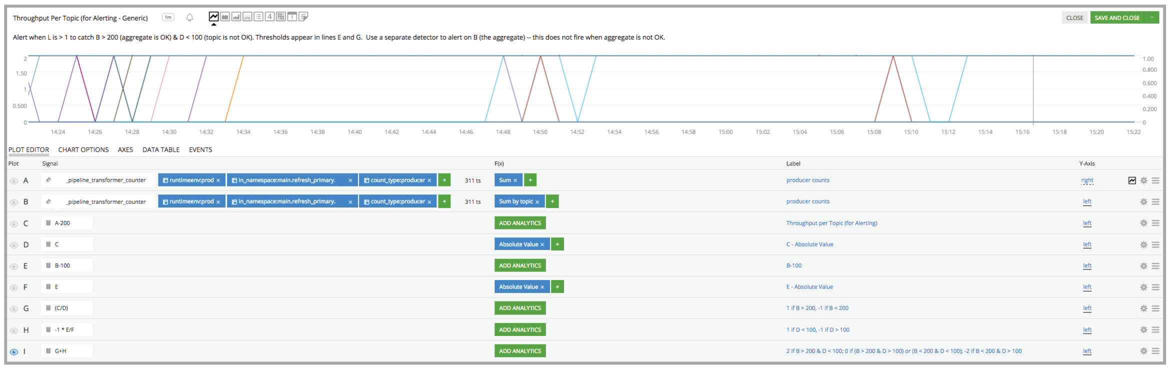
To illustrate how to alert on an individual component only when the aggregate service is ok, let's say your metric is measuring throughput per topic on a Kafka bus and you want to be alerted on individual topics when they are too low, but only when the aggregate is within normal range.

In this case, you have two primary signals: one (A) that is the sum of throughput across all topics and another (B) that is the sum of throughput for each topic. (There will be only one output stream for the former and one output per topic for the latter.) For the sake of the example, let's assume you are using static thresholds: if the sum across all topics is greater than 200, aggregate throughput is considered within normal range, and if the sum per topic is less than 100, it is too low.

Then apply the following formula:

$[(A - 200) / |A - 200|] - [(B - 100) / |B - 100|]$

If the result of this equation is greater than 1, A is greater than 200 and B is less than 100, and you should receive an alert. Note that if you want to alert on the aggregate, you must build a separate detector.



Alert when L is > 1 to catch B > 200 (aggregate is OK) & D < 100 (topic is not OK). Thresholds appear in lines E and G. Use a separate detector to alert on B (the aggregate) -- this does not fire when aggregate is not OK.

PLOT EDITOR   CHART OPTIONS   AXES   DATA TABLE   EVENTS

Plot	Signal		F(x)	Label	Y-Axis
A	_pipeline_transformer_counter · runtimeenv:prod · in_namespace:main.refresh_primary. · count_type:producer	311 ts	Sum	producer counts	right
B	_pipeline_transformer_counter · runtimeenv:prod · in_namespace:main.refresh_primary. · count_type:producer	311 ts	Sum by topic	producer counts	left
C	A-200		ADD ANALYTICS	Throughput per Topic (for Alerting)	left
D	C		Absolute Value	C - Absolute Value	left
E	B-100		ADD ANALYTICS	B-100	left
F	E		Absolute Value	E - Absolute Value	left
G	(C/D)		ADD ANALYTICS	1 if B > 200, -1 if B < 200	left
H	-1 * E/F		ADD ANALYTICS	1 if D < 100, -1 if D > 100	left
I	G+H		ADD ANALYTICS	2 if B > 200 & D < 100; 0 if (B > 200 & D > 100) or (B < 200 & D < 100); -2 if B < 200 & D > 100	left

Large Numbers Of Sources

Today, a single SignalFx plot line, represented by one letter or row in a chart or detector builder view, can process up to 5,000 metric time series. Each metric time series is equivalent to the combination of a measurement being taken—denoted by the metric name, such as `memory.free`—and unique permutations of sets of one or more dimensions used to characterize or describe the scope of the measurement—denoted as key:value pairs, such as `hostname:host1234`.

For metrics emitted from a very large number of sources, performing computations accurately requires that you break down the metric time series into groups that fit within the current per-plot line limits. For example, if you have 17,000 hosts emitting a free-memory metric `memory.free` and you want to sum up the total free memory across all your hosts, you must use a dimension to filter the metrics into groupings of 5,000 or fewer hosts. A common example of such a dimension is by data center or (in the case of Amazon Web Services) region or availability zone, each of which may have anywhere from 3 to 4,000 hosts each.

This method for breaking down computations also applies to detectors, because each plot line used to compose the detector signal or dynamic threshold must also conform to the time series limit. For example, if you want to ensure that your aggregate free memory across 17,000 hosts exceeds a certain threshold, you first break the metric down into groupings of 3 to 4,000 hosts each (e.g., using `aws_availability_zone:us-east-1a, aws_availability_zone:us-east-1b, aws_availability_zone:us-west-1a,` and `aws_availability_zone:us-west-1c` as your filters, respectively, on plots A, B, C, and D) and then use another plot line to sum up across them, such that E = A + B + C + D. E then becomes your signal for the detector.

Ephemeral Infrastructure

In modern application environments, it is increasingly common to use ephemeral infrastructure—instances that are auto-scaled up or down; containers that are spun up on demand; or immutable infrastructure that is brought up with new code versions and discarded when the next code version is deployed. Although there are many advantages to ephemeral infrastructure, it also comes with new variants of traditional monitoring challenges.

Even More Sources

The first challenge is that the use of ephemeral infrastructure can cause you to encounter the per-plot line limits—see the section "Large Number of Sources"—more quickly, because each metric on a new instance or container is represented in SignalFx as a new time series. This is especially true when you are looking over a longer time range: the shorter the lifespan of your instances or containers and the longer the period of time over which you want to look at their metrics, the more quickly you get to large numbers of time series in a single plot line. The approach to handle this situation is the same irrespective of whether the sources are ephemeral: make sure the groupings of metric time series fit within the limit.

Is It Supposed to Be Down?

The second challenge is that in an environment where components are constantly going up and down, traditional mechanisms for monitoring do not work. For example, manual configuration is required for new elements and any non-reporting of a metric is assumed to be problematic and alert-worthy versus being the expected effect of auto scaling, when, say, an instance is turned down on purpose. Using analytics, however, you can only alert when the non-reporting is unexpected.

The analytics function that helps in this situation is `count`. (Be sure you are selecting the analytics function and not the rollup.) `count` tells you how many time series are reporting a value at a given point in time; if an instance stops reporting a metric—because it has been terminated purposely, for example—then its time series will not be counted.

You can take advantage of this function to learn how many instances are reporting, but you need one more thing: a property that tells you the expected state of the instance. For example, Amazon publishes the state of an EC2 instance (terminated, running, etc.), and SignalFx imports that as `aws_state.` With this information, you set up a plot that uses a heartbeat metric of your choosing (say, `memory.free`); you filter out the emitters that have been purposely terminated (`!aws_state:terminated`); and apply the count function, with a group-by on a dimension that represents a single emitter (e.g., `aws_tag_Name`). This plot then emits 0 or 1, and an alert when the output is 0 tells you that an emitter (the instance) is unexpectedly down.

You can apply this general concept to anything you want; you just need a heartbeat metric that reports regularly; a canonical dimension that represents the emitter or source you care about; and a property on that dimension that denotes the expected state of that emitter.

New Sources in Existing Detectors

The third challenge is the amount of time required for newly created instances' or containers' metrics to be visible to existing detectors. While this typically does not affect detector functioning, in more extreme cases (e.g., when an immutable infrastructure approach is used, where some set of sources goes down completely and is replaced wholesale) your detector may need to account for the gap in data points seen by the detector.

SignalFx detectors periodically update the sources included in its computations. Thus, one approach is to simply use a duration that is long enough to account for that period. A second approach is to treat the data source as if it is aperiodic (see the section "Aperiodic Data") and adjust your detector logic accordingly.

A third approach is to re-save the detector when new sources are added. This forces the detector to update the metric time series included in its computation and therefore ensures that newly created sources are included.

Conclusion

Designed to eliminate complexity and maximize productivity, SignalFx offers tools to expedite creation, deployment, and tuning of alerts. These best practices for creating alert detectors with SignalFx allow your team alert on patterns related to performance before issues emerge and customers are impacted.