

What should be alerted?



taowen

2015-07-26



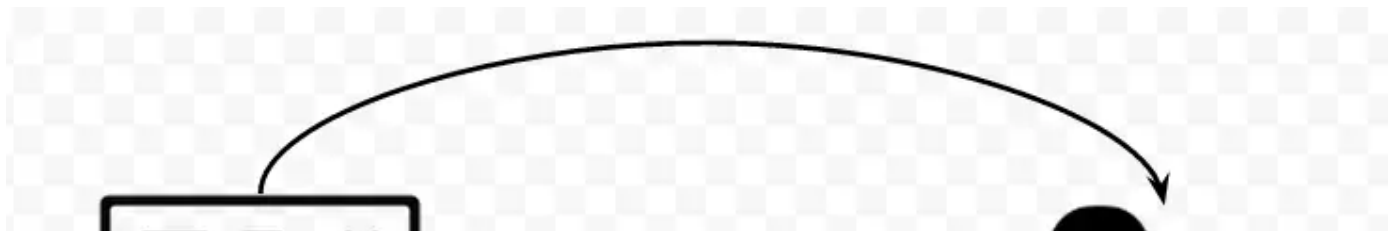
read 9 minute

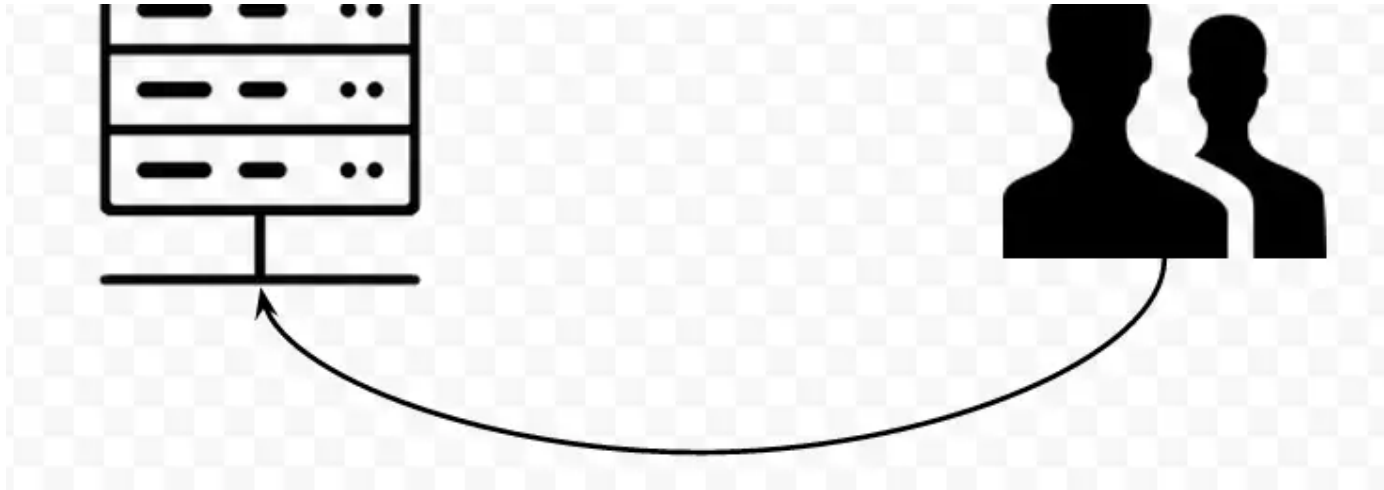
The nature of the alert

Few systems have well-designed alerts. Good alert design is a difficult task. How do you know if an alert you're receiving is bad? How many times have you received an alert and immediately closed it? Are you constantly overwhelmed by these seemingly useless alerts? The most common alert setting: "Alert when CPU usage exceeds 90%." This setting doesn't provide high-quality alerts in most situations.

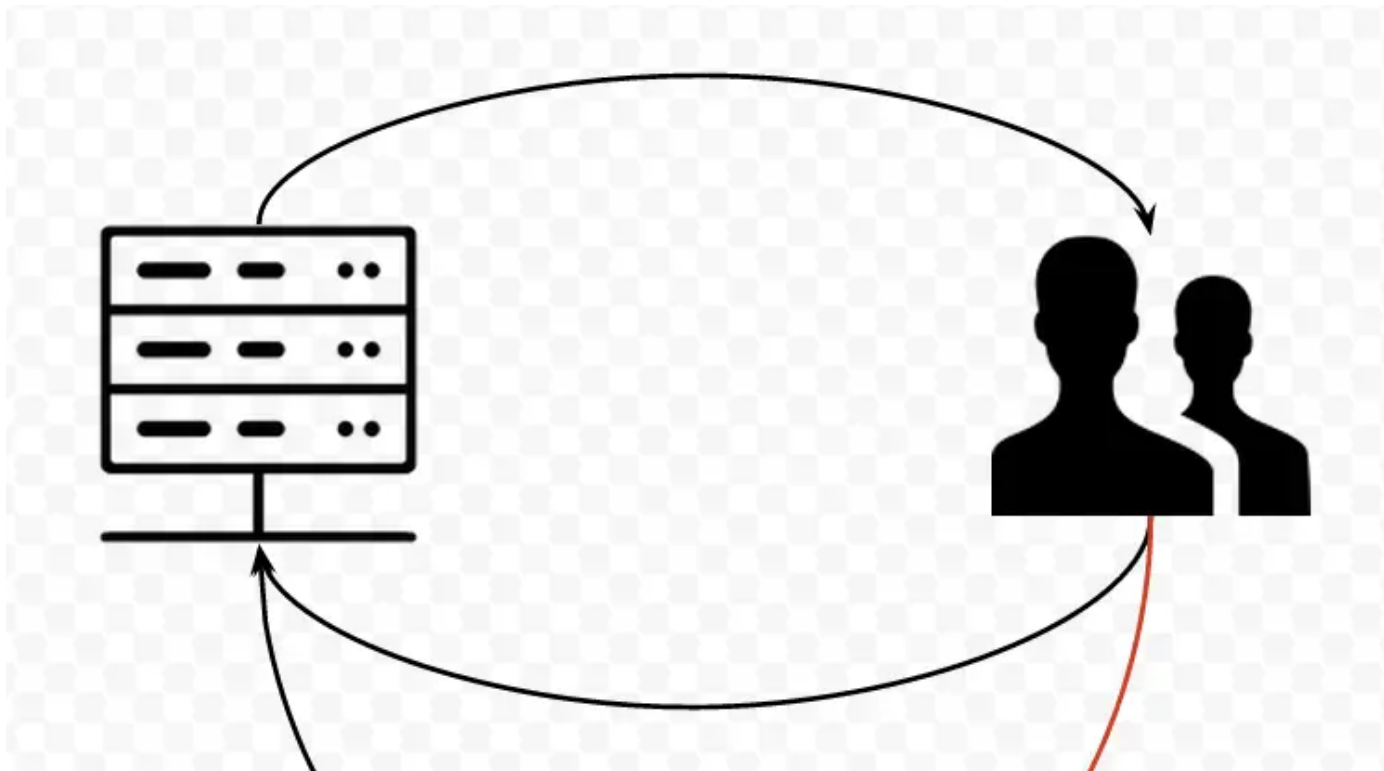
High-quality alerts should be like this: you can immediately assess the scope of impact after each receipt, and each alert requires you to make a graded response. So-called every alert should be actionable.

The essence of the alarm can be illustrated by the following figure:





The server should be designed for such unattended operation. Assuming that all operations and maintenance personnel are on vacation, the service can still run automatically 24/7.





The essence of alerting is "using people as services." When certain tasks cannot be executed programmatically, alerts are used to notify people and allow them to intervene in the system to correct them. An alert is like a service call. If an alert is issued but the recipient does not take any action, it constitutes a DDoS attack, disrupting the lives of operations and maintenance personnel.

Many times, the tasks that alarm notifications require people to perform can truly be automated. For example, if a server crashes, a new one is brought up. In smaller systems, the system might simply be shut down for a short period while a cold standby machine is manually replaced. Larger systems, with more servers, cannot have daily downtimes; hot standby is required, with the system automatically switching to the backup machine. In even larger systems, because switching is so frequent, the removal of failed machines and the maintenance of backup machines become a management burden. This can be integrated with other operations and maintenance processes to create a fully automated system. It's simply a matter of choosing different implementation strategies for different business processing stages. When the business volume is small, using flesh and blood as machines is sometimes more economical. Of course, for the guy who's being treated like a robot, life is indeed a bit unfair.

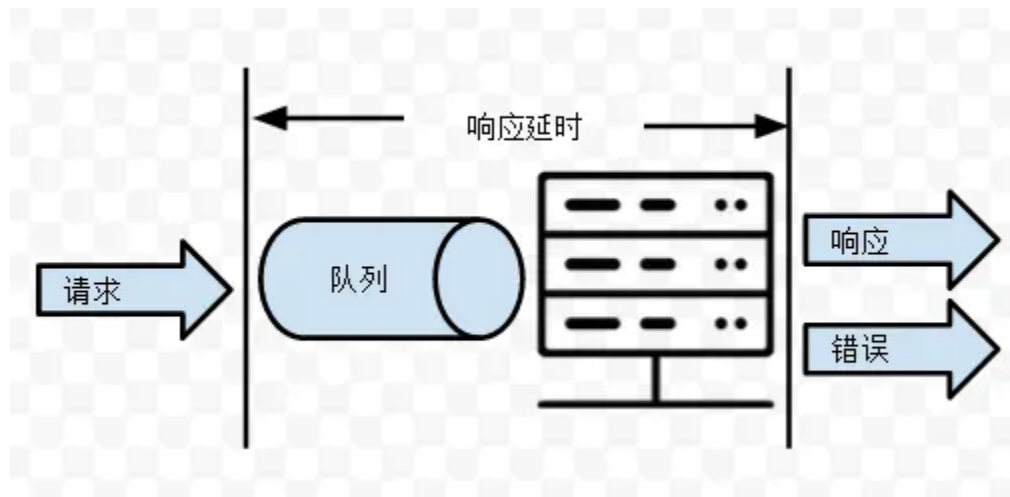
Alarm object

Alarm objects can be divided into two types:

- Business rule monitoring
- System reliability monitoring

For business rule monitoring, consider a game. For example, in DNF, characters with certain equipment have a maximum damage output per hit. Exceeding this limit indicates cheating. Another example is a game called Landlord (Dou Dizhu), where a player's winning streak and daily win rate are capped. Exceeding the average by a significant margin suggests cheating. Business rule monitoring isn't about hardware or software functioning properly. Rather, it's about whether the software is implemented according to business rules and whether there are any vulnerabilities. This can also be understood as monitoring "correctness."

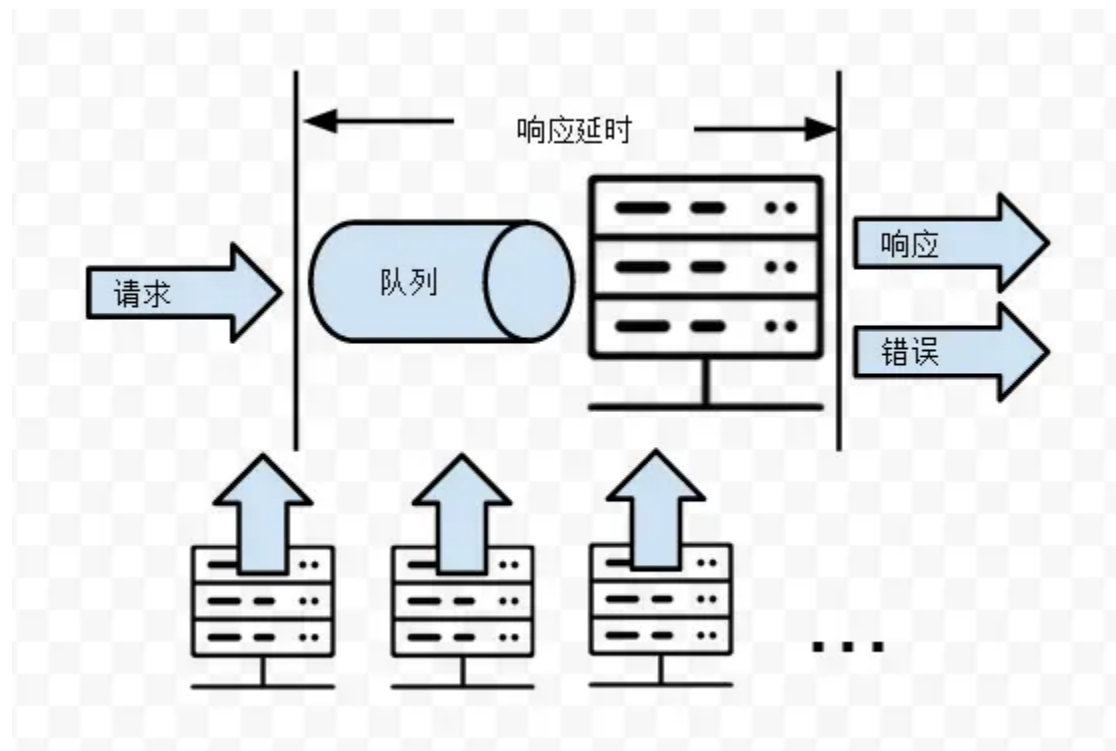
System reliability monitoring is the most common form of monitoring, such as detecting whether the server is down, whether the service is overloaded, etc. For most backend services, the system can be abstractly modeled like this:



What metrics can be collected for such a system?

- Number of requests, request arrival rate
- Number of normal responses, normal response ratio
- Number of error responses, percentage of error responses
- Response delay
- Queue length, waiting time

The reality is that almost no system operates in isolation. Instead, it works like this:

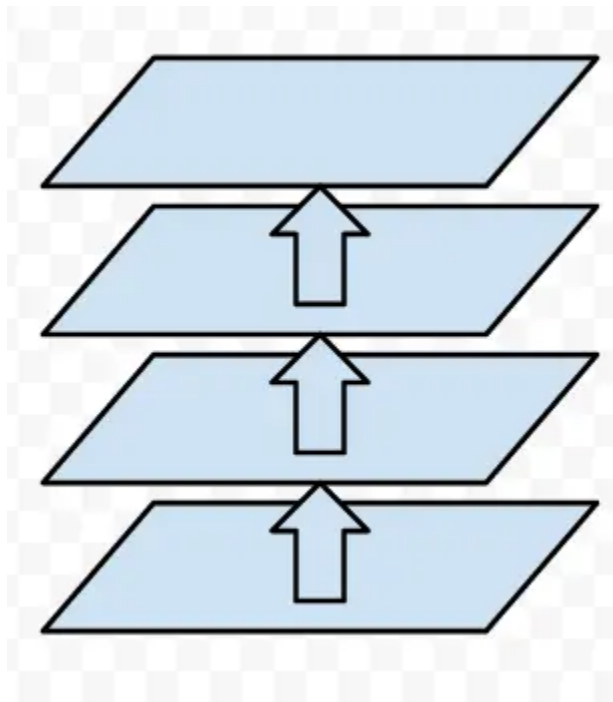


A DB will depend on the underlying CPU, memory, disk and other resources. An HTTP service will depend on the underlying DB service. An application will depend on several underlying RPC services. So there are several more indicators.

- The call volume of resource A (such as CPU usage)
- The amount of resource B used (such as memory allocation and deallocation)
- The call volume of resource C (such as the number of packets sent on the network)
- ...

This hierarchical structure can be generally divided into four layers:

- Product strategy and marketing: They determine the fundamental rate at which requests arrive
- Application layer (more vulgarly called web layer): the top glue
- Service layer: db, various RPC services, and nested services
- Hardware layer: cpu, memory, disk, network



Because of this dependency hierarchy, the resource consumption of the previous layer on the

next layer is translated into the number of requests to the next layer. For example, the amount of DB resources consumed by the HTTP service corresponds to the number of requests the DB service needs to handle. Whether the DB is busy depends on the HTTP service requests, which in turn depends on how many people open the client, which in turn depends on product strategy and marketing activities. This hierarchical structure means that simply tracking a single metric, such as the absolute number of requests, is unlikely to indicate whether a service at that layer has experienced a failure.

With so many layers, and each layer capable of collecting numerous metrics, what metrics should be collected, and what alerting strategies should be used? As mentioned earlier, alerts must be actionable, but in practice, this general requirement alone is often difficult to implement. At the very least, we can mention a few things we shouldn't do:

- You shouldn't let the difficulty of collecting data determine which metric to use for alerting. In many cases, CPU usage might be the easiest metric to collect, but it might not be the most worthy of alerting.
- Don't give operations staff the alerts they want; instead, give them the alerts they truly want. Most of the time, people are giving you a solution. For example, if operations staff tells you they need an alert when the DB process's CPU usage exceeds x%, they'll give you what they believe to be the optimal solution. But what they really want is to know if the DB service is experiencing an anomaly. CPU usage exceeding x% isn't necessarily the best indicator of service anomalies.

Blindly collecting easily accessible metrics and arbitrarily setting threshold alerts are the root causes of most poor alert quality.

Monitoring indicators and strategies

So what metrics should be collected? I believe that most system reliability monitoring has three goals:

- Is the work getting done? Is the system continuing to complete the work it was set to do?
- Is the user having a good experience?
- where is the problem/bottleneck?

The most important and critical issue is the first one, which is the work getting done. For the database, we can collect:

- CPU usage
- Network bandwidth
- Number of db requests
- Number of db responses
- Number of db error responses
- DB request latency

Obviously, to answer whether a database has completed its designated work, the two indicators that should be paid more attention to are:

- The absolute number of db requests
- The ratio of correct db responses to the number of requests

These two metrics are more indicative than simply collecting CPU usage. Request volume and correct response percentage can be used to reflect the performance of services at all levels, not just databases. Examples include HTTP request count (compared to correct HTTP response count) and app open count (compared to the number of online users recorded by the server).

Why can't CPU usage explain the problem? Most of the time, we don't care about the CPU itself, but rather the services that use the CPU as a resource. So CPU usage is just the number of requests for a resource. A concept related to the number of requests is saturation (upper limit). When the upper limit is reached, processing begins to queue, latency begins to increase, and error rates begin to rise. So can CPU usage explain the upper limit? The upper limit of CPU usage is 100%, so wouldn't it be reasonable to start alerting at 90%? After all, a 100% CPU is almost equivalent to the database being unable to process requests normally.

This approach of using the underlying resource call volume to assess whether the upper limit has been reached has two fundamental flaws:

- You cannot know to what extent the upper-layer services can utilize the underlying resources
- The saturation of underlying resources may not be easy to measure

Specifically, whether a database can truly utilize 100% of the CPU is uncertain. If requests are locked or sleep, the CPU may never reach 100%. 90% is probably the limit. Furthermore, modern CPUs have multiple cores. If request processing can only utilize a single core, and processing jumps between multiple cores, a single core will never maintain 100% utilization.

The CPU might have a 100% upper limit. However, for many non-hardware services, such as a login service that relies on a database, it's difficult to measure the number of different SQL combinations the database can process per second. Unlike disks, which have an absolute MB/s limit for comparison, it's not possible to measure the number of different combinations per second.

Another drawback of measuring low-level resource usage is that you can't enumerate all dependent resources. So, rather than indirectly monitoring the health of upper-level services through low-level resources, it's better to directly measure whether work is getting done.

For the second question, is the user having a good experience? The metrics that can be collected are

- Average queuing time, average total response delay
- 99/95/90 percentile queue times, 99/95/90 percentile response delays

The user here does not necessarily refer to a person or a player, but may be a service caller on the upper layer or another system.

The third issue is fault location. If done manually, the most common approach is to receive an alarm, log in to a CRT, and start typing various commands to find the cause. For the system, the most appropriate approach is not to execute a bunch of commands after a problem occurs, but to:

1. Each level warns itself
2. The top-level service generates an alarm, triggering the automatic positioning program
3. According to the service dependencies and approximate time range, locate the correlation between alarms to find the problem or bottleneck.

Of course, the actual situation is very complex. Many causes and effects are causally related to each other. It is difficult to clearly determine whether the two alarms represent two different phenomena, or whether they represent the same cause and phenomenon.

From the perspective of alerting algorithms, it's very easy to alert on successful request rates or average response latency. Static thresholds are often looked down upon as simplistic, but most alerts can be solved using static thresholds.

Theory and reality

So, do alerts require complex algorithms? My view is that if the correct indicators are collected, complex algorithms are unnecessary; static thresholds can be sufficient. However, there are at least three situations where algorithms are necessary:

- Unable to directly collect the number of errors: automatic classification of error logs is required
- The request success rate cannot be directly collected: anomaly detection is required for the absolute value of the number of requests or responses.
- Only the total number is available, and the proportion of each subdivision item cannot be collected: the algorithm fitting of the participating factors is required

These three items actually relate to one theme. When you can't directly obtain the indicators needed for alerts, things become much more complicated. An analogy is Kepler 452b, recently announced by NASA as Earth's twin. If our probes could reach 1,400 light-years away, discovering it would be very easy. Precisely because direct data is so difficult to obtain, scientists rely on the brightness changes caused by planets blocking their stars (the so-called occultation method) to discover these distant planets.

The difficulty in collecting the required metrics can be due to several factors. One reason is that collecting metrics is very resource-intensive. For example, obtaining the CPU usage of each MySQL query. Tracking every request is impossible. This is where algorithms come in handy. You can take a closer look at this video by vividcortex: <http://www.youtube.com/watch?v=szAfGjwLO8k>

In many cases, the difficulty in collecting metrics stems from communication issues caused by the D/O separation. The metrics needed by operations and maintenance require development to track, and the areas where development tracks data require operations and maintenance to generate alerts. This often leads to a situation where whatever metrics are available are used.

For example, while there may not be a count of request and response errors, errors are generally recorded in error logs. The speed at which the error logs are updated can provide a rough idea of whether a problem has occurred. This introduces a very difficult log classification problem: which logs represent normal activity, which logs represent abnormalities, and what types of abnormalities are they? Summo Logic (<https://www.sumologic.com/>) has excelled in this area with algorithms . Why are opsdev (a mocking term for devops) companies so enthusiastic about algorithms? For them, the benefits are obvious: fewer changes required for customers, lower integration costs, and a wider customer base. But is using machine learning algorithms to mine massive logs really the best way to answer the question, "Is the work getting done?" Clearly not. It's like using a cannon to suppress a mosquito. Logs exist to solve problems, not to make the problem of using "them" effectively out of the vast amount of logs.

The third situation is when there's no way to collect data on request success rates, and only the absolute volume of successful processing can be measured. If only this type of data is needed for alerting, a simple static threshold cannot be used. For latency, you can generally set an upper limit for latency that's acceptable for the business. For success rate, you can also set an upper limit for acceptable success rates. However, for absolute processing volume, it's impossible to simply compare a static threshold to determine whether it's normal or abnormal.

Before discussing how to implement it, let me emphasize two points:

- The number of successful processes isn't the best indicator of work getting done. Instead of wasting time and effort developing algorithms, it's better to just collect success rate metrics.
- The number of successfully processed requests also depends on the number of requests. And the number of requests ultimately depends on the upper-level services. You're a DBA and notice a sharp drop in the number of requests processed per second by the database. Does this mean the database is down? Or the app? It could be either... The top layer is

product and marketing. You notice a decrease in registrations for a business compared to a few days ago. Does this mean there's a problem with the registration service? It could also be that the product is terrible and no one is playing the game. Or it could be a marketing tactic: no more gold rewards, and players are losing their enthusiasm.

Anomaly Detection

How can we detect anomalies when we only have the number of requests, no reference upper limit (saturation), no success rate, and no failure rate?



The yellow line in the above figure is yesterday's value, and the green line is today's value. Most service monitoring graphs look like this. Four ideas can be drawn:

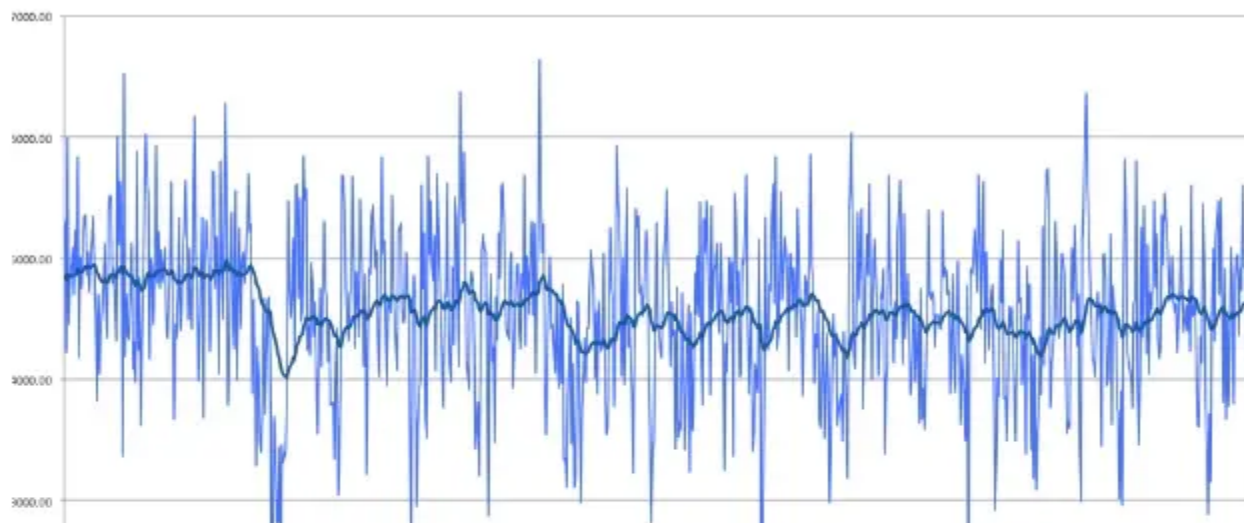
- Curve smoothness: Failure is generally a disruption to the recent trend, which is visually not smooth.
- Time periodicity of absolute values: the two curves almost coincide
- Time periodicity of fluctuations: Assuming that the two curves do not overlap, the fluctuation trends and amplitudes at the same time point are similar
- There is a considerable length of pit: when the curve begins to rise back to the historical range, it can generally be confirmed that this period of time is a real failure

Expanding from these four intuitions, we can derive various algorithms, some complex and some simple. The algorithms we will discuss below are very simple and do not require advanced mathematical knowledge.

Detection based on curve smoothness

This detection is based on a recent time window, such as one hour. The curve tends to follow a certain trend, but new data points disrupt this trend, making the curve less smooth. In other words, this detection exploits the temporal dependency of the time series: T has a strong trend dependency on $T-1$. From a business logic perspective, if many people log in at 8:00, then there's a high probability that many will also log in at 8:01, because the factors that attract people to log in have strong inertia. However, if many people log in at 7:11, then there's also a strong inertia of many people logging in at 8:11.

To generate alerts based on recent trends, you need to fit the trend of the curve. There are two fitting methods: moving average or regression. These two fitting methods have different biases.





www.vividcortex.com

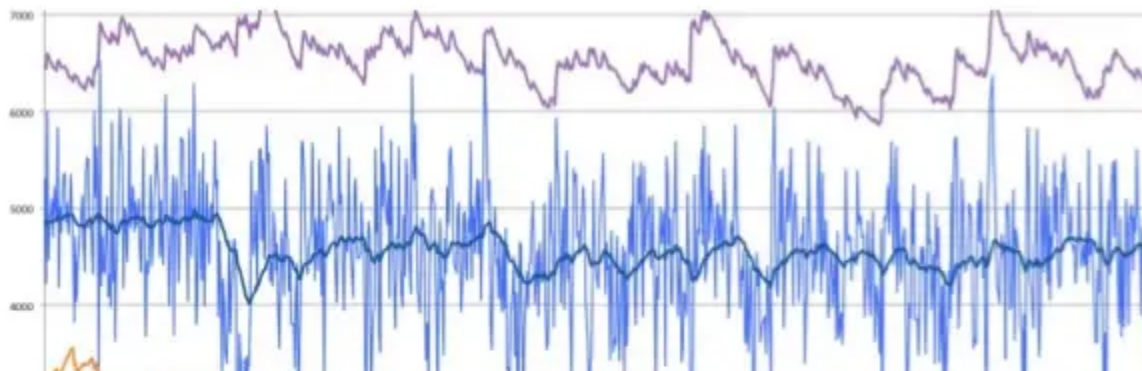
| info@vividcortex.com

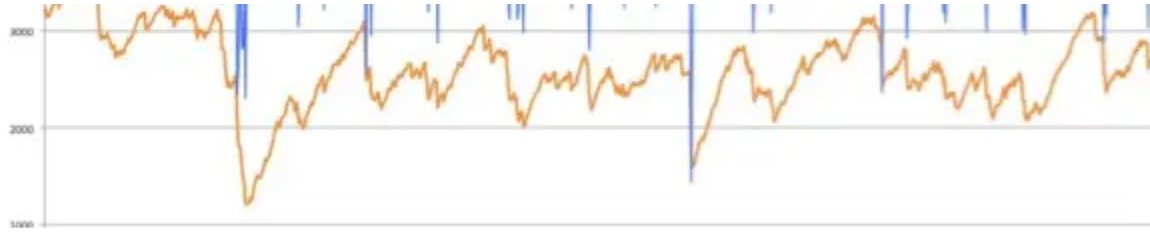
VividCortex

This is a diagram of a moving average algorithm called exponentially weighted moving average. Its calculation is very simple

$$S_t = \alpha x_t + (1-\alpha) S_{t-1}$$

Where x is the actual value, and s is the average calculated by the EWMA. In other words, the average of the next point is the average of the previous point, corrected by the actual value of the current point. The proportion of this correction depends on the decay factor of α . Visually, this indicates whether the EWMA curve closely follows the actual curve, that is, its smoothness.






EWMA Chart

www.vividcortex.com

| info@vividcortex.com

 VividCortex

Once we have the average, we can calculate the variance. Multiplying the variance by a certain factor gives us the tolerance range for the amplitude. Comparing the actual value to see if it exceeds this range tells us whether an alert is warranted. Exceeding the upper bound could indicate a sudden surge in user volume. Exceeding the lower bound could indicate a marketing campaign has ended, users have quickly left, or a fiber optic cable has broken, causing players to disconnect. For more details on the EWMA algorithm, follow Baron Schwartz (<http://www.slideshare.net/vividcortex/statistical-anomaly-detection-fo...>)

The moving average assumes that the curve is tending towards history. If the momentum of the curve is rising, then it believes that the next point should start to decline. Regression believes that the curve is tending towards the future. If the momentum of the curve is rising, then it believes that the next point should maintain this upward momentum. There are more complex models that combine moving average and regression. Regardless of the algorithm, it is impossible to accurately predict the next 10 minutes based on the past 10 minutes. If this prediction can be accurate, then the stock god would have been born long ago. Using the moving average may mask the decline caused by the failure (because its bias is downward). If

regression is used, it is possible to mistake the failure to rise so fast as a failure (because its bias is upward).

Another flaw in this algorithm, which calculates variance based on recent trends, is that when the first few points experience significant fluctuations, the variance value is inflated. This obscures subsequent faults, making it impossible to detect consecutive fault points. In essence, the algorithm has no concept of normality and assumes that past history is normal. If a fault has occurred for the past few minutes, then the fault curve is considered normal.

In actual use, it is found that the advantages of this algorithm based on curve smoothness are

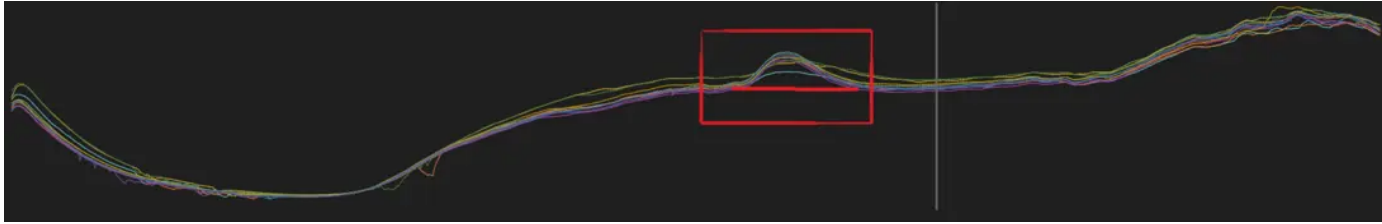
- Depends on less data, only recent history is needed, and does not rely on periodicity
- Very sensitive. If the historical fluctuation is small, the variance is small and the tolerance range of fluctuation will also be very small.

The disadvantages are also obvious

- Too sensitive and prone to false alarms. Because the variance increases with the introduction of abnormal points, it is difficult to use a strategy such as only alerting when three consecutive points are present.
- The business curve may have regular steep increases and decreases.

The best approach is to avoid using a single curve as an alert. Instead, combine several related curves. If smoothness is disrupted and the trend deviates from business trends (for example, a decrease in online users and an increase in login requests), it can be considered a business failure.

Time periodicity based on absolute values



In the above figure, different colors represent curves on different dates. Many monitoring curves have a daily periodicity (lowest at 4 am, highest at 11 pm, etc.). One of the simplest algorithms that uses time periodicity is

$\min(14 \text{ days history}) * 0.6$

Take the minimum value of the 14-day historical curve. How do you do this? For the point at 12:05, there are 14 days corresponding to it. Take the minimum value. For the point at 12:06, there are 14 days corresponding to it. Take the minimum value. This will create a single-day curve. Then multiply this curve by 0.6. If the curves for several days fall below this reference line, an alarm will be issued.

This is actually an upgraded version of static threshold alerting: dynamic threshold alerting. Previously, static thresholds were based on random guesswork based on historical experience. This algorithm uses historical values at the same point in time as a basis to calculate the least likely lower bound. Furthermore, the threshold isn't unique; instead, it has one for each point in time. If there's one lower bound every minute, there would be 1,440 lower bounds per day.

In practice, the value of 0.6 should be adjusted accordingly. A serious issue is that if there were downtime or failures in the 14-day history, the minimum value will be affected. In other words, the historical data cannot be considered normal; instead, the historical data should be removed to eliminate outliers before calculation. A pragmatic approximation is to take the second smallest value.

To make alerts more precise, the difference between the actual curve and the reference curve can be accumulated. This is the area of decline relative to the reference curve. If this area exceeds a certain value, an alert is triggered. For a deep decline, a few accumulated points can be enough to trigger an alert. For a shallow decline, even a few accumulated points can also trigger an alert. In other words, if the price drops significantly all of a sudden, it's likely a fault. If the price deviates from the normal value for a long period of time, it's also likely a problem.

advantage:

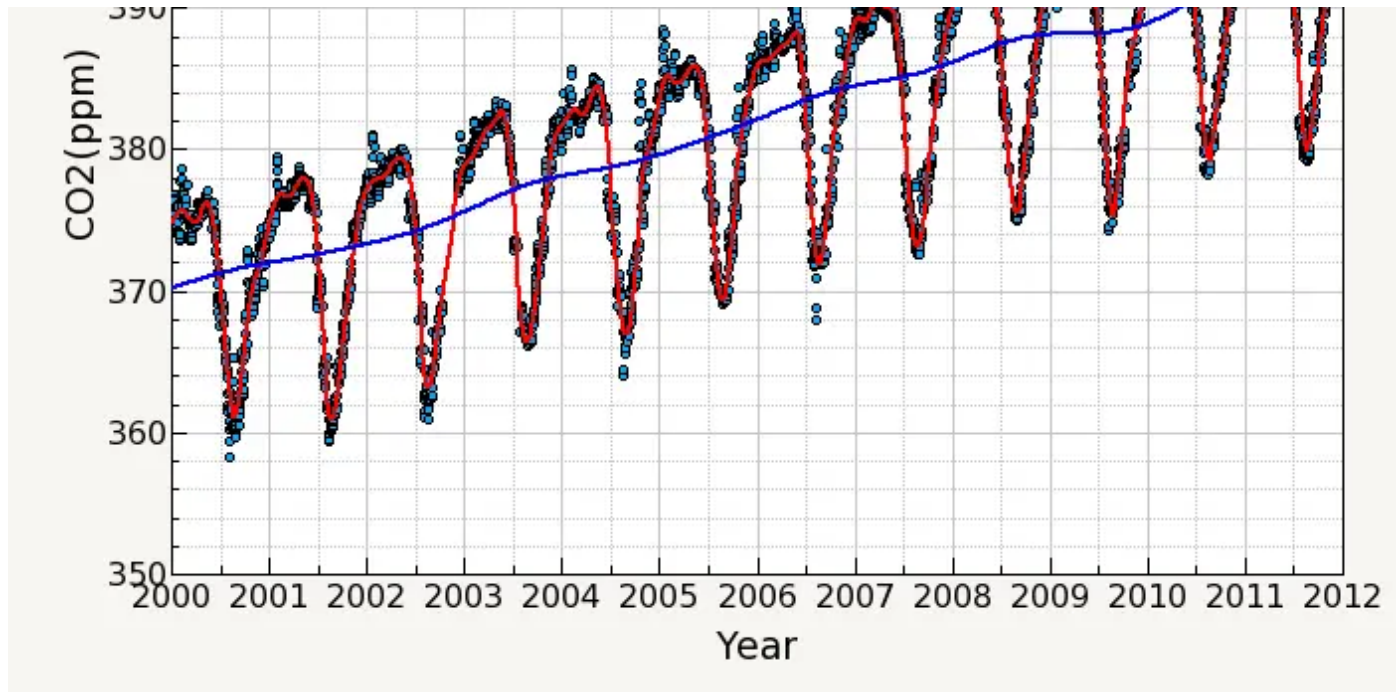
- Simple calculation
- It can ensure that major faults are discovered. If an alarm is issued, it must be a major problem. You can call directly

shortcoming:

- Relying on periodic historical data, the computation is intensive and it cannot generate alerts for newly added curves.
- Very insensitive, small fluctuations cannot be detected

Amplitude-based temporal periodicity





Sometimes, a curve exhibits periodicity, but the superposition of two periodic curves does not overlap. For example, in the figure above, the overall trend of the curve is online. When the two periodic curves are superimposed, one will be significantly higher than the other. In this case, using absolute value alarms can be problematic.

For example, today is October 1st, the first day of the holiday. The historical curve for the past 14 days is bound to be much lower than today's curve. But if a minor malfunction occurs today, causing the curve to drop, it's still significantly higher than the curve for the past 14 days. How can such a malfunction be detected? An intuitive explanation is that the two curves, while not at the same height, are "similar in appearance." So how can we exploit this similarity? The key is amplitude.

Rather than using the value $x(t)$, it's better to use $x(t) - x(t-1)$, essentially converting the absolute value into a rate of change. This rate can be used directly, or $x(t) - x(t-1)$ divided by

$x(t-1)$, giving a ratio of the rate to the absolute value. For example, if 900 people were online at time t and 1000 were online at time $t-1$, we can calculate that 10% of the users were offline. Is this offline rate high or low at the same time in history? Then proceed as before.

There are two practical techniques: you can use $x(t) - x(t-1)$ or $x(t) - x(t-5)$. The larger the span, the more likely it is to detect slow declines.

Another trick is to calculate $x(t) - x(t-2)$ and $x(t+1) - x(t-1)$. If both values are abnormal, they are considered to be true anomalies, which can avoid the data defect problem of a single point.

advantage:

- More sensitive than absolute value
- Utilizes time periodicity to avoid the periodic steep drop of the business curve itself

shortcoming:

- The original curve is required to be smooth
- The time points of periodic steep drops must coincide, otherwise false alarms will occur.
- Calculating by percentage is prone to false alarms during off-peak periods
- A sharp drop does not necessarily mean a failure. A surge followed by a fall due to fluctuations in upper-layer services often occurs.

This anomaly alerting algorithm is quite good, but it also has a number of shortcomings. Therefore, some workarounds are available to make it work. To avoid false alarms during off-peak periods based on amplitude percentages, a lower limit on the absolute amplitude can be added. From a business perspective, this means that small fluctuations, while large in relative proportion, are not a problem if the absolute impact is small. For issues with surges followed by declines, the situation can be assessed and blocked for a period of time after the surge.

Abnormal judgment based on curve recovery



When we look at Figure 2, we're even more certain of a fault than when we look at Figure 1. Why? Because there's a clear rebound in Figure 2. The algorithm, like the human eye, can predict a fault more accurately if we wait a few more time points and see that the curve rebounds. However, this type of anomaly detection based on rebounds doesn't offer much of an "alarm" mechanism. The purpose of an alarm is to encourage human intervention to help the curve rebound. If the curve has already started to rebound, wouldn't issuing an alarm be a case of hindsight?

The significance of this type of testing lies in confirming machine-generated alarms. When we need to calculate false alarm and missed alarm rates, re-running the algorithm with a different perspective can reveal many issues with the original algorithm. It can also be used semi-

automatically to build a sample library of historical faults. This sample library can then serve as a training set for more complex machine learning algorithms.

Summarize


Key take away

- High-quality alerts are actionable
- The difficulty of data collection should not determine which indicators you use for alerting.
- Don't just follow the warnings of others, but make "real" useful warnings: especially CPU usage warnings.
- is work getting done: number of requests + success rate
- is the user having a good experience: response delay
- As long as the right indicators are collected, most of the time alarms do not require complex algorithms
- Algorithm-based anomaly detection: The algorithm is not difficult and can be done if necessary

[Alert](#) [Architecture](#)

 Like 20

 Collection 52

 share

reads 17.6k • Published on July 26, 2015

[report](#)



taowen

4.1k reputation • 1.4k followers •

Go developers please join us, Didi Chuxing Platform Technology
Department, taowen@didichuxing.com

Follow the author

« Previous

Next Post »

[The Power of Ten – Rules for Developing Safe...](#) [Database Consistency and Leaky Abstraction](#)

Quotes and Comments

Recommended Reading



Can R&D effectiveness be measured?

taowen • likes 3 • views 2.9k



Nightingale open source monitoring, template function overview

SRETALK • Read 1.3k



Innovative practice of number generation system: design of lucky code for weekly game

vivo Internet Technology • Reading 843



得物灵犀搜索推荐词分发平台演进 3.0

得物技术 • 阅读 823



什么是数据聚合 (Data Aggregation) ?

镜舟科技 · 阅读 715



文档：架构师的“编程语言”

葡萄城技术团队 · 阅读 687



数据湖典型架构解析：2025 年湖仓一体化解决方案

镜舟科技 · 阅读 684

4 条评论

得票

最新



撰写评论 ...



提交评论

评论支持部分 Markdown 语法： ****粗体**** *_斜体_* [链接](http://example.com) `代码` - 列表 > 引用。你还可以使用 @ 来通知其他用户。



knightuniverse：非常专业啊，干货

👍 · 回复 · 2015-08-03



huandu：感觉这些理论和算法似乎都跟业务没有强绑定，不知道是否有现成的开源项目可以学习一下实现方法，求指点~

👍 · 回复 · 2015-11-08



youzhengchuan：“如果告警了，但是收到告警的人并不需要做任何处理，那么这就是一种DDoS攻击，攻击的是运维的幸福生活。”——工作这些年受到的DDOS攻击都可

以打穿整个地球了。

👍 · 回复 · 2017-05-11



不用Google你有bing啊： 干货很多，谢谢分享。

“第三类情况是没有办法采集到请求成功率，只能对绝对的处理成功的量。”

其实除了这种情况，有时候能采集到成功率，但是请求量依然是需要的。比如请求量下跌严重，但是成功率保持很高，依然需要去action。这可能和我们的业务有关，如果客户那边请求降低，我们也会配合他们排查的。

最近一直在做请求量曲线的异常点检测，暂时还没有找到啥有效的方法.....

👍 · 回复 · 2018-07-09

©2025 taowen

除特别声明外，作品采用《署名-非商业性使用-禁止演绎 4.0 国际》进行许可

 使用 SegmentFault 发布

SegmentFault - 凝聚集体智慧，推动技术进步

服务协议 · 隐私政策 · 浙ICP备15005796号-2 · 浙公网安备33010602002000号