NODE.JS MONITORING, ALERTING & RELIABILITY 101

A detailed guide for building and monitoring reliable applications

From the Engineers of trace by RisingStack

TABLE OF CONTENTS

INTRO

In the past couple of years, most of the businesses moved to the internet. These companies develop and maintain systems that have to handle a huge amount of visitors and millions of transactions. It's crucial for these internet companies to serve their customers' needs in a reliable way, especially during peak hours (like a Black Friday sales rush). For most enterprises, a service disruption is unacceptable because it leads to million dollars in lost revenue.

At RisingStack, we are responsible for many critical systems. Our intention with this book is to share the most important aspects of building a reliable service.

This guide contains our experiences, as well as many ideas collected from different sources like conference talks and books. However, keep in mind that there are no two identical systems. Some of these advices might make no sense in your situation.

The first part of this white paper introduces common techniques to measure reliability and understand its cost and other aspects. The second chapter introduces the reader to monitoring and alerting basics, as well as paging and postmortem best practices.

The third part focuses on technologies and patterns which help to improve the reliability of your services. The fourth chapter focuses on setting up and using an open-source Node.js monitoring solution called Prometheus.

ABOUT THE AUTHOR

Peter Marton (@slashdotpeter) is the co-Founder and CTO of RisingStack, currently working on Trace — a Node.js Performance Monitoring and Debugging tool designed for understanding and troubleshooting Node.js Applications. **Peter is an expert of Node.js, Kubernetes & Microservices.**

ABOUT RISINGSTACK

RisingStack enables amazing companies to succeed with Node.js and related technologies to stay ahead of the competition. We provide professional Node.js development, consulting & training services.

CHAPTER I. : UNDERSTANDING AND MEASURING RELIABILITY

FIGURING OUT YOUR RELIABILITY INDICATORS

To determinate the reliability of your system, first you need to define what our service quality requirements are, and what is acceptable for your customers. After defining your needs, you have to measure how well your system fulfills these requirements.

Based on these measurements, you can agree upon a minimum quality which can be expected from your application. These requirements may vary and need to be aligned with your business. If you determine them correctly, they can drive your product development and provide a basis of a contract with your customer.

There are three kinds of reliability indicators. Let's see what they mean and what can you use them for.

Service Level Indicator (SLI)

A Service Level Indicator (SLI) is a carefully defined quantitative metric of some aspect of the level of service that is provided.

A Service Level Indicator in your system can be for example the uptime percentage of your app or the 99th latency of the service that you provide.

Service Level Objective (SLO)

The Service Level Objective (SLO) is a goal for the SLI. It defines what's acceptable and what's not. (It's different for most of the services.)

For example, a Service Level Objective can be the frequency of successful probes of your system, or the percentage of requests where the 99th latency is below 250ms. With the SLO, you can calculate the service availability rate or measure whether or not a service has a significant issue.

Based on the SLO, the Stakeholders can decide to put more effort into reliability improvements or even hold back releases.

Stricter SLO requirements always come with a higher cost of development, maintenance, and resource allocation. For example, to match higher standards, you need to implement advanced architecture reliability patterns and provision more resources to improve the redundancy of your system.

Service Level Agreement (SLA)

The Service Level Agreement (SLA) is a contract between the service provider (in this case probably you) and the customer. It usually includes the consequences of meeting or missing your SLA. In a nutshell, the Service Level Agreement is an SLO + penalties.

To avoid penalties, you should always have stricter internal SLOs than the ones you agreed upon with your customers in you SLA. For example, with 99.8% SLO, you shouldn't contract for 99.9% SLA.

CALCULATING YOUR ERROR BUDGET

The Error Budget tells you how often you can miss your objective (SLO) in a given period before paying for penalties (thanks to violating your agreement, the SLA). It also tells how much did you affect your customers' user experience in a certain period.

For example, with a 99.9% uptime objective, you are allowed to have 43.2 minutes of downtime per month, and still, meet the requirements you agreed upon. How was this calculated?

A month (30days) takes up 43200 minutes. If you agree to provide a service with 99.9% uptime, you are allowed to have outages in 0.1% of the time.

43200 * 0.001 is 43.2 minutes.

Let's see the following Error Budget calculations:

1 - uptime = Error Budget
1 - 99.9% uptime = 0.1% Error Budget
1 - 99.99% uptime = 0.01% Error Budget

The Error Budget should be taken account in your development processes and business decisions as well.

The Cost of Nines

You can often hear people discuss nines when availability is the topic. What does it mean when a service provides three nines (99.9%) or five nines (99.999%) availability?

You might think that the availability difference between a three and five nines service is negligible, but in practice, it's huge! Especially if we are talking about the effort and cost needed to run these highly resilient systems.

A three nines service

A three nines service that has 99.9% uptime leads to 0.1% error budget. It means that before you violate your SLO, you can have:

- **43.2 minutes of downtime** (in a 30 day period)
- **2.16 hours of downtime** (in a 90 days period)
- **8.76 hours of downtime** (in a 365 days period)

A five nines service

With a five nines service, (99.999% uptime and 0.001% error budget) the difference is shocking.Before you violate your SLO you can have:

- **25.9 seconds of downtime** (in a 30 day period)
- **77.8 seconds of downtime** (in a 90 days period)
- **5.26 minutes of downtime** (in a 365 days period)

Only 25.9 seconds, let that sink in.

Can you or your monitoring product respond that fast to an ongoing incident?

Error Budget in practice

The Error Budget helps you to know how much you can miss your objectives (SLO) before you violate your agreement (SLA). But how can you use it for making product and business decisions?

For example, when you use up your monthly error budget, you should consider to hold back a release as shipping new piece of code or configuration always brings a risk of failure.

The Error Budget can also help to optimize your resource usage. For example, when your uptime is much better than expected, maybe you should revisit your redundant resources. The Error Budget can also help the Stakeholders decide between including more features or more reliability focused developments into the product.

To summarize, Error Budget can help to answer the following questions:

- Should I release? Did I use all of my error budget for this month?
- Should I overprovision my resources or can I go hotter?
- Should I write a new feature or improve reliability?
- Should we do more testing?

The use of an Error Budget resolves the fundamental conflict of incentives between development and SRE (Site Reliability Engineering) because the goal is no longer "zero outages." As we mentioned earlier, increasing reliability usually costs incrementally more. The amount comes from:

- The cost of redundant resources.
- The opportunity cost of engineers building failover solutions instead of user-facing features.

Target level of availability

The acceptable availability level of a service strongly depends on your business and customer needs, but you should bear in mind that consumer devices like smartphones, WiFi networks, power sources, and ISPs have much lower availability than 99.999%. This is why users usually cannot make a difference between 99.99% and 99.999% availability. They cannot make a difference between WiFi connectivity and services disruptions.

To calculate the target availability, Take the following criteria into account:

- What level of service will the users expect?
- Does this service tie directly to revenue (either your income or your customers' revenue)?
- Is this a paid service, or is it free?
- If there are competitors in the marketplace, what level of service do those competitors provide?
- Is this service targeted at consumers (B2C), or at enterprises (B2B)?
- How much would it cost to increase availability and how much would it increase the revenue?

CHAPTER II. : MONITORING BASICS

"To observe and check the progress or quality of (something) over a period of time; keep under systematic review." - Monitoring, Oxford Directory

Under the term "service monitoring", we mean tasks of collecting, processing, aggregating, and displaying real-time quantitative data about a system. To analyze the data, first, you need to extract metrics from your system - like the CPU usage of a particular application instance. We call this extraction instrumentation.

You can instrument your system manually, but most of the available monitoring solutions provide out of the box instrumentations. In many cases, instrumentation means adding extra logic and code pieces that come with a performance overhead. However, with monitoring and instrumentation, you should aim to achieve low overhead, but it doesn't necessary mean that a bigger performance impact is not justifiable for better system visibility.

It's worth to mention that logging is not equivalent with monitoring. Logs are immutable entries with a timestamp, from which you can extract metrics and monitor their quantities over time - like the number of error logs during a given period.

The most common use cases of monitoring your infrastructure are discovering trends, comparing metrics over time, building dashboards for real-time insights, as well as debugging and alerting based on your metrics.

THE FOUR SIGNALS

Every service is different, and you can monitor many aspects of them. Metrics can range from low-level resources like CPU usage to high-level business metrics like the number of signups. We recommend you to watch these signals for all of your services:

- **Error rate:** Because errors are user facing and immediately affect your customers.
- **Latency:** Because the latency directly affects your customers.
- **Throughput:** The traffic helps to understand the context of increased error rates and the latency too.
- **Saturation:** It tells how "full" your service is. If the CPU usage is 90%, can your system handle more traffic?

BLACK BOX AND WHITE BOX MONITORING

We can differentiate two kinds of monitoring: black box and white box monitoring. Both of them are necessary to thoroughly observe a system.

Black Box Monitoring (probing)

We use the term Black Box Monitoring (or probing) when you observe a system externally. As the two systems (the monitoring & the monitored) are independent of each other, there is a smaller chance for shared failures.

Uptime checking is a good example for Black Box Monitoring. In this case, an external service probes your system on a public interface. Checking can be stateless (e.g., periodically calling HTTP endpoint), but for stateful services, you may want a stateful probe with real data writes and reads.

It's also possible to check your system's status from different locations around the globe and gather detailed information about availability and latency of an application.

White Box Monitoring

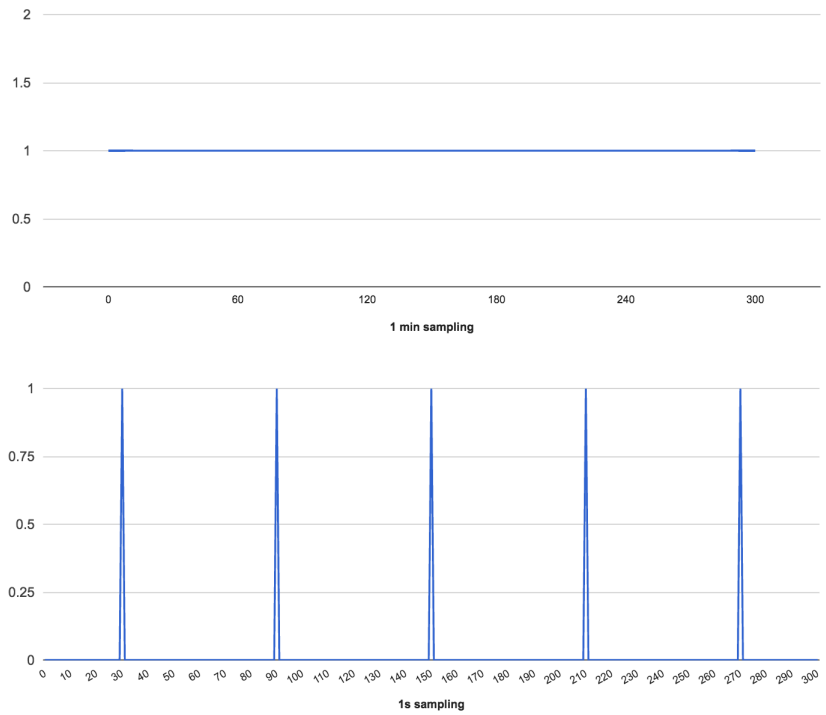
We use the term White Box Monitoring when your metrics are provided by the system you monitor via instrumentations. This kind of monitoring is crucial for debugging and gathering data about memory usage, the heap size, and so on.

As White Box Monitoring metrics are provided by the running system itself, there is a chance to lose access to data during an incident.

SAMPLING RATE

As most of your metrics are continuous data, you need to sample them. With sampling, you are losing accuracy, which means that the data can be misleading or even prevent you from discovering the root cause of a problem. More frequent sampling provides accurate results, but the overhead can be significant, so you have to choose a reasonable sampling rate.

More frequent sampling puts a greater performance overhead to the observed application and also costs more since you have to store and process these samples. Check out the following pictures of the same data with different sample rates:



Choosing the proper sampling rate is difficult because you have to take into account the monitored resources and your monitoring tools' capabilities as well. I think it's safe to say that to monitor the number of running instances in your application, you can choose a lower sampling rate (like one minute) while monitoring the CPU usage needs higher frequency.

Metrics aggregations

Another important topic is selecting the aggregation method to sample, analyze and visualize your metrics data.

You can use many aggregation methods to sample your response times. For example, you can look for averages, medians, 95th percentiles and 99th percentiles (which means 95 or 99 percent of the requests were served below that time), but keep in mind that these contain very different information about your latency.

For example, the 95th and 99th response times are useful metrics to measure general user experience but doesn't provide any information about the slowest response times.

Be careful when you process and visualize your already sampled metrics! Some aggregations are not compatible with each other, for example, getting a median of the last hour will give a false result if you use a larger sampling period or sample with 85th aggregation.

ALERTING PHILOSOPHY

In case you need any help with Node.js, feel free to ping me at

peter@risingstack.com

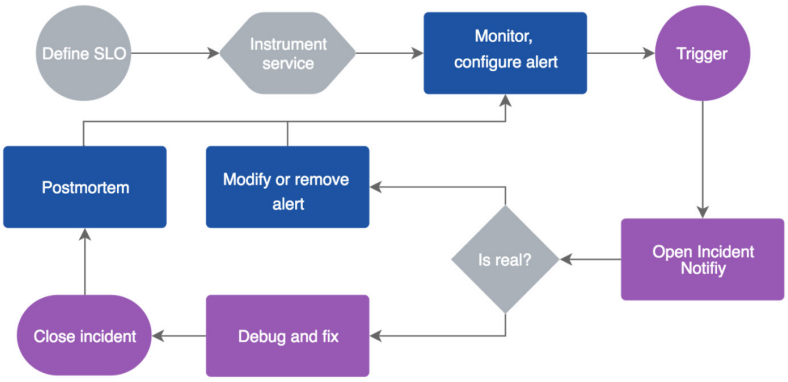
Alerting is a process that matches rules set up by you against your metrics and sends out notifications via one or multiple channels when your criteria match. Alerts are essential for running a reliable system. They help you to spot issues rapidly in a system, and address or minimize them before they become disastrous.

Setting up alerts can be a complex process and properly configuring them needs time.

There's an important thing you have to know about alerting: You can never reach the point when everything works perfectly, where you never have to touch your alert configurations ever again. Your services, business, and SLOs change over time, so you need to re-visit your alerts from time to time to match these new situations and requirements.

Alerting is an ever evolving process: If the issue your monitoring system reveals is real, you need to handle it. In case it's a false alarm, you need to improve your alerting configuration, or even remove insufficient alerts.

Below you can see a recommended alert refining loop, which helps you to decide how to handle issues and configuration:



Alert handling flow

Alerting Symptoms

The fact that your system stopped doing useful work is a symptom. The most effective alerts are usually based on symptoms. These alerts depend on the externally visible behavior of your services. Learn what matters to your users and use this knowledge to set up your criteria. Don't waste your time on meaningless alerts. With symptom-based alerts, we usually recommend to use a pager system - since these are the issues that can hurt your business the most.

Examples or symptoms:

- **APIs:** Downtime of your system, slow response times for minutes
- **Jobs:** Slow batch time completion time
- **Pipelines:** Big Buffer sizes, old data waiting for too long

JUAN criterion

JUAN is a mosaic word of Judgement, Urgent, Actionable, Necessary.

Don't be surprised if you have never heard about the JUAN criterion system. It's a new concept introduced at Google's latest Cloud Next conference. The important thing here is that the JUAN criterion provides help to validate your alerts, so you can stop wasting your time on meaningless alerts.

The criterion says that every alert should be tested against these points:

Name	Definition	Example for wrong action item
Judgement	Needs human input *(Otherwise why waste the person's time)*	"Always turn this knob."
Urgent	Timely resolution *(Right now or next business day, otherwise why bother person)*	"Can wait for next week."
Actionable	Something can be done *(Doesn't have to be a fix, maybe just a status page update)*	"Went away by itself."
Necessary	The problem is ongoing or imminent *(Is it an important symptom?)*	"Users won't be affected."

Any alert that doesn't pass these tests should be a candidate for removal. The JUAN criterion can help to reduce the alerting noise and make paging more efficient by eliminating unnecessary triggers.

Check out the following good and bad examples for alerts:

Name	Good	Bad
Judgement	SLO alert - something is wrong with a new release	The issue can be fixed with autoscaling
Urgent	SLO impact - you now have security issues	The issue would need immediate action but we postpone it
Actionable	You have to roll back, or add capacity manually	Autoscaling handles the issue, goes aways on it's own
Necessary	Affects users, impacts the business	No impact on users or services

Alerting risks

Setting up alerting introduces several risks: With too high objectives (SLO) you will end up with too many alerts, with too low goals, you might not even notice bad service quality. There's an even worse scenario, when a huge proportion of the alerts are not necessary, which leads people to ignore them altogether (including the important ones).

A relatively low SLO can be acceptable with a rapidly growing user base when the team cannot keep up with the alerts. But it's only acceptable if it's a conscious business decision, and you are confident that your users won't have a bad experience which turns them away from your service. Your alerting has to be controlled and aligned with your business.

Cliff alerts

SLO alerts only fire when a service is disrupted, but cliff alerts can even forecast SLO violations, for example when disk usage is already at 90% & growing. You can set up cliff alerts for many types of metrics. For example, if you know that your machine can only handle a limited number of Docker containers with specific memory settings you want to be alerted when you should add more VM to your cluster to prevent downtimes.

Notification delivery

Without notifications, most of the alerts are useless - since they will never reach you. You should never depend only on one notification method, because a problem can occur with any delivery provider, receiving device or with the person who is on paging.

That's why it's important to:

- Use multiple channels (Phone, Slack, etc.)
- Add policies and escalations with multiple persons
  (Call XY if Z doesn't respond in 2 minutes)
- Use multiple providers, as they can fail, especially during a global attack
  (Dyn DDoS attack in 2016 killed most of the pagers.)
- Don't flood channels with 100's of notifications (Group alerts)

You can use external services for handling escalation policies, multi-channel alerting and on-call rotations, so you don't have to build your own solution.

Think about the person

Paging a human is quite an expensive use of an employee's time. Alerts have human cost as well because alerts often wake up engineers in the middle of night. Staying on-call can lead to the person's burnout or might decrease her/his work quality through focus loss.

This is why you should keep the frequency of alerts low and provide help to the engineering team through clear alerts, better training, and up-to-date documentation.

A fair number of outages from the handling person's point of view is one or two outages per a 12 hours shift.

Far more incidents can lead to burnout. Also, night shifts have detrimental effects on people's health. An exhausted person cannot focus and will start to ignore paging, which leads to inferior service quality.

Google's use case clearly highlights that very frequent alerting can lead to burnout. Their data also shows that 70% of engineers "didn't respond in time" only after one week of over-alerting.

Far fewer incidents can cause a lack of practice, as your engineers will lose the ability to handle incidents effectively. In case you don't have to deal with a lot of incidents, you should limit the number of engineers in the on-call rotation. Handling the same amount of errors with fewer people ensures that engineers don't lose touch with the production systems.

To help the humans when alert fire, be sure that the person handling them:

- **Understands the alert**
  (Create clear alerts, documentations and set up dashboards)
- **Understands the (sub)system**
  (Provide documentation and training)
- **Is provided with mitigations**
  (Set up automation, make it easy to rollback)
- **Is provided with escalations**
  (Don't spend too much time when stuck, rolling back is totally okay)

INCIDENT HANDLING

Efficient incident handling is crucial for preventing and limiting service disruptions. It's a best practice to set up pager for your most critical alerts and put developers or operations people on-call. This way your team can help when your system needs human interaction to recover its functionality. In some cases, you might not be able to help much in a technical way, but still, you can communicate the ongoing incident to your customers and prevent a business disaster.

Being on-call can be tricky, this is why we discussed the human aspect of it in the previous section. Now let's see what's needed to handle alerts efficiently:

First of all, you need multiple people who can receive alerts at the same time, as you cannot expect from anyone to be on-call 24/7. Usually, at least three people are required to be on-call to cover 24 hours with 8 hours shifts, but it highly depends on the team, the incident frequency, and the legal requirements as well. You should also consider other aspects, like multiple time zones. It's a smart idea to minimize the number of night shifts with using the benefits of distributed team members.

To prepare for longer incidents and other negative circumstances, it's a good practice to have a secondary team on standby which you can debrief in case you need to hand over the ongoing incident. To prepare for larger outages and human aspects, I recommend to set up escalation policies and an alerting system in your company. It can be handy when someone is not available at the moment of the incident.

Postmortem

Every incident that had user impact or widely affected your service should end with a postmortem meeting. The meeting's outcome and the event itself should be documented too. During the postmortem meeting, the team should discuss what went well and what didn't.

They should review their processes against their guides and policies, then optimize the processes and the system if it's necessary. Your team should be sure that the same kind of issue doesn't happen again or will be handled more efficiently next time. It's important to keep your postmortems blameless. The goal is to optimize the team's performance and prepare for future situations, not to blame individuals.

Postmortems should discuss the following topics:

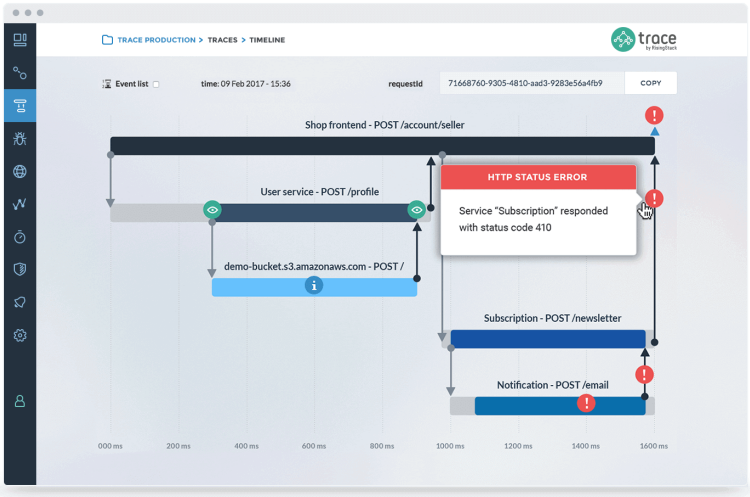
- What happened?
- What went well/poorly/luckily?
- Action items for poor/lucky situations.
- Do we need additional alerts?
- Postmortems that did not trigger a pager are even more valuable, as they likely point to important monitoring gaps.

TROUBLESHOOTING

It's hard to tell what kind of tools are efficient to discover, understand and successfully debug an issue. It always depends on your product and team, but here is a list what you probably want to see in your monitoring toolbox:

- **Dashboard and metrics:**
  For being able to check your system's health quickly.
- **Alerting:** To get notified about ongoing incidents.
- **Uptime probes:** To have black-box monitoring.
- **Distributed tracing:** I
  n case you have a distributed system (like microservices).

Some other tools can also be handy to debug your system: language specific metrics, CPU profiling, requesting memory heap dumps, infrastructure topology view, slow database query analysis - just to mention a few.



Distributed tracing can help to debug distributed transactions in a microservices architecture.

PICK THE RIGHT MONITORING & ALERTING TOOL

Building a proper monitoring solution needs a lot of effort and investment. It also needs business knowledge that may require additional hiring in your team. Instrumentations can be very specific and usually need more development time. Also, a bad instrumentation can have a massive overhead.

When you pick your monitoring tool, I recommend to ask these questions to yourself:

- **Do you have the know-how?**
  If not, I recommend something simple and specific to your challenges.
- **Do you have time to develop or configure your own solution?**
  If not, I recommend a SaaS monitoring tool. With a SaaS solution, you don't have to spend time to set up, maintain and debug your monitoring tool.
- **Do you have a strict data compliance policy?**
  If yes, I recommend an on-premises solution.
- **Do you want to do instrumentations on your own?**
  If you don't, I recommend using out of the box instrumentations. A bad instrumentation can be a source of bugs or have a negative effect on your application's performance.
- **Do you have a microservice architecture or do plan to build one?**
  If yes, I recommend a specialized tool with specific tools like distributed tracing.
- **Do you have a homogeneous or heterogeneous infrastructure?**
  General tools usually give less value for specific languages and technologies.
- **What's your budget for monitoring?**
  It's a hard question, I know. Sometimes an open-source solution is cheaper, but sometimes it costs more if you count the setup and learning costs or support options. You can also find SaaS monitoring tools with very different pricing models: one can be reasonable for your architecture while the other one can cost more.

CHAPTER III. :
TECHNIQUES TO BUILD RELIABLE SYSTEMS

There are many architectural and infrastructure techniques that you can apply to your services to build a reliable product.

In this chapter, we will check out some of the most powerful ones. If you are not familiar with these patterns, it doesn't necessarily mean that you do something wrong. Remember, building a reliable system always comes with an extra cost.

Change management

Google's site reliability team has found that roughly 70% of the outages are caused by changes in a live system. When you change something in your service - like deploying a new version of your code or changing some configuration - there is always a chance for failure or the introduction of a new bug to your application. To minimize these failures and limit their negative effect, you can implement change management strategies and automatic rollouts.

For example, when you deploy new code, or you change some configuration you should apply these changes to a subset of your instances gradually, monitor them and even automatically revert the deployment if you see that it has a negative effect on your key metrics.

Another solution could be that you run two production environment, you always deploy to only one of them, and you only point your load balancer to the new one after you verified that the new version works as it is expected. This is called blue-green, or red-black deployment.

Reverting code is not a bad thing. You shouldn't leave broken code in production and then think about what went wrong. Always revert your changes when it's necessary, sooner is better.

Multiple regions and clouds

Most cloud providers offer highly resilient systems, some of them even have products with 99.999% availability. For most of the services, it may be enough, but like every system, their solution can also fail. To protect yourself from provider issues, you can distribute your applications between multiple availability-zones, regions or even between multiple cloud providers.

In most of the cases, two regions are more than enough to achieve very high availability. At a great cloud provider, availability zones are close to each other but remain physically separated. Most of cloud providers also have restrictions on making changes at the same time in more than one zone.

When we are talking about multi-cloud and multi-region solutions, it's important to mention that duplicating all of the resources can be very expensive for most of the companies. A more efficient solution is to distribute the load between these locations and only scale up some of them when one has an outage. Multiple clouds also come with the cost of preparing your monitoring, automation systems, and developer team to deal with different locations.

Auto Scaling

Temporary traffic peaks are one of the most common causes of problems after change management and provider issues. When your services receive more traffic than what they can handle in a fast a stable way, they usually slow down or even fail immediately. In the further sections of this chapter, we will check out how you can protect yourself from these situations.

In case you need any help with Node.js, feel free to ping me at

peter@risingstack.com

In some cases, your infrastructure needs to scale up to handle increased traffic. In this situation, auto scaling can help a lot. Auto scaling can add new machines to your cluster automatically and therefore prevent a disaster. However, starting a new machine can take longer time, but after it's running, it can handle the increased load, and your system can recover. If your peak times come at the same period every day or on the same day every month, you can build an advanced auto-scaling solution which can forecast peak times and spin up new machines even before your services start struggling. This kind of auto scaling is called predictive auto scaling.

You can also use different metrics as a base of your auto-scaling algorithm. One of the most common technique is to watch CPU utilization and start new machines when the CPU usage is high for a certain period. You can also add more machines to your cluster when you see that your current resources are full and you cannot start more containers in the current setup.

As most of the cloud providers provide built-in auto-scaling solutions, so you don't have to build your own. Also, keep in mind that these solutions are usually limited and can only scale based on CPU usage. They usually lack predictive auto-scaling as well.

You should be aware of auto scaling events that come from releasing a new version of your app. It can easily happen that the new version is less performant and uses more CPU. In this case, you probably want to optimize or revert the new version, especially if it costs a lot more. Automatic scaling is a powerful tool but not a silver bullet. You should re-visit your settings on a regular basis and keep your eyes on them.

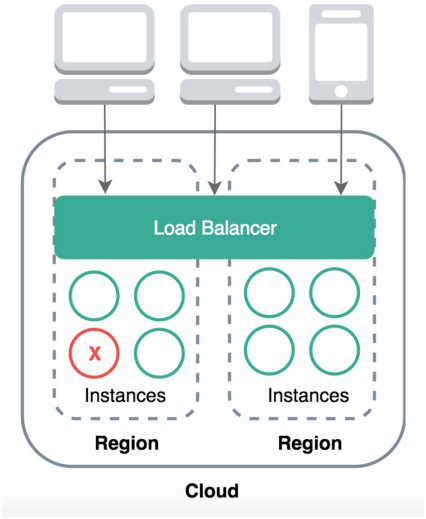
Health-check and load balancing

Instances continuously start, restart and stop because of failures, deployments or autoscaling. It makes them temporarily or permanently unavailable. In most of the cases, it's acceptable if you let your load balancer to find and skip unhealthy instances from the routing as they cannot serve your customers or sub-systems need.

It leads to the question: How can you determine from a single instance if it's healthy and ready to accept traffic? In clustered environments, instances have a unique endpoint - like an IP address - what you can use to make TCP, UDP or HTTP probes against specific ports or routes to continuously check your instance's health.

For example, you can set up a special GET /healthz endpoint that returns a 200 status code in any cases when it's ready to accept traffic. The logic behind this /healthz endpoint strongly depends on your application. A best practice is to have something that is close to your business logic but cheap, as you will call it frequently.

Modern load balancers can periodically collect health status from their balanced instances and send traffic only to healthy ones. The latest generation of load balancers is also capable of collecting statistics about response times and error rates and allocate the traffic based on these parameters as well. This way, load balancers can prefer closer regions with lower latency and more stable instances to optimize the customer's satisfaction.



A load balancer can skip unhealthy instances from routing

Self-healing

Self-healing is when an application can do the necessary steps and changes to recover from a broken state. In most of the cases, it is implemented by an external system that watches instance's health and restarts them when they are in a broken state for a longer period. However, self-healing can be very useful in most of the cases, in certain situations it can cause trouble by continuously restarting the application. This might happen when your application cannot give positive health status because it's overloaded or it's database connection timeouts.

Implementing an advanced self-healing solution which is prepared for a delicate situation - like a lost database connection - can be tricky. In this case, you need to add extra logic to your application to handle edge cases and let the external system know that the instance is not needed to restart immediately.

Failover caching

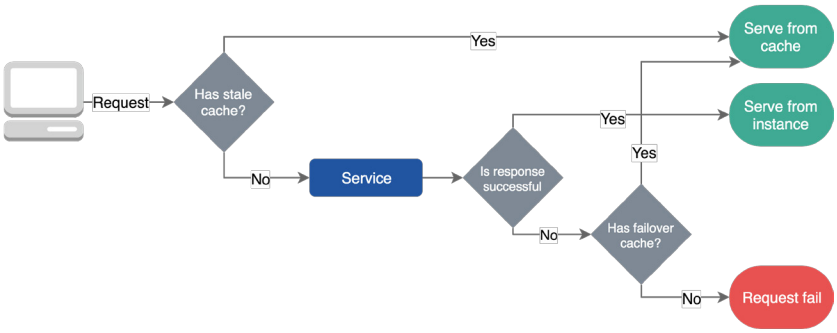
Most of the services have some kind of caching enabled to speed up their response time and decrease the actual traffic on expensive resources. But only a few of them has failover cache that only kicks in when instances struggle to serve traffic.

Failover caches usually use two different expiration dates; a shorter that tells how long you can use the cache in a normal situation, and a longer one

that says how long can you use the cached data during failure. It's important to mention that you can only use failover caching when it serves better the outdated data than nothing.

To set cache and failover cache, you can use standard response headers in HTTP. For example, with the "max-age" header you can specify the maximum amount of time a resource will be considered fresh. With the "stale-if-error" header, you can determine how long should the resource be served from a cache in the case of failure.

Modern CDNs and load balancers provide various caching and failover behaviors, but you can also create a shared library for your company that contains standard reliability solutions.



Cache with failover logic

Client retry

Numerous components of a service can generate errors anywhere in the life of a given request. To deal with these error responses in a networked environment, a best practice is to implement retries in the client application. This solution increases the reliability of the application and reduces operational costs for the developer.

You should be careful with adding retry logic to your applications and clients, as a larger amount of retries can make things even worse or even prevent the application from recovering. In distributed system, a microservice retry can trigger multiple other requests or retries and start a cascading effect.

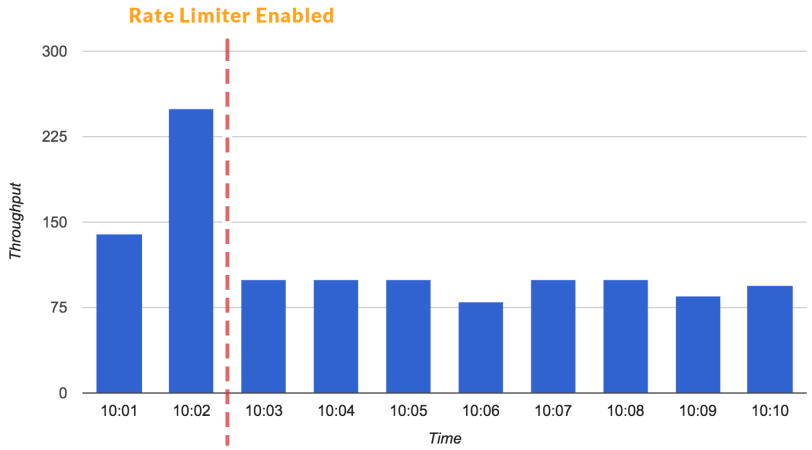
To minimize the impact of retries, you should limit the number of retries and use exponential backoff or similar algorithm to continually increase the delay between retries, until you reach the maximum limit.

You can implement retries for various network-based solutions from HTTP to messaging queues. Most of the messaging brokers already provide some retry logic. You can also build your retry logic in a company-wide shared library.

Rate limiters and load shedders

Rate limiting is the technique of defining how many requests can be received or processed by a particular customer or application during a time frame.

With rate limiting, for example, you can filter out clients who are responsible for traffic peaks, or you can ensure that your application doesn't overload until autoscaling doesn't come to rescue. You can also hold back lower-priority traffic to give enough resources to critical transactions.



Rate limiter can hold back peaking traffic

Concurrent request limiters can be useful when you have expensive endpoints that shouldn't be called more than a specified times, while you still want to serve any other traffic.

A fleet usage load shedder can ensure that there are always enough resources available to serve critical transactions. It keeps some resources for high priority requests and doesn't allow for low priority transactions to use all of them. A load shedder makes its decisions based on the whole state of the system, rather than based on a single user's request bucket size. Load shedders help your system to recover, since they keep the core functionalities working while you have an ongoing incident.

To read more about rate limiter and load shredders, I recommend to check out Stripe's article.

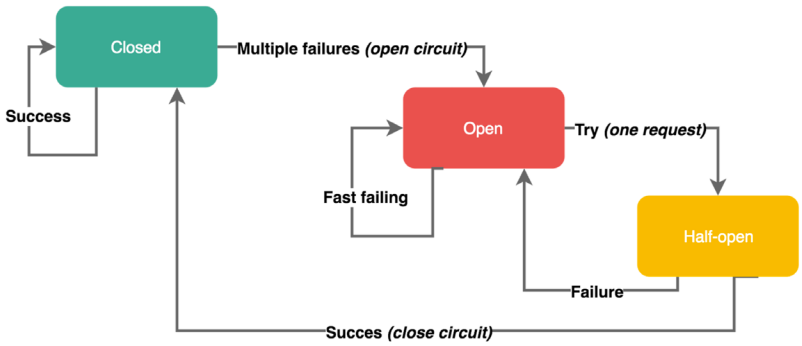
Circuit breakers

Circuit breakers are named after the real world electronic component because their behavior is identical. You can protect resources and help them to recover with circuit breakers. They can be very useful in distributed system, where a repetitive failure can lead to cascading effect and bring the whole system down.

A circuit breaker opens, when a particular type of errors occurs multiple times in a short period. An open circuit breaker prevents further requests to be made, like the real one prevents electrons from flowing. Circuit breakers usually close after a certain amount of time, giving enough space for underlying services to recover.

Keep in mind that not all errors should trigger a circuit breaker. For example, you probably want to skip client side issues like requests with 4xx response code, but include 5xx server-side failures.

Some circuit breakers can have a half-open state as well. In this state, the service sends the first request to check system availability, while letting the other requests to fail. If this first request succeeds, it restores the circuit breaker to a closed state and lets the traffic flow. Otherwise, it keeps it open.



Circuit Breaker State Diagram

You can read more about circuit breakers from Ebay.

Load testing

You can discover bottlenecks in your application or infrastructure by load testing or stress testing services. Testing your system under heavy load allows you to gain knowledge about its behavior under different circumstances and prepare for various situations in a controlled environment. However, in most of the situations, you should never run a load test against your production environment.

For setting up an efficient load testing scenario, you should decide what will be tested, and you should generate fake traffic that is as close to the real one as possible. For example, making 4xx requests probably won't test your real business logic with database calls.

You should implement load tests to extract various metrics that can help to judge and understand your system's behavior (like your API's response time, distribution and error rates). To measure these outputs, you can use your monitoring system.

During load testing, you can find your system's baseline - the maximum load that your system can handle without starting to collapse.

Testing for failure

You should continually test your system against common issues to make sure that your services can survive various failures.You should test for failures frequently to keep your team prepared for incidents.

For testing, you can use an external service that identifies groups of instances and randomly terminates one of the instances in this group. With this, you can prepare for a single instance failure, but you can even shut down entire regions to simulate a cloud provider outage.

One of the most popular testing solutions is the ChaosMonkey method by Netflix.

NODE.JS MONITORING WITH PROMETHEUS

Prometheus is an open-source solution for Node.js monitoring and alerting. It provides powerful data compressions and fast data querying for time series data.

The core concept of Prometheus is that it stores all data in a time series format. Time series is a stream of immutable timestamped values that belong to the same metric and the same labels. The labels cause the metrics to be multi-dimensional.

Funfact: Prometheus was initially built at SoundCloud, in 2016 it joined the Cloud Native Computing Foundation as the second hosted project after Kubernetes.

You can read more about how Prometheus optimizes its storage engine in the Writing a Time Series Database from Scratch article.

DATA COLLECTION AND METRICS TYPES

Prometheus uses the HTTP pull model, which means that every application needs to expose a GET /metrics endpoint that can be periodically fetched by the Prometheus instance.

Prometheus has four metrics types:

- **Counter:**
  cumulative metric that represents a single numerical value that only ever goes up
- **Gauge:**
  represents a single numerical value that can arbitrarily go up and down
- **Histogram:**
  samples observations and counts them in configurable buckets
- **Summary:**
  similar to a histogram, samples observations, it calculates configurable quantiles over a sliding time window

In the following snippet, you can see an example response for the /metrics endpoint. It contains both the counter (nodejs_heap_space_size_total_bytes) and histogram (http_request_duration_ms_bucket) types of metrics:

```
# HELP nodejs_heap_space_size_total_bytes Process heap space
size total from node.js in bytes.
# TYPE nodejs_heap_space_size_total_bytes gauge
nodejs_heap_space_size_total_bytes{space="new"} 1048576
1497945862862
nodejs_heap_space_size_total_bytes{space="old"} 9818112
1497945862862
nodejs_heap_space_size_total_bytes{space="code"} 3784704
1497945862862
nodejs_heap_space_size_total_bytes{space="map"} 1069056
1497945862862
nodejs_heap_space_size_total_bytes{space="large_object"} 0
1497945862862

# HELP http_request_duration_ms Duration of HTTP requests in
ms
# TYPE http_request_duration_ms histogram
http_request_duration_ms_bucket{le="10",code="200",route="/",m
ethod="GET"} 58
http_request_duration_ms_bucket{le="100",code="200",route="/",
method="GET"} 1476
http_request_duration_ms_bucket{le="250",code="200",route="/",
method="GET"} 3001
http_request_duration_ms_bucket{le="500",code="200",route="/",
method="GET"} 3001
http_request_duration_ms_bucket{le="+Inf",code="200",route="/"
,method="GET"} 3001
```

Prometheus offers an alternative, called the Pushgateway to monitor components that cannot be scrapped because they live behind a firewall or are short-lived jobs.

Before a job gets terminated, it can push metrics to this gateway, and Prometheus can scrape the metrics from this gateway later on. To set up Prometheus to periodically collect metrics from your application check out the following example configuration.

MONITORING A NODE.JS APPLICATION

When we want to monitor our Node.js application with Prometheus, we need to solve the following challenges:

- **Instrumentation:**
  Safely instrumenting our code with minimal performance overhead
- **Metrics exposition:**
  Exposing our metrics for Prometheus with an HTTP endpoint
- **Hosting Prometheus:**
  Having a well configured Prometheus running
- **Extracting value:**
  Writing queries that are statistically correct
- **Visualizing:**
  Building dashboards and visualizing our queries
- **Alerting:** Setting up efficient alerts
- **Paging:**
  Get notified about alerts with applying escalation policies for paging

In case you need any help with Node.js, feel free to ping me at

peter@risingstack.com

Node.js Metrics Exporter

To collect metrics from our Node.js application and expose it to Prometheus we can use the prom-client npm library.

In the following example, we create a histogram type of metrics to collect our APIs' response time per routes. Take a look at the pre-defined bucket sizes and our route label:

```
// Init
const Prometheus = require('prom-client')
const httpRequestDurationMicroseconds = new
Prometheus.Histogram({
  name: 'http_request_duration_ms',
  help: 'Duration of HTTP requests in ms',
  labelNames: ['route'],
  // buckets for response time from 0.1ms to 500ms
  buckets: [0.10, 5, 15, 50, 100, 200, 300, 400, 500]
})
```

We need to collect the response time after each request and report it with the route label.

```
// After each response
httpRequestDurationMicroseconds
  .labels(req.route.path)
  .observe(responseTimeInMs)
```

We can register a route a GET /metrics endpoint to expose our metrics in the right format for Prometheus .

```
// Metrics endpoint
app.get('/metrics', (req, res) => {
  res.set('Content-Type', Prometheus.register.contentType)
  res.end(Prometheus.register.metrics())
})
```

QUERIES

After we collected our metrics, we want to extract some value from them to visualize. Prometheus provides a functional expression language that lets the user select and aggregate time series data in real time.

The Prometheus dashboard has a built-in query and visualization tool:

Prometheus dashboard

Let's see some example queries for response time and memory usage.

Query: 95th Response Time

We can determinate the 95th percentile of our response time from our histogram metrics. With the 95th percentile response time, we can filter out peaks, and it usually gives a better understanding of the average user experience.

```
histogram_quantile(0.95,
sum(rate(http_request_duration_ms_bucket[1m])) by (le,
service, route, method))
```

Query: Average Response Time

As histogram type in Prometheus also collects the count and sum values for the observed metrics, we can divide them to get the average response time for our application.

```
avg(rate(http_request_duration_ms_sum[1m]) /
rate(http_request_duration_ms_count[1m])) by (service, route,
method, code)
```

For more advanced queries like Error rate and Apdex score check out our Prometheus with Node.js example repository.

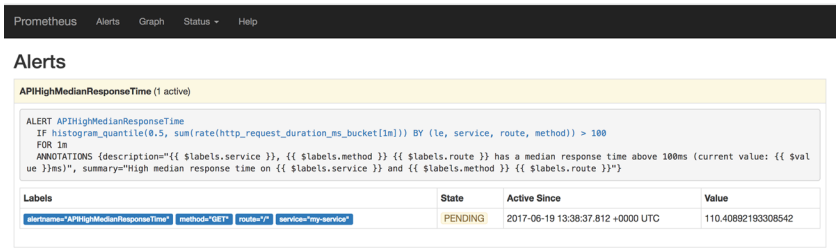
ALERTING

Prometheus comes with a built-in alerting feature where you can use your queries to define your expectations, however, Prometheus alerting doesn't come with a notification system. To set up one, you need to use the Alert manager or an other external process.

Let's see an example of how you can set up an alert for your applications' median response time. In this case, we want to fire an alert when the median response time goes above 100ms.

```
# APIHighMedianResponseTime
ALERT APIHighMedianResponseTime
  IF histogram_quantile(0.5,
sum(rate(http_request_duration_ms_bucket[1m])) by (le,
service, route, method)) > 100
  FOR 60s
  ANNOTATIONS {
    summary = "High median response time on {{ $labels.service
}} and {{ $labels.method }} {{ $labels.route }}",
    description = "{{ $labels.service }}, {{ $labels.method }}
{{ $labels.route }} has a median response time above 100ms
(current value: {{ $value }}ms)",
  }
```

This is how the Prometheus active alert looks like in pending state:



KUBERNETES INTEGRATION

Prometheus offers a built-in Kubernetes integration. It's capable of discovering Kubernetes resources like Nodes, Services, and Pods while scraping metrics from them.

It's an extremely powerful feature in a containerized system, where instances born and die all the time. With a use-case like this, HTTP endpoint based scraping would be hard to achieve through manual configuration.

You can also provision Prometheus easily with Kubernetes and Helm. It only needs a couple of steps. First of all, we need a running Kubernetes cluster!

As Azure Container Service provides a hosted Kubernetes, I can provision one quickly:

```
# Provision a new Kubernetes cluster
az acs create -n myClusterName -d myDNSPrefix -g
myResourceGroup --generate-ssh-keys --orchestrator-type
kubernetes

# Configure kubectl with the new cluster
az acs kubernetes get-credentials --resource-
group=myResourceGroup --name=myClusterName
```

After a couple of minutes when our Kubernetes cluster is ready, we can initialize Helm and install Prometheus:

```
helm init
helm install stable/prometheus
```

For more information on provisioning Prometheus with Kubernetes check out the Prometheus Helm chart.

GRAFANA

As you can see, the built-in visualization method of Prometheus is great to inspect our queries output, but it's not configurable enough to use it for dashboards. As Prometheus has an API to run queries and get data, you can use many external solutions to build dashboards. One of my favorite is Grafana.

Grafana is an open-source, pluggable visualization platform. It can process metrics from many types of systems, and it has built-in Prometheus data source support. In Grafana, you can import an existing dashboard or build you own.

CONCLUSION

Prometheus is a powerful open-source tool to monitor your application, but as you can see, it doesn't work out of the box.

With Prometheus, you need expertise to instrument your application, observe your data, then query and visualize your metrics.

You can find our example repository below, which can help you with more in-depth advice in case you'll choose this way of monitoring your Node.js application.

Example repository: RisingStack/example-prometheus-nodejs

RECOMMENDED TOOLS TO BUILD A RELIABLE NODE.JS APPLICATION

We at RisingStack - based on our consulting, development & monitoring experience - usually recommend the following libraries and solutions to build reliable applications with Node.js:

- Out of the box SaaS and on-premises Node.js monitoring and debugging: https://trace.risingstack.com
- Open-source self hosted monitoring and alerting: https://prometheus.io/
- Hosted Kubernetes: https://azure.microsoft.com/en-us/services/container-service
- Production-grade container orchestration with self-healing and auto scaling: https://kubernetes.io
- Redis based rate-limiter: https://github.com/tj/node-ratelimiter
- Rolling rate-limiter: https://github.com/RisingStack/rate-limiter
- Circuit breaker: https://github.com/krakenjs/levee
- Open-source failover cache with multi level storage: https://github.com/RisingStack/cache
- Retry with exponential backoff logic: https://github.com/MathieuTurcotte/node-backoff
- Proactively protect your Node.js web services: https://github.com/RisingStack/protect

OUTRO

Implementing and running a reliable service is not easy. It takes a lot of effort from your side and also costs money.

Reliability has many levels and aspects, so it is important to find the best solution for your team. You should make reliability a factor in your business decisions process and allocate enough budget and time for it.

You should also pick the best tools for the job and choose the right monitoring and alerting solution that fits your application, technology, budget, and team.

SOURCES

- SLOs, SLIs, SLAs, oh my
- Alerting best practices
- Designing reliable systems with cloud infrastructure
- Site Reliability Engineering
- Scaling your API with rate limiters
- CircuitBreaker
- Application Resiliency Using Netflix Hystrix