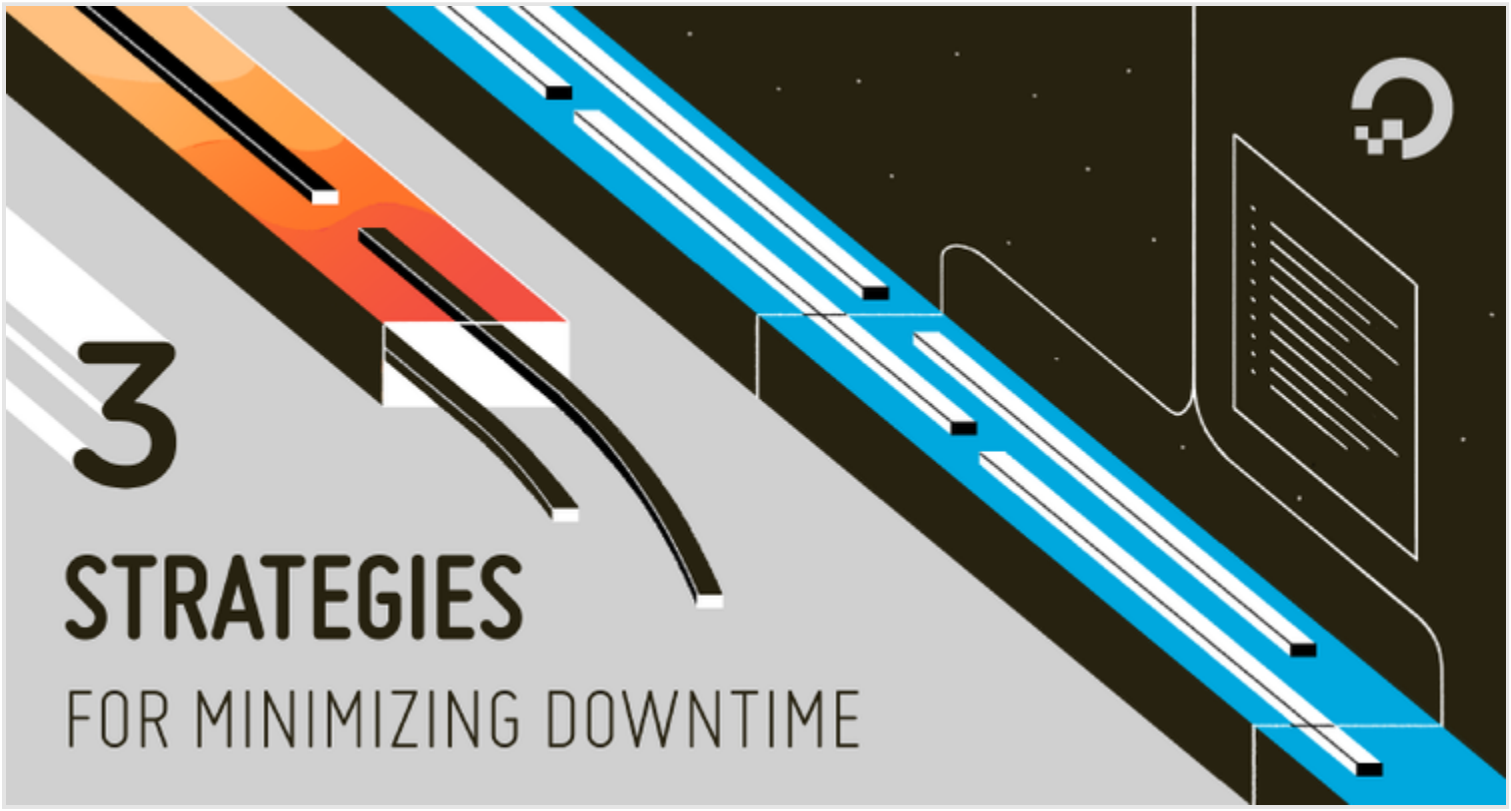

3 Strategies for Minimizing Downtime

Posted September 26, 2017 21.6k HIGH AVAILABILITY CI/CD LOAD BALANCING MONITORING CONCEPTUAL

By: Brian Boucheron

Introduction

As businesses and other organizations increasingly depend on internet-based services, developers and sysadmins are focusing their attention on creating reliable infrastructure that minimizes costly downtime.

A website, API, or other service being unavailable can have significant monetary costs resulting from lost sales. Additionally, downtime can lead to:

- **Unhappy Customers and Users:** Users expect stable services. Interruptions can lead to increased support requests and a general loss of confidence in your brand.
- **Lost Productivity:** If your employees depend on a service to do their jobs, downtime means lost productivity for your organization. Also, if your employees are spending their time rebooting servers and fighting downtime, they're not developing new features and products.

- **Unhappy Employees:** Frequent downtime alerts can lead to alert fatigue, and constantly scrambling to solve problems can take a toll on your team and their morale.

The modern field that has coalesced to address these issues is called *Site Reliability Engineering* or SRE. SRE was created at Google starting in 2003, and the strategies they developed were gathered into a book titled *Site Reliability Engineering*. Reading up on this field is a good way to explore techniques and best practices for minimizing downtime.

In this article, we will discuss three areas where improvements could lead to less downtime for your organization. These areas are **monitoring and alerting**, **software deployments**, and **high availability**.

This is not an exhaustive list of strategies, but is intended to point you toward some common solutions you should consider when improving the production readiness of your services.

1. Monitoring and Alerting

Properly monitoring your infrastructure will enable you to proactively watch for impending issues, alert your team to problems, and more easily investigate the causes of previous downtimes. Monitoring involves aggregating and recording statistics such as system resource utilization and application performance metrics. Alerting builds upon this metric collection by constantly evaluating alert rules against current metrics to determine when action needs to be taken.

Monitoring is often implemented with a client on each host that gathers metrics and reports back to a central server. The metrics are then stored in a time series database (a database that specializes in storing and searching timestamped numerical data) and made available for graphing, searching, and alerting. Some examples of monitoring software are:

- **Prometheus** can pull data from a wide variety of official and community-supported clients (called *exporters*). It is highly-scalable, has built-in alerting, and has client libraries available for most programming languages.
- **Graphite** provides a now-ubiquitous API that is supported by dozens of programming languages and applications. Metrics are pushed from nodes to the central Graphite installation, where they are stored and graphed.

It's also common to push log files to a central service and parse them for relevant metrics such as exceptions and error rates. The Elastic stack (Elasticsearch, Logstash, Kibana) and Graylog are examples of software stacks that can facilitate log file parsing and analysis.

Which Metrics to Monitor

What are the most useful metrics to collect when attempting to increase reliability and reduce downtime?

Most of the monitoring clients and exporters will have a set of default metrics that are a good starting point, but your application or service will have unique needs that you'll want to consider when deciding which measurements

to export. Thankfully, the SRE literature has some general guidelines for what are the most useful metrics to monitor. These metrics are grouped into the **Four Golden Signals**, summarized here from the SRE book:

- **Latency:** how long it takes to respond to a request. For example, a server's response time for an HTTP request.
- **Traffic:** how much demand your system is experiencing. This could be request rate for a web server, network I/O, logins per second, or transactions per second for a database.
- **Errors:** the failure rate of transactions or requests. Bear in mind that not every error is as clear-cut as an HTTP 500 error. For instance, it may be your policy that clients should receive responses in 500ms or less. Any responses with higher latency should show up as an error in this case.
- **Saturation:** how "full" a service is. This could be measuring available space on a hard drive, network throughput, or even how much CPU is available on a CPU-bound service.

Visualizing Metrics

Once you have your monitoring system set up, you'll want to explore the data you're collecting. Grafana is a software package that provides very flexible exploration of metrics through graphs and dashboards. Visualizations can be aggregated from multiple backend data sources, including Graphite, Prometheus, and Elasticsearch.

Setting Up Alerts

In addition to visualizing your metrics, you'll also need to set up some way to automatically alert your team when problems arise. Grafana has alerting capabilities, as does Prometheus through its Alertmanager component. Other software packages that allow you to define alert rules for metrics include Nagios and Sensu.

How you structure your alerting system will depend greatly on your organization. If you have multiple teams, alerts may be routed directly to the team responsible for the misbehaving service. Or you may have a scheduled on-call rotation that handles all alerts for a specific time period. Alerts may be sent via email, push notifications, or a third party paging service.

The main thing to remember is that the goal should be to alert as little as possible, and only for events where your team can take immediate action to solve the problem. Too many alerts, or alerts for situations that aren't actually fixable can lead to *alert fatigue*. This fatigue can result in unhappy employees who can miss or ignore alerts that *are* critical and actionable.

The SRE book summarizes some principles to keep in mind when setting up alerts:

- Every time the pager goes off, I should be able to react with a sense of urgency. I can only react with a sense of urgency a few times a day before I become fatigued.
- Every page should be actionable.

- Every page response should require intelligence. If a page merely merits a robotic response, it shouldn't be a page.
- Pages should be about a novel problem or an event that hasn't been seen before.

If your alert doesn't meet these criteria, it may be best to send it to a lower priority ticketing system or log file, or remove the alert entirely.

For more information on setting up some open-source monitoring and alerting systems, please refer to the related tutorials below.

Related Tutorials:

- How To Use Prometheus to Monitor Your Ubuntu 14.04 Server
- How To Install and Use Graphite on an Ubuntu 14.04 Server
- An Introduction to Tracking Statistics with Graphite, StatsD, and CollectD
- How to Install and Configure Zabbix to Securely Monitor Remote Servers on Ubuntu 16.04

2. Improving Software Deployments

Another area that can have an impact on downtime is your software deployment strategy. If your deploy process requires multiple manual steps to complete, or is otherwise overcomplicated, your production environment is likely to lag behind your development environment significantly. This can lead to more risky software releases, as each deploy becomes a much larger set of changes with more potential complications. This in turn leads to bugs, slows down development velocity, and could result in unintentionally deploying degraded or unavailable services.

It will take some up-front time and planning to smooth out your deployments, but in the end, coming up with a strategy that allows you to automate the workflow of code integration, testing, and deployment will lead to a production environment that more closely matches development — with more frequent deploys and fewer complications between releases.

CI/CD Best Practices

With the diversity of container technologies, configuration management software, programming languages, and operating systems available today, the specific details on how to automate your deployments is beyond the scope of any one article. In general, a good place to start is making sure you're following best practices regarding continuous integration (CI) delivery (CD) and testing of your software. Some of these best practices include:

- **Maintain a Single Repository:** everybody should be regularly integrating code into the same repository, which should contain everything needed (including tests and config files) to build the project on a relatively bare machine or in a CI environment. This ensures that everybody is working on a similar code base, integrations are relatively painless, and everybody can test and build their changes easily.

- **Automate Testing and Building:** your CI software or service should be able to automatically run tests and build your project.
- **Automate Deployment:** your CI software should be able to deploy and test a build in a test environment that closely simulates the final production environment.

These practices describe a continuous integration and delivery environment, but they don't get us all the way to a production deploy. It's possible that — given enough confidence in your automated testing — you could be comfortable automating the deploy to production. Or you may choose to make this a manual task (or rather, an automated task that requires human judgement to initiate).

Either way, you'll need a way to push the new version into production without causing downtime for your users. Frequent deploys would be a major inconvenience if they involved service disruptions while infrastructure is being updated.

Blue-Green Deployments

One specific way to implement a safe final push to production — with seamless cutover between versions — is called a **blue-green deployment**.

A blue-green deployment involves setting up two identical production environments behind a mechanism that can easily switch traffic between the two. This mechanism could be a floating IP address that can be quickly switched between a blue or green server, or a load balancer for switching between whole clusters of multiple servers.

At any point in time only one of the environments — let's say blue in this case — is live and receiving production traffic. The process to release a new version is:

- Push the build to the inactive environment (green, for this example).
- Test the build in this production environment.
- If all tests are passed, cut traffic over to the green environment by updating the load balancer config or by assigning the floating IP to the green server.

This technique has the added benefit of providing a simple mechanism for rolling back to the previous version should anything go awry. Keep the previous deploy up and on standby until you're comfortable with the new release. If a rollback is necessary, update the load balancer or floating IP again to revert back to the prior state.

Again, there are many ways to achieve the goal of automating your deploy process in a safe and fault-tolerant way. We've only highlighted one possibility here. The related tutorials below have more information on open source CI/CD software and blue-green deployments.

Related Tutorials:

- How To Use Blue-Green Deployments to Release Software Safely
- An Introduction to Continuous Integration, Delivery, and Deployment

- CI/CD Tools Comparison: Jenkins, GitLab CI, Buildbot, Drone, and Concourse

3. Implementing High Availability

One final strategy for minimizing downtime is to apply the concept of *high availability* to your infrastructure. The term *high availability* encapsulates some principles of designing redundant and resilient systems. These systems typically must:

- **Eliminate Single Points of Failure:** this usually means multiple redundant servers, or redundant containerized services. Also under consideration is the possibility of spreading infrastructure between multiple datacenters in multiple regions. How deep you go in eliminating these bottlenecks will vary depending on your tolerance to cost and complexity, and the needs of your organization and users. Not all services will need redundant load balancers and automatic failover between regions, for instance.
- **Seamlessly Redirect Traffic:** to minimize downtime, cutting traffic between servers must be quick, with minimal service interruption.
- **Detect the Health of the Redundant Systems:** to inform the decision of when to reroute traffic, the system must be able to determine when a service is failing.

Upgrading Web Servers with a Load Balancer

One example of upgrading to more highly available infrastructure would be moving from a single web server to multiple web servers and a load balancer. The load balancer will perform regular health checks on the web servers, and will route traffic away from those that are failing. This infrastructure can also enable more seamless code deployments, as discussed in the previous section.

Increasing Database Resilience

Another example of adding resilience and redundancy is database replication. The MySQL database server, for example, has many different replication configurations. *Group replication* is interesting in that it enables both **read** and **write** operations on a redundant cluster of servers. Instead of having all your data on a single server, vulnerable to downtime, you can replicate between multiple servers. MySQL will automatically detect failing servers and route around the issue.

> Newer databases, such as CockroachDB are being built from the ground up with these redundant replication features in mind, enabling highly available database access across multiple machines in multiple datacenters, out of the box.

Using Floating IPs and a Hot Spare Server

One last example of highly available architecture is to use floating IPs to fail traffic over to a *hot spare* server. Many cloud providers have special IP addresses that can be quickly reassigned to different servers or load balancers using an API. A hot spare is a redundant server that's always ready to go, but receives no traffic until the primary server fails a health check. At that point the floating IP is reassigned to the hot spare

and the traffic follows along. To implement these health checks and the floating IP API calls, you'll need to install and configure some additional software, such as keepalived or heartbeat.

The related tutorials below highlight more software and tools that can be used to increase the resilience of your infrastructure.

Related Tutorials:

- What is High Availability?
- 5 DigitalOcean Load Balancer Use Cases
- How To Use Floating IPs on DigitalOcean
- An Introduction to HAProxy and Load Balancing Concepts
- How To Configure MySQL Group Replication on Ubuntu 16.04
- How To Deploy CockroachDB on a Three-Node Cluster on Ubuntu 16.04

Conclusion

In this article we reviewed three areas where improvements to processes or infrastructure could lead to less downtime, more sales, and happier users. Monitoring metrics, improving deployments, and increasing infrastructure availability are good places to start investigating the changes you can implement to make your app or service more production-ready and resilient.

There is, of course, more to reliability beyond just these three points. For more information, you can dive deeper into the suggested tutorials at the end of each section, and check out the *Site Reliability Engineering* book, which is available for free online.

By: Brian Boucheron

Related Tutorials

How To Use Traefik as a Reverse Proxy for Docker Containers on Debian 9

How To Install Elasticsearch, Logstash, and Kibana (Elastic Stack) on CentOS 7

How To Set Up Buildbot on FreeBSD

How To Install Elasticsearch, Logstash, and Kibana (Elastic Stack) on Ubuntu 18.04

How To Use Traefik as a Reverse Proxy for Docker Containers on Ubuntu 18.04

0 Comments

Leave a comment...

Log In to Comment