elastisys

2022-09-14

Free Guide

How to Operate a Secure Kubernetes Platform

Table of Contents

Introduction

This guide discusses the four most important aspects when operating a secure Kubernetes platform. Having been with Kubernetes since 2015, we identified these aspects to still challenge platform teams, even as technology matures.

The guide lists the aspects in increasing order of importance. For better understanding, we will illustrate these aspects through the lens of Elastisys Compliant Kubernetes, our Certified Kubernetes, open-source, security-hardened Kubernetes platform. While a certain “survival bias” cannot be avoided, we believe that these aspects are relevant for anyone embarking on the Kubernetes journey.

Although this guide is by no means exhaustive, we believe it sets a solid foundation for organizations embarking on their Kubernetes journey.

1. Setting the Scope

Simply put, the scope is a mental fence: Everything inside that fence is the Kubernetes platform, and is the responsibility of the platform team. Everything outside that fence is “not the Kubernetes platform”.

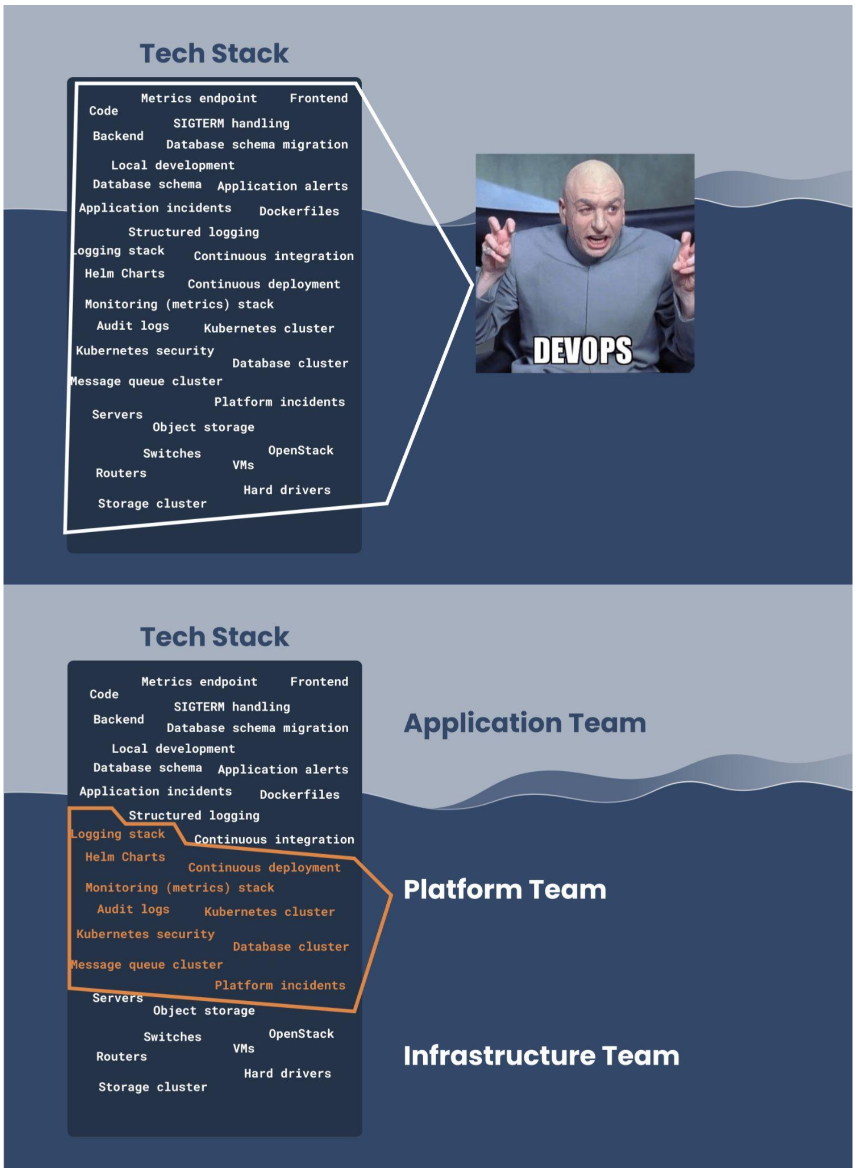
Why is Scope Important?

- **It avoids blame games.** How often do you hear the frontend folks saying “it’s a backend issue” only to hear the backend folks reply “it’s a frontend issue”? Setting a clear scope for the Kubernetes Platform not only avoids blame games. It holds the platform team accountable for the platform, while allowing them to escalate issues outside their accountability to adjacent teams, for example, the application team, infrastructure team or networking team.

- **It empowers the platform team.** Knowing exactly what you “own” also gives you tremendous freedom to make the right decisions within your area of responsibility. For example, instead of holding an “all-hands” meeting to decide if running containers as root is allowed, the platform team can assess that the risk is too high and that application containers must run as non-root.

- **It makes your auditor happy.** If you aim for an SOC-2 or ISO 27001 certification, you will need to talk to an auditor. Although the auditors (generally) can't tell how to properly configure Kubernetes RBACs, they will quickly identify if you rely on too few people to know too many things. That's a business risk.

- **It assigns the right skills to the right task.** While we all wish we could hire mythical DevOps creatures that can do everything from design, to frontend development, to backend development, to database administration, and Kubernetes administration, that is simply not realistic. Remember the platform team not only needs to know Kubernetes. They need to have "muscle memory" and be able to solve any stability- or security-related issues even at 2 AM – both tired and stressed.



A Real-Life Scope Example

So what is the scope of the Kubernetes platform? Let us inspire you with a known-to-work scope that has been demonstrated to work.

In Elastisys Compliant Kubernetes, we set the scope of the platform as follows:

- the Kubernetes cluster itself;
- all Pods needed to make the Kubernetes cluster secure, observable and stable;
- continuous delivery (CD) tool;
- additional platform services, such as databases, message queues and in-memory caches.

Outside the scope of the Kubernetes Platform are:

- application code and Pods;
- continuous delivery (CD) code;
- continuous integration (CI) code and tool;
- identity provider (IdP);
- infrastructure, e.g., the OpenStack or VMware vSphere installation needed to instantiate VMs, load-balancers or object storage buckets.

A common struggle that we see is demarcation around CI/CD, especially for teams that haven't worked extensively with containerization. In particular, some organizations are struggling to understand the difference between CI/CD tooling – i.e., the generic software solution – and the application-specific CI/CD code.

The most common (working!) division of responsibilities we see is the following:

- The IT systems team is responsible for setting up and maintaining the CI tool, e.g., GitLab CI or GitHub Actions.
- The platform team is responsible for setting up and maintaining the CD tool, e.g., ArgoCD.
- The application team is responsible for:
    - maintaining the CI code, i.e., building container images from source code.
    - maintaining the CD code, i.e., pushing container images into production.

Don’t Neglect Things Outside the Scope…

Although many things are outside the scope of the Kubernetes platform, they should still be audited regularly to ensure they don’t act as the weak link which may compromise security. In our experience, many platform teams struggle because they end up securing the Kubernetes platform from the application team or against the cloud provider, instead of working with these teams to achieve a common security goal. These are questions that you may need to ask:

- **Application audit:** Is the application written in a way which ensures it can run securely and stably on a Kubernetes platform? Make sure that all application team members are aware of the Principles for Designing and Deploying Scalable Applications on Kubernetes.

- **Provider audit:** Is the underlying cloud infrastructure offering sufficient protection? To answer this question, we recommend conducting a provider audit on a yearly basis, focusing on:
    - technical aspects, e.g., does the provider offer encryption at rest;
    - organizational aspects, e.g., does the provider follow an Information Security Management System (ISMS), such as ISO 27001;
    - legal aspects, e.g., is the provider subject to a jurisdiction with insufficient data protection.

…and the Organization Around your Kubernetes Platform

Don’t forget that a Kubernetes platform is only as secure as the organization around it. It is important to set the right “tone at the top” (yes, I mean the CEO). Security must be the number one, two and three priority of the company. Make sure you recruit and retain people that are taking security seriously.

Finally, let’s not forget “laptop hygiene”. For example, full-disk encryption can help secure Kubernetes clusters against an admin losing their laptop. Within 15 years of my career, it happened to me only once, and I was glad the laptop was well protected so production Kubernetes clusters were not at risk.

Now that we set the scope, let us dive deeper into how to enable the team to take ownership of the Kubernetes platform.

Further Reading

- Principles for Designing and Deploying Scalable Applications on Kubernetes
- Provider audit
- Privacy Law and Data Protection

2. Alerting Culture

Alerts are a way for Kubernetes clusters to ask for human attention, either by sending an email, a Slack message or making a phone vibrate.

Why is Alerting Necessary?

Different roles within your organization will see different benefits from alerting. The CEO will see alerts as a tool to avoid disappointed customers. The CISO[1] or DPO[2] will see alerting as a component of the incident management process. The platform team will see alerting as a confidence boost. Instead of fearing "unknown unknowns", the platform team can determine (by the presence or absence of alerts) the status of the Kubernetes platform.

Alerting Primer

Alerting is tricky. Too much alerting may result in alert fatigue and "crying wolf". Eventually the platform team gets used to ignoring alerts and real downtime risk is around the corner. Too little alerting may result in inability to proactively fix issues which can lead to downtime or security incidents.

Alerts are generally categorized as P1, P2 or P3.

P1 or "Needs Attention Immediately" or "2 AM" Alerts

These alerts are only relevant for 24/7 uptime. As their nickname suggests, these alerts should wake up the platform on-call engineer. Since the rest of the team might be asleep – plus a drowsy platform engineer is not very smart – a clear workflow should highlight precisely what needs to be done. In case of doubt, the platform engineer should be instructed to escalate the issue to the platform team lead or postpone resolution to business hours when more team members are available.

To avoid disturbance, P1 alerts should only be used when downtime or data loss is imminent. Example alerts include "etcd quorum lost", "disk space full" or "PostgreSQL split brain / two primary". Given their disruptive nature, P1 alerts should best be

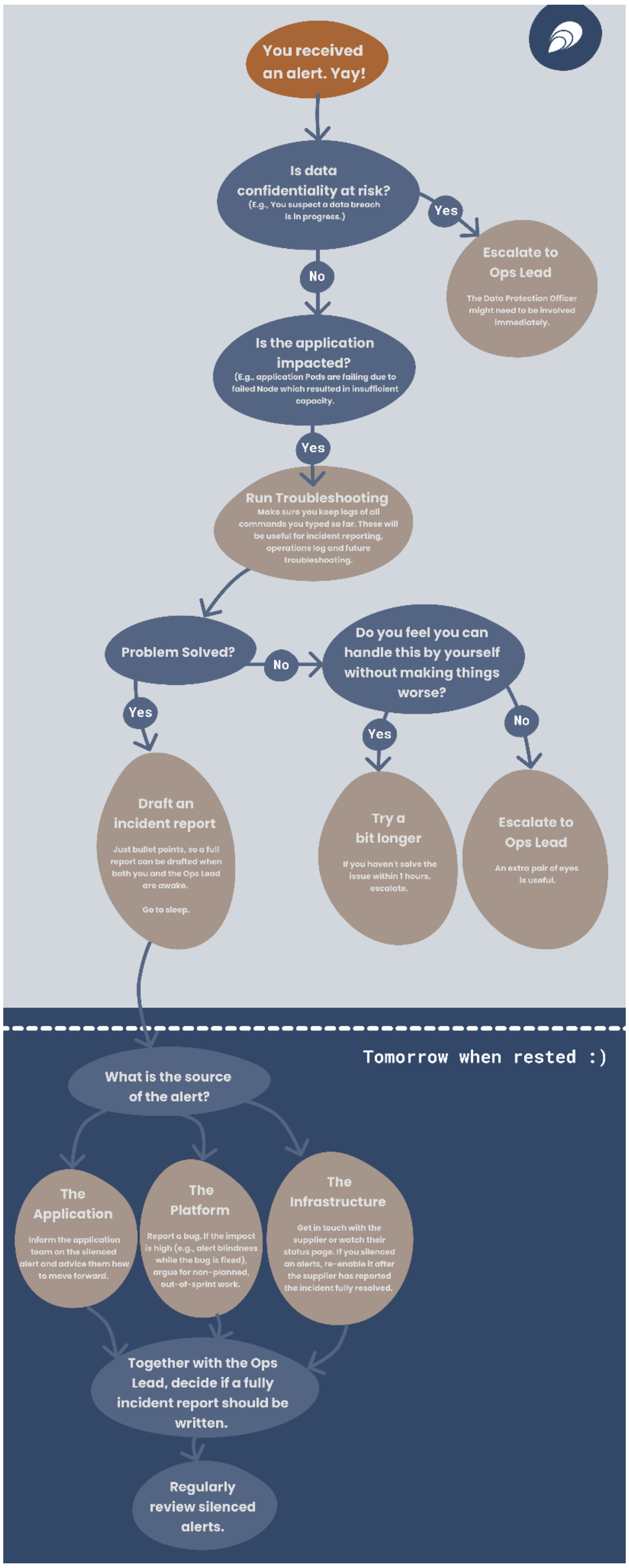
[1] Chief Information Security Officer
[2] Data Protection Officer - a role legally defined in EU GDPR, roughly equivalent to a CISO.

minimized by investing in redundancy and capacity predictions.

Example of an Alert Workflow

Let us illustrate an example of an alert workflow via the flowchart below. As you can see, the flowchart is composed of two parts: night time and day time. The flowchart includes decisions (blue circles) and actions (gray circles).

Feel free to steal [this flowchart](#) and adapt it to your own needs.


You received an alert. Yay!
Is data confidentiality at risk? (E.g., You suspect a data breach is in progress.)
Yes
Escalate to Ops Lead
The Data Protection Officer might need to be involved immediately.
No
Is the application impacted? (E.g., application Pods are failing due to failed Node which resulted in insufficient capacity.
Yes
Run Troubleshooting
Make sure you keep logs of all commands you typed so far. These will be useful for incident reporting, operations log and future troubleshooting.
Problem Solved?
No
Do you feel you can handle this by yourself without making things worse?
Yes
Draft an incident report
Just bullet points, so a full report can be drafted when both you and the Ops Lead are awake.
Go to sleep.
Yes
Try a bit longer
If you haven't solve the issue within 1 hours, escalate.
No
Escalate to Ops Lead
An extra pair of eyes is useful.
Tomorrow when rested :)
What is the source of the alert?
The Application
Inform the application team on the silenced alert and advice them how to move forward.
The Platform
Report a bug. If the impact is high (e.g., alert blindness while the bug is fixed), argue for non-planned, out-of-sprint work.
The Infrastructure
Get in touch with the supplier or watch their status page. If you silenced an alerts, re-enable it after the supplier has reported the incident fully resolved.
Together with the Ops Lead, decide if a fully incident report should be written.
Regularly review silenced alerts.

P2 or "Needs Attention Within a Business Day" Alerts

These are alerts that only notify a human during business hours. While they do disturb productivity – a person constantly interrupted by alerts will be unable to focus on code writing or improvements – they do not ruin evenings and weekends. Most importantly they do not ruin sleep.

P2 alerts can be taken "slowly" and "smartly" since other platform engineers might be available too. However, make sure to avoid the whole platform team being involved in each and every alert. Ideally, one-or-two members should be handling alerts, while the rest of the team should focus on improvements to reduce the number of future alerts.

Example P2 alerts include:

- Kubernetes Node went down, assuming redundancy ensures no application downtime.
- PersistentVolume will be full within 3 days.
- Host disk will be full within 3 days.
- Cluster will run out of capacity within 3 days.
- PostgreSQL primary failed, secondary promoted to primary.
- Nightly backup failed.

P3 or "Review Regularly" Alerts

P1 and P2 alerts need to be actionable. There needs to be something that the platform engineer can do to fix them, even if that implies escalating to the application team and/or infrastructure team. P3 alerts can be used for all kinds of symptoms that are sporadic in nature, but when looked at over long periods of time, a pattern emerges.

Example P3 alerts include:

- Pods that restart, but only rarely.
- Rare CPU throttling.
- Fluentd log buffer went above a certain threshold.

But How Do I Know if an Alert Should be P1, P2 or P3?

In a nut-shell: engineer for change, not perfection. You are unlikely to get the alerting level right the first time. Even if at some point you reached "alerting Zen", you will want to tune alerting shortly afterwards.

Alerting should be fine-tunable across the Kubernetes platform components and environments. Let me give you an example of alerting categories across the platform components: A cleanup Pod failing might be P3, a backup Pod failing might be P2, while the Ingress Controller Pod failing might be P1.

A similar differentiation is usual across environments: The Ingress Controller Pod failing in a staging environment might be P2, yet the same failure in a production environment might be P1.

Who Watches the Watchers?

A frequent dilemma is what to do in case the alerting infrastructure is down. To avoid this phenomenon, make sure that the alerting system is at least one order of magnitude more resilient than the Kubernetes platform as a whole. Specifically, make sure your alerting components are replicated and allocate them capacity generously.

Furthermore, alerting systems can be configured with "heartbeats": If a heartbeat is not received on time by the on-call management tool, then the on-call engineer is notified.

Metrics-based vs. Log-based Alerting

Alerts may be triggered either based on a condition on a metric or the presence of a log line. We recommend focusing on metrics-based alerting for the following reasons:

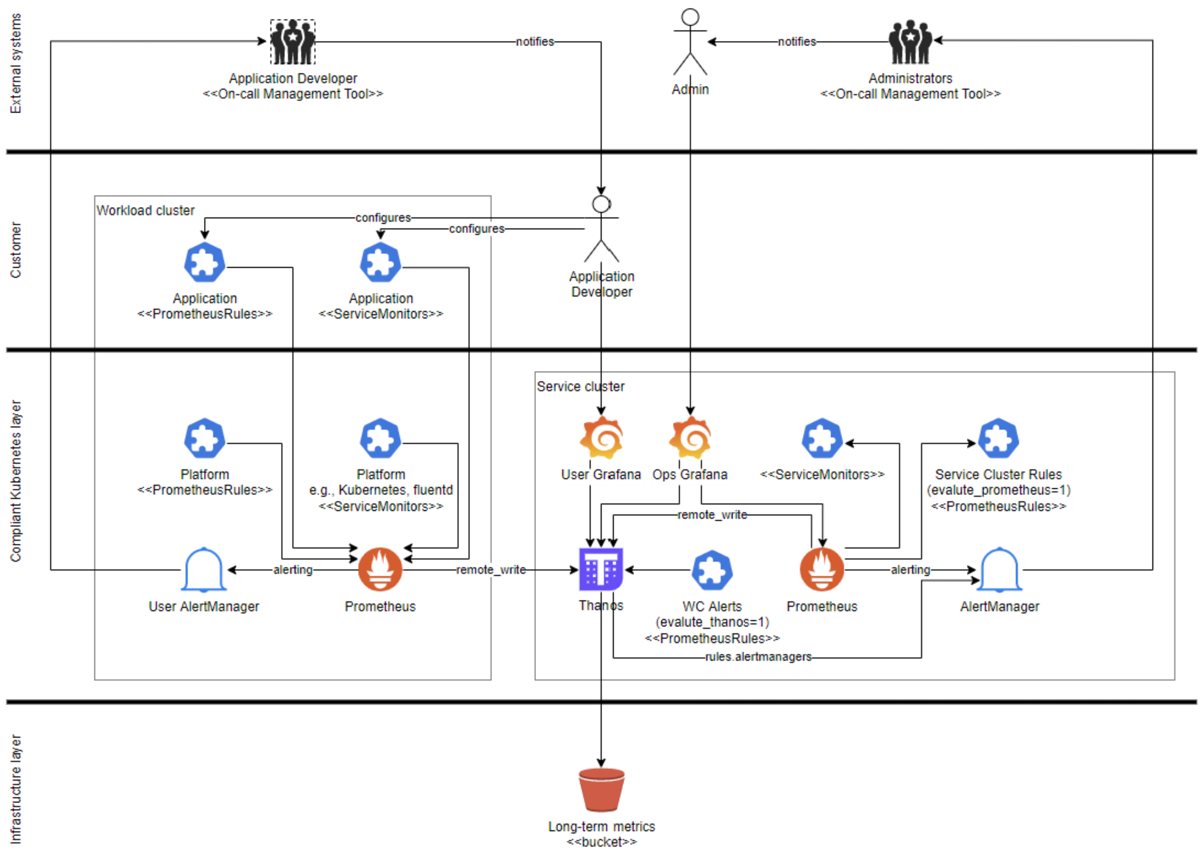
1. Collecting metrics is cheaper compared to collecting logs.
2. Collecting metrics is more robust than collecting logs.
3. Metrics can more readily be used for predictions.

How to Build Alerting?

The go-to tools for alerting in a Kubernetes platform are Prometheus and Thanos. Both are CNCF projects, which means they are open source and open governance, hence you avoid the risk of vendor lock-in.

While both Prometheus and Thanos can notify the platform team directly, we recommend using an On-call Management Tool (OMT), such as PagerDuty or OpsGenie. Both integrate well with Prometheus and Thanos. These tools considerably simplify on-call management, e.g., avoid sending P2 alerts during national public holidays. We found the OMTs to be somewhat easier to fine-tune than changing Prometheus configuration. Hence, we took the following architectural decision: Our Kubernetes platform "over alerts" and labels alerts with sufficient information on the Kubernetes platform component and environment. Then, we configure alerting fine-tuning in the on-call management tool.

The diagram on the next page shows how alerting works in Elastisys Compliant Kubernetes.



Your aim should be to bring alerting to a manageable level, not by “sweeping problems under the rug”, but by constantly improving platform stability and security, as well as tuning alerting.

So what do you do with a really tricky alert? Next section will discuss how to ensure that your platform team always has an emergency exit.

Further Reading

- Prometheus
- Awesome Prometheus Alerts
- Google SRE book
- Why we selected Thanos for long-term metrics storage?
- Configure Alerts in On-call Management Tool
- What was observability again?
- Zen and the Art of Application Dashboards

3. Disaster Recovery Training

Disaster recovery is a fancy way of saying “all other measures have failed, we need to restore from backups”. Too many organizations end up paying the ransom after a ransomware attack, so it is obvious that organizations neglect disaster recovery. This section discusses how to go from “we make backups – and hope for the best” to “we know exactly what to do in case of a disaster”.

Why Disaster Recovery Training?

Airline pilots train countless failures in a flight simulator every year. Why? For two reasons. First, airline incidents leave little room for thinking and often need to be executed from memory. Second, airline incidents are rare, hence it is impossible “to be ready for them” except by preparing.

Although Kubernetes technology is rapidly maturing, we noticed that platform teams are often denied the training they need to safely recover from a disaster. I often ask teams “So how did your last disaster recovery drill go?” and the room stays silent … uncomfortably silent. Although training for an unlikely scenario – i.e., a disaster – may seem like a waste of time, here is why “we make backups” is simply not enough:

- **Incidents are rather stressful**. The team’s stress level can be reduced by making sure that there is always a reliable, known-good last resort to resolving any incident: disaster recovery.
- **Disaster recovery is complex and stressful.** And Kubernetes admins are not very smart when they are stressed. Hence, most disaster recovery needs to be executed by mechanically following a runbook and/or executing scripts. But how do you know that the runbook and/or scripts are up-to-date? Disaster recovery training.
- **Disaster recovery training boosts team confidence and morale.** This happens naturally as a consequence of having eliminated the stress associated with disaster recovery.
- **Disaster recovery training allows you to measure** the time from a disaster’s start to the time when all systems are completely recovered. (See discussion on RTO below.)
- **Some data protection regulations make it clear that disaster recovery training is mandatory** (e.g., Swedish Patient Data Ordnance – HSLF 2016:40).

Hence, if none of the arguments above convinced you, perform disaster recovery training to please your auditors and avoid fines.

Disaster Recovery Primer

Let us go deeper into what disaster recovery is. Perhaps the best way to do this is to contrast it with business continuity. Business continuity means taking measures to prevent incidents from becoming a disruption. For example, it is known that servers fail. Hence, by adding enough redundancy – e.g., running 3 Kubernetes control-plane Nodes – you ensure that a routine event does not cause downtime.

Disaster recovery can be seen as a way to deal with the residual risk of business continuity. No matter how hard you try, there will always be events for which you are not prepared, either due to ignorance, ineptitude or cost. For example, no matter how many safeguards you put in place and how many certificates you shower over your team, human error – perhaps of someone outside your platform team – will always be a factor.

When talking about disaster recovery, one measures three parameters: Recovery Point Objective (RPO), Backup Retention and Recovery Time Objective (RTO).

Recovery Point Objective (RPO) measures what is the maximum amount of data you may lose in case of a disaster. For example, assume you run nightly backups. If the data-center hosting your Kubernetes cluster catches fire right before the backup is completed, then you lost the data generated within the last 24 hours. Hence, your RPO is 24 hours. If you take backups every 6 hours, then your RPO is 6 hours. While not commonly used with Kubernetes clusters, there are techniques for constantly streaming incremental backups – i.e., backups from the last full backup – which can reduce RPO to as low as 5 minutes. RPO in essence gives the upper bound of your recovery window. Generally, the lower the RPO, the lower the risks. However, lower RPO also entails higher costs, both in terms of storage cost of the backups themselves and in terms of overhead for taking the backups.

Backup retention determines the lower bound of your recovery window. For example, if you retain 7 nightly backups, then you can restore from as far back into the past as 168 hours. Long retention reduces data loss risk, however, it causes problems with GDPR “right to be forgotten”. Specifically, for data stored in the recovery window, you need to remember to forget.

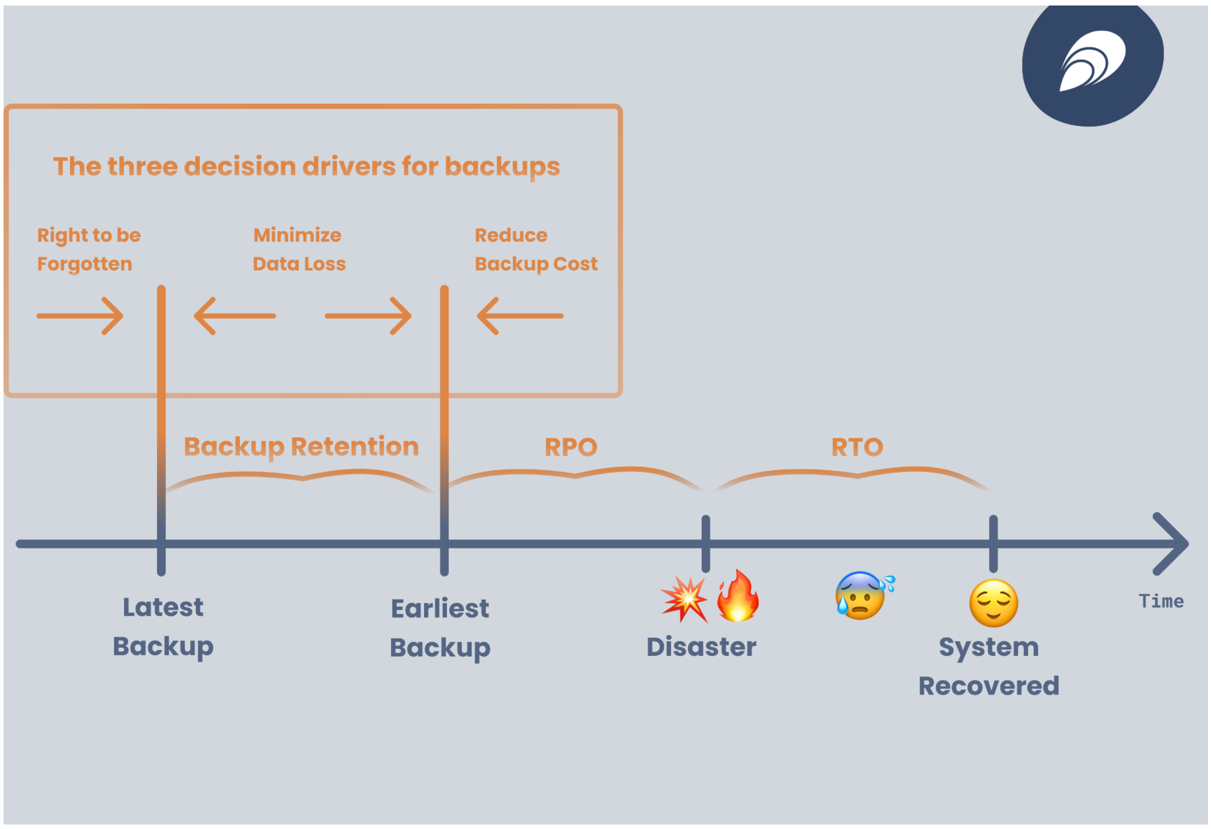
Recovery Time Objective (RTO) is the amount of time in which you aim for a recovery to be completed after a disaster happens.

RPO, backup retention and RTO are decisions to be taken on an organizational level, not merely implementation details that the platform team can determine. They are generally taken by looking at applicable regulations and making a risk assessment.

There are three decision drivers to keep in mind:

- "Right to be forgotten" will push for shorter retention times.
- "Backup cost" will push for higher RPO.
- "Minimize data loss" will push for longer retention times and lower RPO.

Typical choices include a 24 hour RPO, 7 days backup retention and 4 hours RTO.



Building Blocks for Backups

When designing backup for a Kubernetes platform, be mindful that there are really two high-level approaches: use application-specific backup; use application-agnostic backup. Application-specific backup – if available – tends to be smaller and faster. The application knows best the semantics of its data and how to avoid backing up redundant data. On the downside, application-specific backup is – as the name suggests – application-specific, hence the platform team needs to implement and train a solution for each application, which adds overhead.

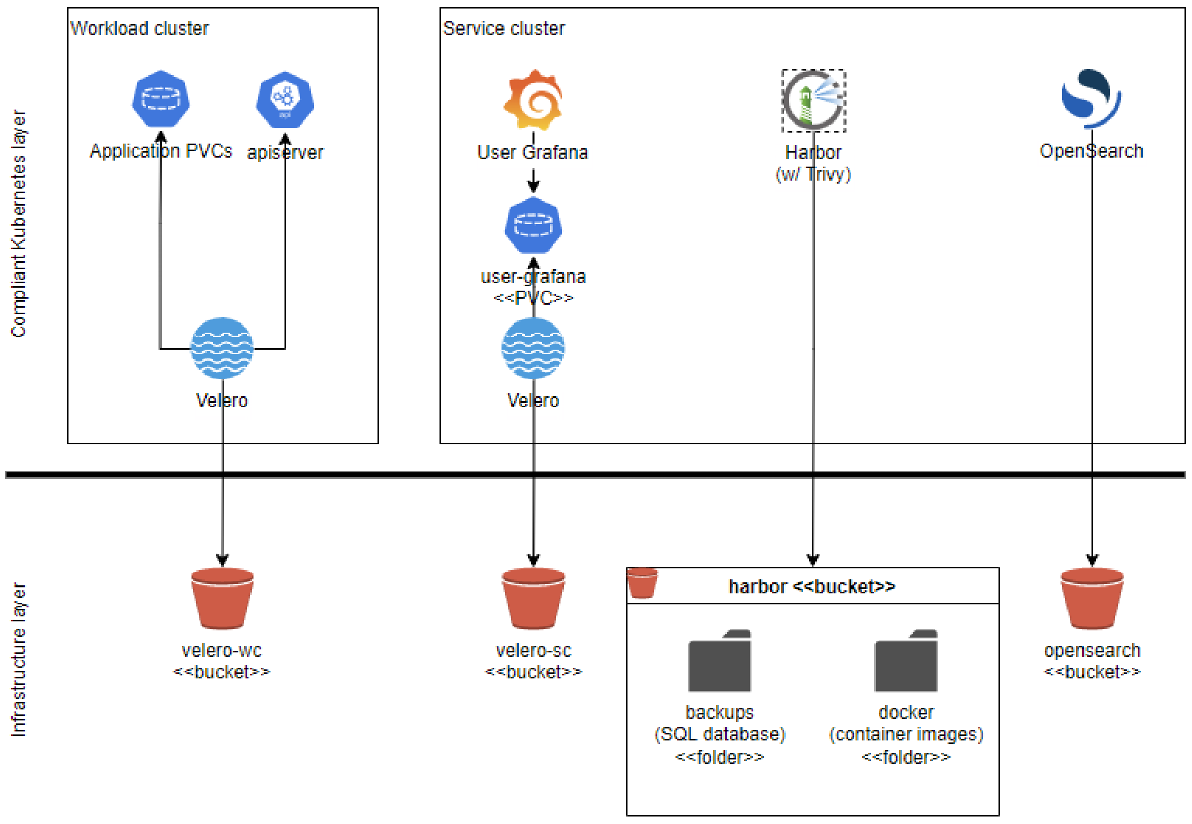
Application-agnostic backups try to use a "one size fits all" solution. For example, Velero can be used to back up PersistentVolumes of any application running inside a Kubernetes cluster. This reduces tooling sprawl, but backups tend to be a bit larger. For example, by default, Grafana stores dashboards into an SQLite database. If a large amount of dashboards are removed, then the associated SQLite file keeps its size, but is filled with "empty" pages, which still use up backup space.

Application-specific and application-agnostic backups can be combined based on the skills and needs of the platform team.

For backup storage, we recommend standardizing on S3-compatible object storage. Why? Object storage has the following advantages:

- More reliable: object storage offers a simpler API than a filesystem, hence object storages tend to be simpler and more reliable.
- Cost-effective: object storage tends to be cheaper, since it is a very thin layer on top of disks and doesn't need fully-featured CPU and memory-hungry VMs.
- Simpler synchronization: object storage only allows "full object writes", hence simplifies synchronization, e.g., to off-site locations.
- Available: object storage is widely available, either on US cloud providers, EU cloud providers and on-prem.

The diagram below illustrates typical building blocks for backups.



Protection of Backups

Too often, companies end up paying the ransom of a ransomware attack, due to two reasons:

1. **Unknown or unacceptably high RTO.** If one were to trust the attackers, then paying the ransom would reduce RTO to zero.
2. **Insufficiently protected backups.** The attackers rendered backups unavailable, so paying the ransom is the organizations only choice for disaster recovery.

To avoid the latter, it is important for the Kubernetes platform backups to be sufficiently protected. The following methods can be used:

- **Object lock:** Configuring the object storage with object lock ensures that an object – i.e., the backup – cannot be removed or modified before a certain time window passes.
- **Off-site replications:** A script can regularly copy backups from one location to another.
- **Off-cloud replications:** A script can regularly copy backups from one cloud provider to another.

Object lock, off-site and off-cloud replication can be combined to maximize backup protection. In case of off-site or off-cloud backups, it makes sense to encrypt these to ensure data cannot be exfiltrated via backups. Rclone can be readily used for this purpose.

How to Conduct Disaster Recovery Drills?

Simply stating "let's do disaster recovery" is not enough. A disaster recovery needs to be carefully planned. In particular, the following questions need to be answered:

- What should the disaster recovery drill achieve? Examples include:
  - Verifying backup scope;
  - Clarifying backup demarcation between application team and platform team;
  - Increasing platform team confidence;
  - Making sure documentation is up-to-date;
  - Measuring RTO.
- Who should be part of the disaster recovery drill?
  - Inviting the whole platform team might be overkill, so perhaps only a part of the team should be involved on a rotating basis.
  - Consider having joint drills with the application team.
- What should be the state of the environment before the disaster?
  - Make sure to include a realistic application. Training how to recover an empty Kubernetes platform defeats the purpose of disaster recovery drills. Perhaps the application team can help.
- What is the scenario?
  - What caused the disaster? For example: "We will kill the VMs hosting Thanos." or "We will assume a complete loss of one data-center, and need to recover to a new data-center."
  - How did the platform team find out about the disaster? For example: "We have received a ticket that the application team lost metrics. Please investigate and restore the metrics."
  - Who should lead the disaster recovery?
  - Who should observe and help out disaster recovery?

The disaster recovery drill should include a retrospective, which answers three questions:

- What went well?
- What to improve?

- What action items are needed for making the necessary improvements?

The platform team can then systematically work on the action items to continuously improve the disaster recovery posture of the Kubernetes platform.

Common Disaster Recovery Issues

Let us share the most common issues we discovered during disaster recovery drills:

- Application team was uncertain about the backup scope, e.g., which databases and which PersistentVolumeClaims are backed up.
- Application sometimes crashes if restored from backups and needs a few manual recovery steps.
- Velero does not restore the service account token in certain conditions (see example) which may break integration with a CI/CD pipeline.
- Velero restored resources in the "wrong" order, e.g., Deployments before NetworkPolicies. Some security-hardened Kubernetes distributions, including Elastisys Compliant Kubernetes, do not allow creation of Deployments that are not selected by a NetworkPolicy.

Alerts and disaster recovery equip your platform team with tremendous confidence and empowers them to make high-paced changes while risking little downtime and zero data loss. The next section will discuss the most important – and perhaps also the most neglected – changes that a platform team needs to make.

Further Reading

- Backup Kubernetes – how and why
- Velero
- Compliant Kubernetes Disaster Recovery

4. Maintenance

Ask anyone and they will tell you that maintenance is a complete waste of time … until a bridge collapses. You might think that platform engineering fares better than civil engineering, and you would be wrong. DataDog's Container Report 2020 found that the most popular Kubernetes version at that time already reached End-of-Life (EOL). In other words, not only were those Kubernetes clusters likely vulnerable, they didn't even receive security patches anymore.

Unfortunately, this issue is too prevalent in many orgs running Kubernetes. Even if you are not pressured by tight data protection regulations, your customers' data must be protected by process and not luck.

So let us look closer at how to do maintenance.

Maintenance Primer

In the context of this guide, maintenance refers to applying security patches and updates. Let us zoom in a bit on these concepts.

At the end of the day, your Kubernetes platform is a dependency in a larger software system. To avoid "dependency hell" and teams stepping on each other's toes, Semantic Versioning argues for the following approach. First, you need to declare a scope. Then, depending on how your platform changes towards your application developers, your team distinguishes between:

- Security patches, which are minimal changes to a software required to address a vulnerability.
- Minor updates that add functionality in a backwards-compatible manner, i.e., application developers won't notice.
- Major updates that add functionality in a potentially backwards-incompatible way, i.e., application developers will notice.

At the very least, you must perform security patches, which are small and rather non-risky. However, eventually your Kubernetes version will reach End-of-Life (EOL) when it will no longer receive security patches. Hence, you should plan for minor and major updates as well.

Maintenance can be done "just in time" – e.g., when a security patch is available – or periodically. We recommend doing maintenance periodically[3]. Setting a monthly maintenance window is beneficial for a few reasons:

1. It creates good habits and avoids the platform team having to take a decision every time. ("Should we do maintenance next week?")
2. "Consistency reduces complexity". This quote is used amongst others by the agile manifesto to argue for having a daily stand-up at exactly the same time and location, with exactly the same format. Maintenance is already complex; no need to add a layer of complexity by changing when they occur.
3. It avoids bulking together a large number of changes or migration steps, which increases downtime risk.
4. It avoids being surprised by End-of-Life (EOL).
5. It makes maintenance easy to communicate and agree with external stakeholders, in particular the application team.

How to Perform Maintenance?

Search on the Internet for "Kubernetes upgrades" and you will likely bump into the latest incarnation of GitOps. And while automation is an important tool to reduce maintenance burden, we found that the challenges with maintenance are often around and not with the maintenance itself.

First, make sure you agree on a maintenance window with external stakeholders, in particular the application team. Ideally, the maintenance window should be large enough to make room for complex maintenance without needing renegotiation. Again, "consistency reduces complexity".

For major changes, make sure to inform the application team well in advance, at least 1 month, but perhaps even 2 months ahead. If budgets allow and avoiding downtime is a must, provide the application team with two Kubernetes platform environments: staging and production. These should receive maintenance on a skewed maintenance schedule. Ideally, you should give the application team enough time to check their application in an updated staging environment, before updating the production environment.

[3] Critical security patches should be applied immediately. By "critical" it is usually understood that a working exploit is known to exist and can be used to compromise platform security. Generally, such "drop all" situations should be avoided by investing in defense in depth.

At first, maintenance will be a very manual process, so it needs to be properly prepared. Make sure you decide what to do during the maintenance window:

- Should you update Kubernetes? The Operating System (OS) base image? System Pods? All?
- Consider starting with fewer things to update until you realize that you are underutilizing the maintenance window.
- Make sure to have a contingency plan. What will you do in case something goes wrong during maintenance?
- After each maintenance, make sure to make a retrospective and inject improvements into your quality assurance process.

Automating Maintenance

Hopefully, your platform team will find that maintenance becomes "boring" after some time. Then, it's time to automate it. Here is how:

- For automating OS updates, we recommend the unattended-updates package and kured – if you want or have to update Nodes in-place or Cluster API if you'd rather replace Nodes. Whatever solution you choose here, make sure it safely drains Nodes to reduce application downtime.
- For Kubernetes updates, we recommend using Cluster API.
- For system Pods, e.g., fluentd, we recommend Tekton. The reason why we chose Tekton deserves a post in itself. For now, let's just say that, although there are plenty of other solutions, we found Tekton to be the most suitable for our needs.

Make sure not to confuse platform Continuous Delivery (CD) with application CD. The former implies updating functionality offered by the Kubernetes platform itself and might include changing the Kubernetes API via CustomResourceDefinitions (CRDs), Webhooks, etc. The latter should ideally only consist in deploying a handful of namespaced resources, like Deployments, Services and Ingresses. Depending on how you set the scope of the Kubernetes platform, the application CD might be maintained by the platform team, but configured by the application team. All-in-all, a solution which fulfills application CD requirements might be suboptimal for platform CD. Hence, you should really think of choosing one solution for platform CD and one solution for application CD, even though you might assess that one solution sufficiently fits both needs.

Maintenance Issues We Encountered

Let us share with you some of the issues we encountered during maintenance.

- Insufficient replicas: An application may be configured with a single replica, which causes downtime when the Node hosting the single application Pod is drained.
- Lack of Pod Topology Spread: An application may be configured with two replicas, but the Kubernetes scheduler placed both Pods on the same Node. Needless to say, this causes a short downtime while the Node hosting both application Pods is drained.
- Draining takes too long: This is a symptom that the application is not properly handling SIGTERM and needs to be killed non-gracefully. By default, this adds at least 30 seconds per Node. Unfortunately, we discovered Helm Charts which set terminationGracePeriod to as high as 120 seconds which, multiplied by the number of Nodes which need to be drained, greatly impact the duration of the maintenance.
- Draining is not possible: This is often the case with badly configured PodDisruptionBudgets.
- Insufficient capacity in the cluster: If you update Nodes in-place, you must drain them, which temporarily reduces cluster capacity. Make sure you have extra capacity in the cluster to tolerate a Node leaving.

To avoid the application team and/or the platform team getting surprised by such maintenance issues, we recommend going through a go-live checklist before going to production.

Further Reading

- Kured
- Pod Disruption Budget
- Pod Topology Spread Constraints
- Pod Lifecycle / Pod Termination
- Safely Drain a Node
- Elastisys Compliant Kubernetes Go-Live Checklist

Takeaways

- **Set the scope** to clarify what your platform team needs to deliver and what it is provided with.
- **Foster the right alerting culture** to ensure your team is aware but not overwhelmed with symptoms which may lead to security and stability problems.
- **Practice disaster recovery** to ensure your team can always recover the Kubernetes platform to a known-good state.

With the right scope, right alerting culture and right maintenance, you should be well equipped to maintain the stability and security of your Kubernetes platform against all “known knowns”. In a future guide we will discuss how to security-harden your Kubernetes platform against “known unknowns” and “unknown unknowns” with restrictive NetworkPolicies, intrusion detection and log reviewing.

elastisys

Elastisys AB • elastisys.com