CLOUD NATIVE COMPUTING FOUNDATION

BLOG

Principles for designing and deploying scalable applications on Kubernetes

CNCF Member Blog Post

Posted on February 17, 2022 By Elastisys' team

Guest post originally published on Elastisys' blog by the Elastisys' team

Designing scalable cloud native applications requires considerable thought, as there are many challenges to overcome. Even with the great clouds we have today for deploying our applications, the infamous fallacies of distributed computing are still true. Indeed, the network will cause both slowdowns and errors. Cloud native applications, which are often microservices, must be *designed* and *deployed* specifically to overcome these challenges.

To help us, we have at our disposal a large ecosystem of great software targeting Kubernetes. Kubernetes is not a "middleware" in the traditional distributed systems sense, but it does provide a platform for very exciting software components that help us write resilient, performant, and well-designed software. By designing software to *specifically* take advantage of these features, and by deploying them in a way that does the same, we can create software that can truly scale in a cloud native way.

In this article, I present 15 principles on how to design cloud native applications and deploy them on Kubernetes. To get the most out of it, you should **also** read three other articles. The first one concerns how to design scalable applications in general: Scalability Design Principles. The other two concern how we deploy applications, and how they collaborate in a cloud native way: the 12 Factor App manifesto and the research paper "Design patterns for container-based distributed systems". Don't be fooled by that they are all a few years old: they have stood the test of the time and are still highly relevant.

15 Principles

First: What are cloud native applications, anyway?

The Cloud Native Computing Foundation has a formal definition of what "cloud native" means. The main part is in the following paragraph:

"These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil."

Translating this into actionable and concrete characteristics and properties means that cloud native software:

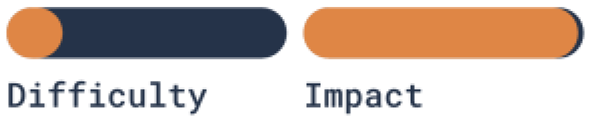
- can run multiple instances of a component, to ensure high availability and scalability;
- consists of components that depend on the underlying platform and infrastructure for scaling, automation, capacity allocation, failure handling, restarts, service discovery, and so forth.

Cloud native software is also predominantly built around a **microservice** inspired architecture, where components handle well-defined tasks. One large effect of this, and because it makes scaling and operations of such components significantly easier, is that components are largely divided into **stateful** and **stateless**. The majority of components of a large architecture are typically stateless, and depend on a few data stores to manage their application state.

Principles for Designing and Deploying Scalable Applications on Kubernetes

Kubernetes makes deploying and operating applications easy. Given a container image, all you need to do is type a single command to deploy it, even in multiple instances (kubectl create deployment nginx –image=nginx –replicas=3). However, this simplicity is both a blessing and a curse. Because an application deployed in this way is unable to take advantage of advanced features in Kubernetes, and therefore, makes suboptimal use of the platform itself.

Principle 1: A single Pod is pretty much never what you want to use

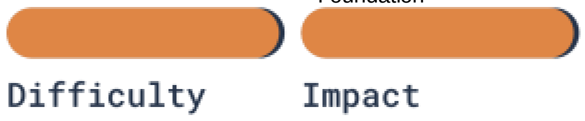


Because Kubernetes gets to terminate *Pods* whenever it needs to, you almost always need to have a Controller that looks out for your Pods. Single straight up Pods are rarely used outside of one-off debugging.

Replica Sets are also pretty much never what you want to use directly, either.

Rather, you ought to have a Deployment or StatefulSet handle Pods for you. That applies regardless of whether you intend to run more than one instance or not. The reason you want that automation is because Kubernetes makes no guarantees about the continued lifecycle of a Pod in case of a container within it failing. In fact, it clearly documents that the Pod will then just be terminated.

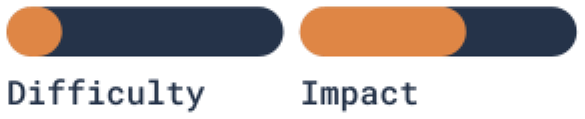
Principle 2: Clearly separate stateful and stateless components



Kubernetes defines many different resources and Controllers that manage them. Each has their own semantics. I've seen confusion around what a Deployment vs. StatefulSet vs. DaemonSet is, and what they can or cannot do. Making use of the right one means that you express your intent clearly, and that Kubernetes can help you accomplish your goals.

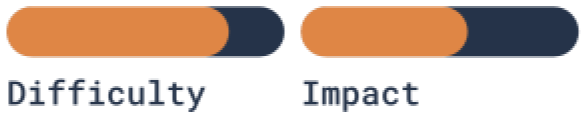
When used correctly, Kubernetes forces you to do this, but **many** convoluted solutions exist out there. The simple rule of thumb is to let everything stateful be in a StatefulSet, and stateless in Deployments, because doing this is the Kubernetes way. But do keep in mind that other Kubernetes Controllers exist, and read up on their respective guarantees and differences.

Principle 3: Separate secret from non-secret configuration to make use clear and for security



There are very few technical differences between a ConfigMap and a Secret. Both in how they are represented internally in Kubernetes, but also in how they are used. But using the right one signals **intent**, in addition to being easier to work with if you use role-based access control. It is much clearer that, e.g., application configuration is stored in a ConfigMap and then a database connection string with credentials belongs in a Secret.

Principle 4: Enable automatic scaling to ensure capacity management



Just like all Pods are realistically managed by a Deployment and fronted by a Service, you should also always consider having a Horizontal Pod Autoscaler (HPA) for your Deployments.

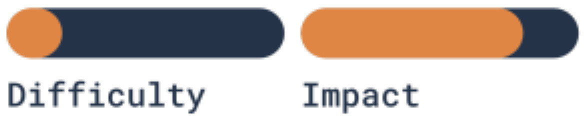
Nobody, ever, wants to run out of capacity in their production environment. Similarly, nobody wants end users to suffer because of poor capacity allocation to Pods. Preparing for this already from the start means you will be forced into the mindset that scaling can and will happen. That is far more desirable than to run out of capacity.

As per the general Scalability Design Principles, you should already *be prepared to* run multiple instances of each of your application components. This is crucial for availability and for scalability.

Note that you can scale a StatefulSet automatically with the HPA, as well. However, stateful components should typically only be scaled up if it is absolutely needed.

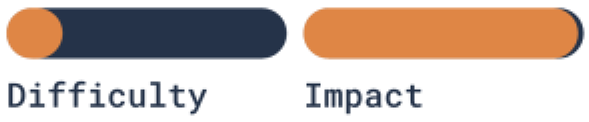
Scaling, for instance, a database can cause a lot of data replication and additional transaction management to occur, which is *definitely* not what you want if the database is under heavy load already. Also, if you do auto-scale your stateful components, consider disabling scaling down auto-matically. In particular, if the stateful component will need to synchronize with the other instances in some way. It is safer to trigger such actions manually instead.

Principle 5: Enhance and enable automation by hooking into the container lifecycle management



A container can define a PostStart and PreStop hook, both of which can be used to carry out important work to inform other components of the application of the new existence of an instance or of its impending termination. The PreStop hook will be called before termination, and has a (configurable) time to complete. Use this to ensure that the soon to be terminated instance finishes its work, commits files to a Persistent Volume, or whatever else needs to happen for an orderly and automated shutdown.

Principle 6: Use probes correctly to detect and automatically recover from failures



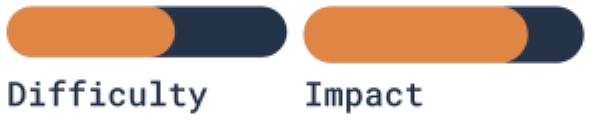
Distributed systems will fail in more and less intuitive ways than single-process systems. The interconnecting network is the cause of a large class of new causes of failures. The better we can detect failures, the better the opportunities we have to automatically recover from them.

To this end, Kubernetes provides us with probing capabilities. In particular the readiness probe is highly useful, since failure signals to Kubernetes that your container (and therefore Pod) is not ready to accept requests.

Liveness probes are often misunderstood, in spite of clear documentation. A liveness probe that fails indicates that the component is permanently stuck in a poor situation that requires forceful restart to solve. Not that it is "alive" (which is what the readiness probe indicates), because in distributed systems, liveness is actually something else.

Startup probes were added to Kubernetes to signal when to start probing with the other probes. Thus, it is a way to hold them off until performing them can start to make sense.

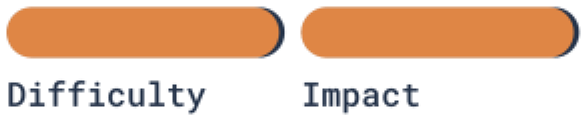
Principle 7: Let components fail hard, fast, and loudly



Make your application components fail hard (crashing), fast (as soon as a problem occurs), and loudly (with informative error messages in their logs). Doing so will prevent data from being stuck in a strange state in your application, will route traffic only to healthy instances, and will also provide all the information needed for root cause analysis. All automation the other principles in this article put into place will help keep your application in good shape while you can find the root cause.

The components in your application have to be able to handle restarts. Failures can and will happen. Either in your components, or in the cluster itself. Because failures are inevitable, they have to be possible to handle.

Principle 8: Prepare your application for observability

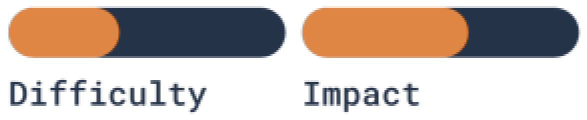


Monitoring, logging, and tracing are the three pillars of observability. Simply making custom metrics available to your monitoring system (Prometheus, right?), writing structured logs (e.g. JSON format), and *not* purposefully removing HTTP headers (such as one carrying a correlation ID), but rather making them part of what is logged, will provide your application with all it needs to be observable.

If you want more detailed tracing information, integrate your application with the Open Telemetry API. But the previous steps makes your application easily observable, both for human operators and for automation. It is almost always better to autoscale based on a metric that makes sense to your application, than on a raw metric such as CPU usage.

The "four golden signals" of site reliability engineering are latency, traffic, errors, and saturation. Tracking these monitoring signals with application-specific metrics is, empirically and anecdotally, significantly more useful than raw metrics obtained with generic resource usage measurements.

Principle 9: Set Pod resource requests and limits appropriately



By setting Pod resource requests and limits appropriately, both the Horizontal Pod Autoscaler and the Cluster Autoscaler can do a much better job. Their job of determining how much capacity is required for your Pods and of the cluster as a whole is much easier if they know both how much capacity is *required* and how much is *available*.

Do not set your requests and limits too low! This may be tempting at first, because it allows the cluster to run more Pods. But unless requests and limits are set equally (giving the Pod the "Guaranteed" QoS class), your Pods may be given more resources during normal (regular amounts of traffic) operations. That's because usually, there's some slack they can be given. So everything hums along nicely. But during peaks, they will be throttled down to the amounts you've specified. That is of course *the worst possible timing* for that to happen, and scaling up actually means you could get worse performance per Pod that is deployed. Unintentional, but also entirely what the scheduler was told to do.

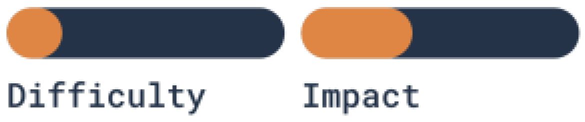
Principle 10: Reserve capacity and prioritize Pods

Difficulty
Impact

On the topic of capacity management, namespace resource quotas, reserved compute resources on your nodes, and setting Pod priorities appropriately help ensure that cluster capacity and stability is not compromised.

I've personally seen a cluster become overburdened to the point where the networking plugin's Pods were evicted. This was **not fun** (but highly educational) to troubleshoot!

Principle 11: Force co-location of or spreading out Pods as needed



Pod Topology Spread Constraints as well as affinity and anti-affinity rules are a great way to express whether you want to co-locate Pods (for network traffic efficiency) or spread them out (for redundancy) across your cloud region and availability zones.

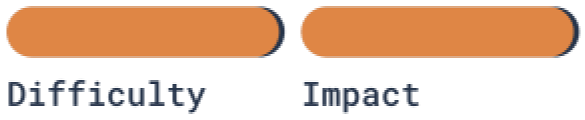
Principle 12: Ensure Pod availability during planned operations that can cause downtime

Pod Disruption Budget specifies how many of a set of Pods (e.g. in a Deployment) are allowed to be *voluntarily disrupted* (i.e., due to a command of yours, not failures), at a time. This helps ensure high availability in spite of cluster Nodes being drained by the administrator. This happens, for instance, during cluster upgrades, and those typically happen on a monthly basis, because Kubernetes moves fast.

Please be aware that if you set Pod Disruption Budgets incorrectly, you may limit your administrator's ability to do cluster upgrades. They can easily be misconfigured to block draining Nodes, which interferes with automatic OS patching and compromises the security posture of the environment.

It is greatly preferable if you can engineer your application to deal with disruptions, and to use PDB to let Kubernetes help you for the cases when this cannot be done.

Principle 13: Choose blue/green or canary deployments over stop-the-world deployments

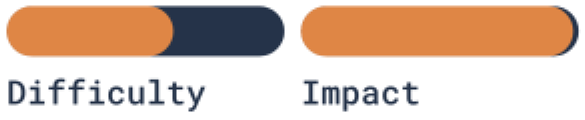


In this day and age, it is unacceptable to bring down an entire application for maintenance. This is now referred to as "stop-the-world deployment", where the application is inaccessible for a while. Smoother and more gradual changes are possible via more sophisticated deployment strategies. End-users need not be aware that the application has changed at all.

Blue/green and canary deployments used to be a black art, but Kubernetes makes it accessible to all. Rolling out new versions of a component, and routing traffic to them using labels and selectors in your Services makes even advanced deployment strategies possible to do even if you implement them more or less manually in a script of your own, and *definitely* via good deployment tools, such as ArgoCD (blue/green or canary).

Note that most deployment strategies, on a technical level, boil down to side-deploying two versions of the same component, and having requests split to them in different ways. You can either do this via the Service itself, by tagging, say, 5% of the new version's Pods with the appropriate label to make the Service route traffic to them. Or, alternatively, the upcoming Kubernetes Gateway resource will feature this out of the box (but Ingress does not).

Principle 14: Avoid giving Pods permissions they do not need



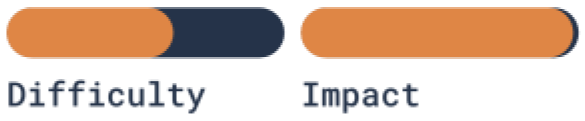
Kubernetes is not safe by itself, nor by default. But you can configure it to enforce security best practices, such as limiting what the container can do on the node.

Run your containers as a non-root user. The fact that building container images in Docker made running as root default for containers has probably brought tears of joy to hackers for close to a decade. Only use the root in your container build process to install dependencies, then make a non-root user and have that run your application.

If your application *actually* needs elevated permissions, then *still* use a non-root user, drop all Linux capabilities, and add only the minimal set of capabilities back.

Just in January of 2022, a container escape exploit that exploited a 3 year old bug surfaced (CVE-2022-0185). Containers without the required Linux capability? Completely unable to perform the attack.

Principle 15: Limit what Pods can do within your cluster



Disable the default service account from being exposed to your application. Unless you specifically need to interact with the Kubernetes API, you should not have the default service account token mounted into it. And yet, that's the default behavior in Kubernetes.

Set and enforce the strictest Pod Security Policy or Pod Security Standard you can, to make sure that you, by default, do not needlessly make insecure modes of operation possible.

Use Network Policies to limit what other Pods your Pod can connect to. The default free-for-all network traffic in Kubernetes is a security nightmare, because then, an attacker just needs to get into one Pod to have direct access to all others.

The perfect 10 CVSS scoring Log4J bug (CVE-2021-44228) humorously named Log4Shell was **completely ineffective** against containers with a locked-down Network Policy in place, which would have disallowed all egress traffic except for what is on the allowlist (and that fake LDAP server from the exploit would not have been!).

Summary

This article presents 15 principles on how to design cloud native applications and deploy them on Kubernetes. By following these principles, your cloud native application works in concert and conjunction with the Kubernetes workload orchestrator. Doing so allows you to reap all benefits offered by the Kubernetes platform, and by the cloud native way of designing and operating software.

You have learned how to use Kubernetes resources correctly, prepare for automation, how to handle failures, leverage Kubernetes probing features for increased stability, prepare your application for observability, making the Kubernetes scheduler work for you, perform deployments with advanced strategies, and how to limit the attack surface of your deployed application.

Taking all these aspects into your software architecture work makes your day to day DevOps processes smoother and more dependable. Almost to the point of being boring, one could say. And that's good, because uneventful deployment and management of software means everything just works as intended. "No news is good news", as they say.

SHARE THIS POST

[<< Previous Post: Chaos Mesh moves to the CNCF Incubator]

[Next Post: Introducing Opta: Terraform on Rails >>]