Fairwinds

WHITE PAPER

Kubernetes Best Practices

A Guide to Scalable, Secure, and Reliable K8s Management

2025

TABLE OF CONTENTS

INTRODUCTION

Kubernetes has become the cornerstone of modern cloud-native infrastructure, yet enforcing foundational best practices remains a persistent challenge for organizations of all sizes. Despite the ecosystem's growing maturity, many organizations continue to struggle with consistently applying and enforcing these practices across their environments.

As Kubernetes adoption accelerates and environments become more complex in 2025, the need for clear, actionable guidance is more critical than ever. While the cloud native industry has evolved, the fundamental best practices have remained. These foundational best practices serve as a comprehensive resource to help teams avoid common pitfalls, implement robust policies, and realize the full value of Kubernetes in a secure, reliable, and cost-effective manner.

THERE IS NO SINGLE "RIGHT" PATH TO KUBERNETES SUCCESS: INSTEAD, SEVERAL GOOD PATHS EXIST. WHICH ONE SHOULD YOU FOLLOW? THE ONE THAT BEST ADDRESSES THE NEEDS AND PRIORITIES OF YOUR BUSINESS.

Are you in the finance or healthcare sector where security is non-negotiable?

Do you have artificial intelligence or machine learning workloads that need to be deployed efficiently?

Can your applications and services tolerate downtime, or is 99.99% (or higher) reliability of paramount importance?

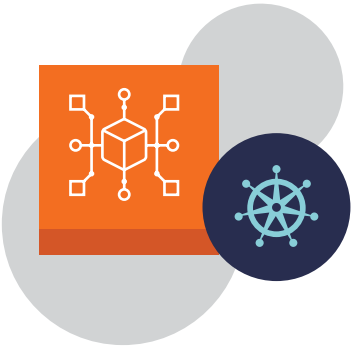
Answering these questions (and dozens more) will help you decide how to implement Kubernetes, create processes, and clarify tasks and priorities. Once you understand the bigger picture of your Kubernetes journey, you'll be better prepared to review your options and apply the best practices in this whitepaper.

Since its inception, Fairwinds has helped our clients adopt cloud-native infrastructure in a secure, efficient, and reliable manner. We've seen hundreds of different use cases and transformations, then used that experience to build, maintain, and optimize our clients' Kubernetes environments. Our guidance enables clients to focus on building apps and services, not infrastructure. Fairwinds provides Managed Kubernetes-as-a-Service, powered by expert people and best-in-class software to make your Kubernetes platform fast, secure, and stable.

This whitepaper shares hard-earned Kubernetes expertise. We dive into Kubernetes' core areas: security, efficiency, and reliability. Our goal is to provide you with Kubernetes best practices for adoption and implementation so you can realize long-term value across your entire organization.

SECURITY BEST PRACTICES

Ensuring the Security of Your Clusters



Kubernetes enables you to **manage how stateless microservices run in a cluster** because the technology enables teams to strike a balance between velocity and resilience.

With increasing adoption, security teams are becoming more concerned with securing Kubernetes infrastructure. The challenge is that many security strategies need to be implemented in code, manually audited, or both, to ensure that security best practices are followed.

As organizations transition to cloud-native technologies, including containers, microservices, and Kubernetes, the core business challenge remains the same: figuring out how to accelerate development velocity while maintaining security.

Kubernetes is becoming a mainstream solution for managing how stateless microservices run in a cluster because the technology enables teams to strike a balance between velocity and resilience. It abstracts away just enough of the infrastructure layer to enable developers to deploy freely without heavy reliance on operations teams or completely sacrificing important governance and risk controls.

But all too often, the governance and risk controls available in Kubernetes go underutilized. Since everything is working, it's easy to think that there aren't any problems. It's not until you get hit with a denial-of-service (DoS) attack or a security breach that you realize a Kubernetes deployment was misconfigured or that access control wasn't properly scoped. Running Kubernetes securely is quite complicated, which can cause headaches for development, security, and operations folks alike.

"The fastest way to set up a Kubernetes cluster is the most insecure way."

KUBERNETES SECURITY CHALLENGES AND BENEFITS

Development teams new to Kubernetes may neglect some critical aspects of deployment configuration. For example, deployments may seem to work just fine without readiness and liveness probes in place or without resource requests and limits set, but neglecting these pieces will (almost) certainly cause headaches down the line. And from a security perspective, it's not always obvious when a Kubernetes deployment is over-permissioned—often the easiest way to get something working is to give it root access. It's easy, but not secure.

While it's best to keep people out of the cluster, the reality is that **engineers need to interact with the cluster, and customers need to interact with the applications**. Optimizing Kubernetes security for all these scenarios is no easy feat.

Security will always make life a bit harder before it makes it easier. Organizations tend to do things in an insecure way at the beginning, because they don't know what they don't know, and Kubernetes is full of these unknown unknowns. It's easy to think your job is done because the site is up and working. But if you haven't tightened up your security posture in a way that adheres to best practices, it's only a matter of time before you start learning lessons the hard way.

Fortunately, Kubernetes comes with some great built-in security tooling and a robust ecosystem of open-source and commercial solutions for hardening your clusters. A well-thought-out security strategy can enable development teams to move fast while maintaining a strong security profile. Getting this strategy right is why DevSecOps is so important for cloud-native application development.

Furthermore, Kubernetes puts many pieces of computing infrastructure in one place, which helps security teams formulate a coherent strategy. This makes it much easier for security teams to conceptualize and address potential attack vectors. The pre-Kubernetes attack surface—the number of ways malicious actors could break into your infrastructure—is larger than the Kubernetes attack surface. With Kubernetes, everything is under one hood.

Optimizing Kubernetes security, however, is no easy feat, as there's not one single way to handle security in Kubernetes. It's designed to be extremely versatile to meet the complex needs of many different use cases. And while it's best to keep people out of the cluster altogether, that goal is hard to achieve because your engineers need to be able to interact with the cluster itself, and your customers need to be able to interact with the applications the cluster is running.

Kubernetes can't secure your application code. It won't prevent your developers from introducing bugs that result in code injection or a leaked or hard-coded secret or password. But Kubernetes can limit the blast radius of an attack: proper security controls will restrict how far someone can get once they're inside your cluster. For example, say an outside attacker found a vulnerability in your application and gained shell access to its container. If you have a tight security policy, they'll be stuck—unable to access other containers, applications, or the cluster at large. But if the container is running as root, has access to the host's filesystem, or has some other security flaw, the attack will quickly spread throughout the cluster. A well-configured Kubernetes deployment provides an extra layer of security.

KEY KUBERNETES BEST PRACTICES RELATED TO SECURITY:

- DoS protection
- Updates and patches
- Role-based access control (RBAC)
- Network policy
- Workload identity
- Secrets

DoS Protection

With Kubernetes, you can make sure your applications respond well to bursts in traffic, both legitimate and nefarious. The easiest way to take a site down is to overload it with traffic until it crashes—an attack known as denial-of-service (DoS). Of course, if you see a giant burst of traffic coming from one user, you can just shut off their access. But with a distributed-denial-of-service (DDoS) attack, an attacker who has access to many different machines (all of which they've probably broken into) can bombard a website with seemingly legitimate traffic. Sometimes these DoS "attacks" aren't even nefarious—it might just be one of your customers trying to use your API with a buggy script.

Kubernetes allows applications to scale up and down in response to increases in traffic. That's a huge benefit because increases in traffic won't result in end-users experiencing any degradation of performance. But if you are attacked, your application will consume more resources in your cluster, and you'll get the bill.

While services such as Cloudflare and CloudFront serve as a good first line of defense against DoS attacks, a well-designed Kubernetes ingress policy can add a second layer of protection. To help mitigate a DDoS threat, you can configure an ingress policy that sets limits on how much traffic a single IP address can consume within a specific time window before it gets shut off. You can also set limits on the number of concurrent connections; the number of requests per second, minute, or hour; the size of request bodies; and even tune these limits for specific hostnames or paths.

tl;dr | set ingress policy limits!

As painful as upgrading can be, **keeping your Kubernetes version up to date is essential**. Old versions quickly become stale, and new security holes are announced with regularity.

Updates and Patches

Kubernetes releases updates a few times a year, each fixing bugs and security holes and making improvements. As painful as upgrading can be, keeping your Kubernetes version up to date is essential. Old versions quickly become stale, and new security holes are announced with regularity.

On top of that, it's common to have several add-ons installed in your cluster to enhance the functionality Kubernetes provides out of the box. For instance, you might use cert-manager to help keep your site's external certificates up to date; service meshes (such as Istio and Linkerd) for advanced traffic management, security, and observability; or metrics-server and observability platforms (such as OpenTelemetry, Prometheus, and Grafana) to enable unified monitoring, tracing, and alerting that show how applications are running. With each of these add-ons, your attack surface grows, and your risks increase. Staying up to date on bug fixes and new add-on releases is important.

Each time a new release comes out, you'll need to test those updates to make sure they don't break anything. Where possible, test on internal and staging clusters and roll updates out slowly, monitoring for potential problems and making course corrections along the way.

Finally, be sure to keep the underlying Docker image up to date for each of your applications. The base image you're using can go stale quickly, and new Common Vulnerabilities and Exposures (CVEs) are always being announced. To fight back, you can use container scanning tools, such as Trivy, to check every image for vulnerabilities. The safest policy is to make sure the base operating system and any installed libraries are up to date.

tl;dr | update the base operating system and installed libraries—and test updates thoroughly.

RBAC

The easiest way to deploy a new application or provision a new user is to give them admin permissions. A person or application with admin permissions has free range to do whatever they want—create resources in the cluster, view application secrets, or delete an entire Kubernetes deployment. One problem with that approach is that if an attacker gains access to that account, they too can do anything they want. They could spin up new workloads that mine Bitcoin, access your database credentials, or delete everything in the cluster.

Applications, like most users, probably don't need extensive control over the cluster, so giving apps admin-level access is quite dangerous. If all an application needs to do is view logs, you can pare down its access to the minimum required so that an attacker can't do anything more than that—no mining Bitcoin, viewing secrets, or deleting resources.

Giving simple applications admin-level access is dangerous. If all an application needs to do is view logs, pare down its access to the minimum required, so an **attacker can't view secrets** or delete resources.

To manage access, Kubernetes provides role-based access control (RBAC). Use RBAC to grant fine-grained permissions to access different resources in the cluster. Setting up thoughtful Kubernetes RBAC rules according to the principle of least privilege is important for reducing the potential for splash damage if (or when) an account is compromised.

It's a delicate balance, as you might end up withholding necessary permissions. But it's worth that minor inconvenience of tweaking permissions to avoid the major headaches that come from a security breach.

While RBAC configuration can be confusing and verbose, open source tools like rbac- manager can help you simplify the syntax. This helps prevent mistakes and provides a clearer sense of who has access to what. You can also use automated policy enforcement tools, such as Polaris, Kyverno, or OPA, to ensure least-privilege policies are consistently applied.

tl;dr Set up RBAC according to the principle of least privilege.

Network Policy

Network policy is like RBAC, but instead of deciding who has access to which resources in your cluster, network policy focuses on what can talk to what inside your cluster. In a large enterprise, dozens of applications may run inside the same Kubernetes cluster, and by default, every application has network access to everything else running inside the cluster.

Some network access is usually necessary. But while a given workload might need to talk to a database and a handful of microservices, that workload probably won't need access to every other application inside the cluster.

It's up to you to write a network policy that cuts off communications to unnecessary parts of the cluster. Without a strict network policy, an attacker will be able to probe the network and spread throughout the cluster. With proper network policies in place, however, an attacker who gains access to a particular workload will be restricted to that one workload and its dependencies.

Network policy can also be used to manage cluster ingress and egress—where incoming traffic can come from and where outgoing traffic can go. You can make sure internal-only applications only accept traffic from IP addresses inside your firewall, and make sure all partner IP addresses are whitelisted

for partner-driven applications. For outgoing traffic, you may also want to whitelist allowed domains. This way, if a hacker gains access to the cluster and tries to push data out to an external URL, they'll be stopped by your network policy. With strict ingress and egress rules, you can limit the potential attack surface of your applications.

Using workload identity, **Google handles all the permissioning under the hood** using short-lived credentials, so you don't need to manage (and possibly expose) your access keys.

Network policy is easy to neglect, especially when you're building out a Kubernetes cluster for the first time. But it's an effective way to harden your cluster from a security standpoint and limit the damage when attackers find a security hole. As with RBAC, there's a trade-off between over-permissioning to make sure everything works properly and limiting permissions to make sure any problems are contained. Again, you're sacrificing short-term convenience to avoid the fallout from a major security breach.

tl;dr | write a network policy that cuts off communications to unnecessary parts of the cluster. Manage cluster ingress and egress based on IP addresses.

Workload Identity

Workload identity is a way to tie RBAC, the cluster's authentication mechanism, to the cloud provider's authentication mechanism, such as Identity and Access Management (IAM) on Google Cloud or AWS. In this way, you can use Kubernetes' built-in authentication mechanisms to manage access to resources that exist outside the cluster. For example, databases typically run outside of the Kubernetes cluster in a managed service, such as AWS's Relational Database Service (RDS). Workload identity allows a workload in your EKS cluster to connect to your RDS instance without you having to provision and manage the credentials yourself.

Without workload identity, you'd have two options, both of which have security concerns. First, you could use IAM to grant the necessary permissions to entire nodes, but this effectively grants those permissions to every workload on the node, not just the workload that needs them. Alternatively, you could generate a long-lived access key for your database, turn that key into a Kubernetes secret, and attach that secret to the workload. But each step in this process introduces the potential for leakage, and because the key is long-lived, anyone with access to that key will be able to access your database in perpetuity.

With workload identity, Google handles all the permissioning under the hood using short-lived credentials, so you don't need to manage (and possibly expose) your access keys.

tl;dr | employ workload identity to tie RBAC to the cloud provider's authentication mechanism.

One caution: workload identity works only within a particular cloud provider. For instance, Google Kubernetes Engine can use workload identity to authenticate databases on Google Cloud but not on AWS; Amazon EKS can use workload identity to authenticate AWS databases but not Google Cloud databases.

By encrypting all of your secrets, you can **safely check them into your repository** without fear of exposing them.

Secrets

Kubernetes empowers Infrastructure as Code (IaC) workflows more than any other platform. By encoding all your infrastructure choices in YAML, Terraform, or another configuration format, you ensure your infrastructure is 100% reproducible. Even if your cluster disappeared overnight, you'd be able to recreate it in a matter of hours or minutes—as long as you're using IaC.

But there's one catch: your applications need access to secrets. Database credentials, API keys, admin passwords, and other pieces of sensitive information are required for most applications to function properly. You may be tempted to check these credentials into your IaC repository, so that your builds are 100% reproducible. But once they're checked in, they're permanently exposed to anyone with access to your Git repository. If you care about security, you need to avoid the temptation to do this.

The solution is to split the difference: by encrypting all your secrets, you can safely check them into your repository without fear of exposing them. Then you'll just need access to a single encryption key to "unlock" your IaC repository and have perfectly reproducible infrastructure. Tools such as Mozilla's SOPS make this easy. Simply create a single encryption key using Google's or Amazon's key management stores, and any YAML file can be fully encrypted and checked in to your Git repository. You can also integrate external secrets management solutions (such as HashiCorp Vault or AWS Secrets Manager) to manage your secrets. You should also enforce regular credential rotation as part of your processes.

tl;dr | encrypt all your secrets. You'll then only need a single encryption key to unlock your IaC repository.

FINAL THOUGHTS ON KUBERNETES SECURITY

Applications change constantly, and there's no way to ensure that your application code is bulletproof. What Kubernetes does well is mitigate the severity of attacks and contain splash damage. When someone penetrates your application and makes it through that first layer, they won't get much (or any) farther if you've optimized security settings in accordance with the Kubernetes security best practices described here.

With the proper knowledge and attention, a Kubernetes implementation will be more secure and easier to maintain than other systems, specifically because it provides a single platform for everything related to cloud computing. Kubernetes has strong built-in security features, a massive ecosystem of third-party security tooling, and major cloud providers now offer confidential computing and node-level security features that you can adopt.

Although it can feel overwhelming to adopt Kubernetes, many tools are available that can help you manage the process. You also might explore a managed Kubernetes-as-a-Service provider that goes beyond some cloud provider managed Kubernetes solutions.

Kubernetes is a dynamic system that **automatically adapts** to your workload's resource utilization needs.

COST OPTIMIZATION BEST PRACTICES

Set Just Right CPU and Memory

One reason container technology has surpassed the capabilities of traditional virtual machines is its inherent efficiency regarding infrastructure utilization. Whereas in a traditional virtual machine environment, one application is typically run per host; in a containerized environment, you can run multiple applications per host, each within its own container. Packing multiple applications in each host reduces your overall number of compute instances and thus your infrastructure costs.

Kubernetes is a dynamic system that automatically adapts to your workload's resource utilization needs. Kubernetes has two levels of scaling. Each individual Kubernetes deployment can be scaled automatically using a Horizontal Pod Autoscaler (HPA), while the cluster at large is scaled using Cluster Autoscaler. HPAs monitor the resource utilization of individual pods within a deployment, and they add or remove pods as necessary to keep resource utilization within specified targets per pod. Cluster Autoscaler, meanwhile, handles scaling of the cluster itself. It watches the resource utilization of the cluster at large and adds or removes nodes to the cluster automatically.

A key feature of Kubernetes that enables both scaling actions is the ability to set specific resource requests and limits on your workloads. By setting sensible limits and requests on how much CPU and memory each pod uses, you can maximize the utilization of your infrastructure while ensuring smooth application performance.

To maximize the efficient utilization of your Kubernetes cluster, it is critical to set resource limits and requests correctly. Setting your limits incorrectly on an application will cause problems. For example, if your memory limits are too low, Kubernetes is bound to kill your application for violating its limits. Meanwhile, if you set your limits too high, you're inherently wasting resources by over-allocating, which means you will end up with a higher cloud bill.

SET KUBERNETES RESOURCES "JUST RIGHT"

Fairwinds created the open-source project Goldilocks to help teams allocate resources to their Kubernetes deployments and get those resource calibrations just right. Goldilocks is a Kubernetes controller that collects data about running pods and provides recommendations on how to set resource requests and limits. It can help organizations understand resource use, resource costs, and best practices around the efficiency of usage.

Goldilocks employs the Kubernetes Vertical Pod Autoscaler (VPA). It takes the historical memory and CPU usage of your workloads into account, along with the current resource usage of your pods, to recommend how to set your resource requests and limits. (While the VPA can set limits for you, it is often best to use the VPA engine only to provide recommendations.) The tool creates a VPA for each deployment in a namespace and then queries that deployment for information.

Teams that are managing multiple clusters may want visibility across their entire environment to undertake cost attribution and resource tuning at scale.

Understanding Your Workloads

Another benefit of Goldilocks is that the information provided can help you understand whether your workloads are CPU-intensive, memory-intensive, or a balance between the two. This data can help you evaluate whether you've selected the most efficient workload for your Kubernetes worker nodes.

To view these recommendations, you would have to use kubectl to query every VPA object, which could quickly become tedious for medium-to-large deployments. That's where the dashboard comes in. Once your VPAs are in place, recommendations will display in the Goldilocks dashboard.

THE DASHBOARD PRESENTS TWO TYPES OF RECOMMENDATIONS DEPENDING ON THE QUALITY OF SERVICE (QOS) CLASS YOU DESIRE FOR YOUR DEPLOYMENTS:

1. **Guaranteed,** which means the application will be scheduled on a node where resources will be assured. In this class, you set your resource requests and limits to the same values, which guarantees that the resources requested by the container will be available to it when it gets scheduled. This QoS class is appropriate to apply to the Kubernetes clusters that need to be the most stable.
2. **Burstable,** which means the application will be guaranteed a minimum level of resources but will receive more if and when available. Your resource requests are lower than your limits. The scheduler will use the request to place the pod on a node, but then the pod can use more resources up to the limit before it's killed or throttled

The dashboard provides recommendations for both the Guaranteed and Burstable QoS classes. In the Guaranteed class, consider setting your requests and limits to the VPA "target" field. In general, using this value along with the HPA allows applications to scale.

Note: a third QoS class, BestEffort, means that no requests or limits are set, and that the application will be allocated resources only when all other requests are met. Use of BestEffort is generally not recommended.

Specializing Instance Groups for Your Cluster

If you are interested in fine-tuning the instances on which your workloads run, you can use different instance group types and node labels to steer workloads onto specific instance types.

Different business systems often have different-sized resource needs along with specialized hardware requirements (such as GPUs). The concept of node labels in Kubernetes allows you to put labels on all your various nodes. Pods, meanwhile, can be configured to use specific "nodeSelectors" set to match specific node labels, which decide which nodes a pod can be scheduled onto.

By using instance groups of different instance types with appropriate labeling, you can mix and match the underlying hardware available from your cloud provider of choice with your workloads in Kubernetes.

As AI/ML workloads become more common, you can also use node selectors and taints/tolerations to schedule GPU workloads onto appropriate nodes, ensuring efficient use of specialized hardware.

If you have different-sized workloads with different requirements, it can make sense strategically and economically to place those workloads on different instance types and use labels to steer your workloads onto those different instance types.

Spot instances (from AWS) and preemptible instances (from Google Cloud) tie into this idea. Most organizations are familiar with paying for instances on demand or on reserved terms over fixed durations. However, if you have workloads that can be interrupted, you may want to consider using spot

instances on AWS or preemptible instances on Google Cloud. These instance types allow you to make use of the cloud provider's leftover capacity at a significant discount—all at the risk of your instance being terminated when the demand for regular on-demand instances rises.

If some of your business workloads can tolerate the risk of random instance termination, you can use the same concept of node labeling to schedule those workloads onto these types of instance groups and gain substantial savings.

FINAL THOUGHTS ON KUBERNETES COST OPTIMIZATION BEST PRACTICES

Setting up and managing clusters and then telling software developers to deploy their apps to those clusters is a complex process. It's not unusual for developers to deploy apps but not know how to set the right resource limits

or requests. However, Goldilocks empowers developers to remove the guesswork by automating the recommendation process for them. In turn, Goldilocks opens the door for you to increase the efficiency of your cluster, reduce your cloud spend, and start leveraging FinOps best practices.

RELIABILITY BEST PRACTICES

Avoid incorrect configuration

Reliability becomes harder and harder to achieve as your business scales. **Achieving Kubernetes reliability is complex** due to the skill it takes to optimize the capabilities Kubernetes offers.

Reliability becomes harder and harder to achieve as your business scales. Consider adopting a more direct and streamlined approach to cloud native applications and infrastructure. Containers abstract and isolate cloud native applications and their dependencies from what's running on the underlying operating system. You can scale these lighter-weight containers instead of scaling application server virtual machines. Cloud-native methodologies provide an opportunity to adjust how these application components communicate and scale.

For instance, components of your application use:

- APIs to communicate instead of sharing a common filesystem.
- Service discovery to route traffic to services as they scale.
- Containers to abstract application dependencies from the underlying operating system.

The more cloud native characteristics an application has, the easier it is to put that application in a container and manage it in Kubernetes. Another way you can ensure the reliability of your clusters is by shifting to IaC.

THE BENEFITS OF INFRASTRUCTURE AS CODE

Simply put, IaC is the process of managing your IT infrastructure using configuration files. Some of the most significant IaC advantages include:

- Reduced human error
- Repeatability and consistency
- Disaster recovery
- Improved auditability

Reduced Human Error and Future Proofing

IaC and automation reduce human error by creating predictable results. You can produce new environments to test infrastructure upgrades and validate changes without impacting production. If you want to apply changes to infrastructure across multiple environments, using IaC reduces the likelihood of introducing errors through repetitive manual work.

Because IaC documents infrastructure, it prevents reliance on individual expertise (reducing the impact of talent loss) and facilitates adaptation to technological advancements. In other words, the code and comments increase awareness about the design and configuration of infrastructure. They also help with training, reducing the need for subject matter experts to get developers up to speed.

Repeatability and Consistency

The repeatability of IaC helps you create consistent infrastructure in other regions much more rapidly. This feature frees up time to move on to the next set of problems, such as how to route traffic to applications throughout the region and how to test failover without impacts on production.

Disaster Recovery

How long does it take to rebuild a container image in an emergency (for example, deploy new code to address an application outage or degradation)? If you're using manual processes or complex chains of tooling, then that disaster recovery (DR) process will take longer. The reliability of an application is impacted by the ability to pivot and the speed to redeploy. Be sure you know what that process looks like and how to put the right practices, tooling, and underlying processes in place to make a Kubernetes deployment as straightforward as possible.

Improved Auditability

IaC also helps track changes to an audit infrastructure. Because your infrastructure is represented in code, commits to your Git repository reflect who, what, when, and why changes were made. You'll be able to look at the code and know how environments were built, what's happening, and why.

KUBERNETES RELIABILITY BEST PRACTICES

KEY KUBERNETES RELIABILITY BEST PRACTICES:

- Simplicity vs. complexity
- High-availability (HA) architecture/fault tolerance
- Resource limits and autoscaling
- Liveness and readiness probes

Simplicity vs. Complexity

Unfortunately, it's easy to introduce too much complexity into your Kubernetes environments. Here are three ways to keep it simple:

3. **Service Delivery vs. Traffic Routing**: Manually maintained DNS entries can be used to point to an application, and DNS hostnames can be hardcoded into application components so they can communicate. However, rather than using traffic routing, use service delivery, which is a more streamlined, dynamic solution. Service delivery enables a user or another application to find instances, pods, or containers. Service delivery is required because your application is scaling in and out, and changes are happening at a fast rate.
4. **Application Configuration**: Shift to files or environment variables in your container. Those are populated by Kubernetes ConfigMaps or Secrets. You can run an application in multiple environments, but the configuration will differ because you have different ConfigMaps or Secrets in Kubernetes for each environment.
5. **Configuration Management Tools:** Containers are ephemeral. If you need to change how an application runs, CI/CD best practices dictate that you build and then deploy a new container image through your CI pipeline instead of attempting to modify an existing container.

HA Architecture/Fault Tolerance

Kubernetes helps improve reliability by enabling the scheduling of containers across multiple nodes and multiple availability zones (AZs) in the cloud. Anti-affinity allows you to constrain which nodes in your pod are eligible to be scheduled based on labels on pods that are already running on the node, rather than on labels on nodes.

With node selection, the node must have each of the indicated key-value pairs as labels for the pod to be eligible to run on a node. When you create a Kubernetes deployment, use anti-affinity or node selection to help spread your applications across the Kubernetes cluster for high availability.

Kubernetes HA means having no single point of failure in a Kubernetes component. An example of a component might be a Kubernetes API server or the etcd database, where state is stored in Kubernetes. How do you help ensure these components are HA? Let's say you are using Kubernetes on-premises, and you have three master servers with a load balancer that runs on a single machine. While you have multiple masters, your one load balancer is a single point of failure for the Kubernetes API. You need to avoid this scenario.

If a redundant component in your Kubernetes cluster is lost, the cluster keeps operating because K8s best practice is to deploy several redundant instances based on the component (for example, etcd requires an odd number, so 3+, API server requires 2+, kube-scheduler requires 2+). If you lose a second component, then what happens? If you have three masters and you lose one, the two remaining masters could get overloaded, contributing to the degradation or potential loss of another master. You may also want to consider incorporating chaos engineering practices to proactively test and improve cluster resilience. Overall, it's important to plan the resiliency of your cluster according to the risk your business can tolerate for the applications running on that cluster.

tl;dr | plan your fault tolerance strategy and employ HA redundancy based on your workload.

Resource Limits and Autoscaling

Resource requests and limits for CPU and memory are at the heart of what allows the Kubernetes scheduler to do its job well. If a single pod is allowed to consume all the node's CPU and memory, then other pods will be starved for resources. Setting limits on what a pod can consume increases reliability by keeping pods from consuming all the available resources on a node (this is referred to as the "noisy neighbor problem").

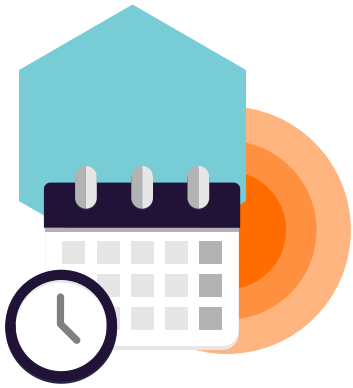
Autoscaling, in turn, can increase cluster reliability by allowing the cluster to respond to changes in load. HPA and cluster autoscaling work together to provide a stable cluster by scaling your application pods and cluster nodes.

Reliability first requires you to set appropriate resource requests and limits, and the Cluster Autoscaler will have difficulty doing its job if your resource requests are not set correctly. The Cluster Autoscaler relies on the scheduler to know that a pod won't fit on the current nodes, and it also relies on the resource request to determine whether adding a new node will allow the pod to run.

tl;dr | set limits on pod resource consumption to increase reliability and avoid the noisy neighbor problem.

Liveness and Readiness Probes

Another important facet of cluster reliability involves the concept of "self-healing." The idea here is to automatically detect and fix issues in the cluster. This concept is built into Kubernetes in the form of liveness and readiness probes.



Kubernetes helps **improve reliability** by making it possible to schedule containers across multiple nodes and multiple availability zones in the cloud.

A liveness probe indicates whether the container is running, and it is fundamental to the proper functioning of a Kubernetes cluster. If this probe is moved into a failing state, then Kubernetes will automatically send a signal to kill the pod to which the container belongs. In addition, if each container in the pod does not have a liveness probe, then a faulty or non-functioning pod will continue to run indefinitely, using up valuable resources and potentially causing application errors.

A readiness probe, on the other hand, is used to indicate when a container is ready to serve traffic. If the pod is behind a Kubernetes service, the pod will not be added to the list of available endpoints in that service until all the containers in that pod are marked as ready. This procedure allows you to keep unhealthy pods from serving any traffic or accepting any requests, thus preventing your application from exposing errors.Kubernetes helps improve reliability by making it possible to schedule containers across multiple nodes and multiple availability zones in the cloud.

Liveness and readiness probes in Kubernetes periodically assess the state of your containers to ensure they are running correctly and ready to serve traffic. Each probe has two states, pass and fail, along with a threshold for how many times the probe must fail or succeed before the state is changed. When configured correctly on all your containers, these two probe types provide the cluster with the ability to "self-heal." Problems that arise in containers will be automatically detected, and pods will be killed or taken out of service automatically. Consider also configuring startup probes for applications with complex initialization logic, as these are now widely supported and help prevent premature restarts.

tl;dr | configure liveness probes and readiness probes to give your cluster the ability to self-heal.

FINAL THOUGHTS ON BUILDING RELIABLE KUBERNETES CLUSTERS

Reliability in a Kubernetes environment is synonymous with stability, streamlined development and operations, and a better end-user experience. Many factors need to be considered when building a stable and reliable Kubernetes cluster, including the possible need for application changes and changes to cluster configuration. Steps include setting resource requests and limits, autoscaling pods using a metric that represents application load, and using liveness and readiness probes.

Reliability becomes much easier to achieve with the right configurations. Simply put, it's time to re-evaluate the pre-Kubernetes tools you use and how you use them. Consider which existing tools and processes still have a place in your environment as you move into the world of IaC, containers, cloud-native apps, and Kubernetes. Also, consider managed Kubernetes-as-a-Service as a way to take full advantage of the benefits of Kubernetes without the complexity of managing infrastructure in-house.

Managing cluster configuration becomes unwieldy fast as workloads are inconsistently or manually deployed and modified.

POLICY ENFORCEMENT BEST PRACTICES

Avoid Inconsistency in Multi-User, Multi-Cluster, Multi-Tenant Kubernetes Environments

In most cases, organizations pilot Kubernetes with a single application. Once successful, these organizations commit to Kubernetes across multiple apps, development, and ops teams. Often used in a self-service model, DevOps and infrastructure leaders will have many users across many different clusters building and deploying apps and services.

Managing cluster configuration becomes unwieldy fast as workloads are inconsistently or manually deployed and modified. Without putting policies in place, there are likely to be discrepancies in configurations across containers and clusters, which can be challenging to identify, correct, and keep consistent. These misconfigurations happen when users copy and paste YAML configurations from online examples taken from Stack Overflow or other dev teams. Often, workloads are over-provisioned "just to get things to work," or there are no existing processes in place to verify configurations.

Manually identifying these misconfigurations is highly error-prone and can quickly overwhelm platform and engineering teams with code review.

WHEN MANAGING MULTI-CLUSTER ENVIRONMENTS WITH A TEAM OF ENGINEERS, CREATING CONSISTENCY REQUIRES YOU TO ESTABLISH KUBERNETES POLICIES TO ENFORCE DEVELOPMENT BEST PRACTICES.

1. **Standard Policies:** enable best practices across all organizations, teams, and clusters. Examples include disallowing resources in the default namespace, requiring resource limits to be set, or preventing workloads from running as root.
2. **Organization-Specific Policies:** enforce best practices that are specific to your organization. Examples include requiring labels on each workload, enforcing a list of allowed image registries, or setting policies that help with compliance and auditing requirements.
3. **Environment-Specific Policies:** enforce or relax policies for particular clusters or namespaces. For example, include stricter security enforcement in prod clusters or looser enforcement in namespaces that run low-level infrastructure.

When Does Policy Enforcement Make Sense

- **Shared Cluster:** You're building or running a shared Kubernetes cluster that serves multiple app teams.
- **Multi-Cluster:** You're running multiple clusters, with plans to expand your footprint in the cloud, on-prem, or both.
- **Service Ownership:** Development teams own all things application and operations and want to avoid mistakes that distract from building their app.

Simply putting a best practices document in place for your engineering team doesn't work — it will be likely forgotten or ignored. By creating your golden path or IDP that includes Kubernetes guardrails and policy enforcement, common misconfigurations will be prevented from being deployed into the cluster, enables IT compliance and governance, and allows teams to ship with confidence knowing that guardrails are in place.

KUBERNETES POLICY ENFORCEMENT OPTIONS

There are several options available for enforcing Kubernetes policies.

Develop Internal Tools

Engineers like to develop their own tools for a problem; however, leaders need to decide whether their team can spend the time, money, and resources developing and maintaining home-grown tooling rather than working on problems specific to their business.

Deploy Open Source

Multiple open-source tools are available that can help with security, reliability, and efficiency configuration. These include open-source auditing tools for container scanning and network sniffing, as well as Fairwinds' own contributions for auditing Kubernetes clusters, such as Polaris, Goldilocks, Nova, and Pluto.

Polaris comes with 30+ built-in checks around security, efficiency, and reliability.

Some Polaris checks include:

- Whether a readiness or liveness probe is configured for a pod
- When an image tag is either not specified or set to pull the image with the latest tag from the configured registry
- When the hostNetwork or hostPort attribute is configured
- When memory and CPU requests and limits are not configured
- When securityContext.privileged is true or when securityContext.readOnlyRootFilesystem is not true (one of several other security configuration checks)

If you select the open-source route, your team will spend time deploying and managing each tool. You'll need to ask whether your team has the bandwidth for this work and whether it will enable you to focus on the apps or services that differentiate your business from the competition. For both internal and open source options, consider GitOps-driven policy management tools like Flux or Argo CD for declarative, auditable enforcement.

You may also choose to purchase a Kubernetes governance solution, such as Fairwinds Insights, or outsource your Kubernetes management to Fairwinds Managed Kubernetes-as-a-Service.

FINAL THOUGHTS ON KUBERNETES POLICY ENFORCEMENT BEST PRACTICES

Your golden path or IDP requires Kubernetes governance, especially when running multi-tenant or multi-cluster environments with many teams and users.

CONCLUSION

Kubernetes security, efficiency, reliability, and policy enforcement remain complex, even as the platform matures. As organizations continue to scale and diversify their workloads, embracing AI/ML, multi-cloud, and hybrid environments, the need for consistent, automated enforcement of best practices is more important than ever.

By adopting the foundational practices outlined in this whitepaper and leveraging modern tools for automation, policy management, and observability, organizations can maximize the value of Kubernetes while minimizing risk and inefficiency. As the Kubernetes ecosystem evolves, your approach must evolve as well. Regularly revisit and update your practices to ensure long-term success.

For teams seeking to accelerate their cloud-native journey, managed Kubernetes services, expert guidance, and open-source tools like Goldilocks and Polaris can provide the guardrails needed to scale confidently. Stay proactive, stay secure, and continue to iterate on your Kubernetes strategy to meet the demands of 2025 and beyond.

Benefits of Fairwinds

Fairwinds provides Managed Kubernetes-as-a-Service, powered by expert people and best-in-class software to make your Kubernetes platform fast, secure, and stable.

With years of experience creating, deploying, and managing production-grade clusters for companies of all sizes, Fairwinds empowers organizations to design and deploy cloud-native applications at scale through its people-led services enabled with software.

Fairwinds SREs provide expertise and deep Kubernetes knowledge, becoming part of your platform engineering team to run the core Kubernetes infrastructure, perform upgrades, and support your decision-making. Fairwinds maintains the core infrastructure so that platform engineering can focus on enabling the business-specific needs of developers.

Fairwinds

WHY FAIRWINDS

Fairwinds builds software for Kubernetes platform engineers to standardize and enable development best practices. With Fairwinds, platform teams decrease friction, increase dev velocity and improve the dev experience to accelerate time to market and revenue generation. Customers ship cloud native applications faster, more cost-effectively and with less risk.

WWW.FAIRWINDS.COM