KUBERNETES BEST PRACTICES FOR ENHANCED CLUSTER EFFICIENCY



With containerization quickly changing architectural patterns of application development, the Kubernetes platform continues to be its flag-bearer. Based on Forrester's 2020 Container Adoption Survey, roughly 65% of surveyed organizations are already using, or are planning to use, container orchestration tools as part of their IT transformation strategy. In all likelihood, the popularity of Kubernetes will only grow.

Although Kubernetes extends a future-proof container solution to improve productivity, use cases also indicate that relying solely on out-of-the-box Kubernetes services to containerize application builds may not always be the best approach. To get the most out of K8s, implement best practices and follow a custom-configured model to ensure the optimal platform your application build requires.

This article will delve into 11 best practices to realize a Kubernetes cluster model that is scalable, secured, and highly optimized.



(This article is part of our Kubernetes Guide. Use the right-hand menu to navigate.)

1. Download the latest version

With its regular version updates, Kubernetes releases new features, bug fixes, and platform upgrades. As a rule of thumb, you must always use the latest Kubernetes version on your cluster. This ensures that your version has:

- All the updated features, so you don't miss out on support from older, unsupported versions.
- Updated security patches that defer potential attack vectors while fixing reported vulnerabilities.

While updated features are good, updated security is crucial.

2. Use namespaces

Adopting Kubernetes in larger organizations with multiple teams accessing the same cluster requires a custom approach to resource usage. Improper accessibility provisioning often leads to

conflicts among teams for resource usage.

To solve this, use namespaces to achieve team-level isolation for teams trying to access the same cluster resources concurrently. Efficient use of Namespaces helps create multiple logical cluster partitions, thereby allocating distinct virtual resources among teams.

3. Maintain small container images

Most developers make the mistake of using the base image out-of-the-box, which may have up to 80% of packages and libraries they don't need.

Always use smaller container images as it helps you to create faster builds. As a best practice, you should:

- Go for Alpine Images, as they are 10x smaller than the base images
- Add necessary libraries and packages as required for your application.

Smaller images are also less susceptible to attack vectors due to a reduced attack surface.

4. Set resource requests & limits

To avoid cases where a single team or application drains all the cluster resources, set requests and limits for cluster resources, specifically CPU and Memory. Doing so limits disproportionate resource usage by applications and services, thereby avoiding capacity downtime.

To set requests and limits on a container, you may use the following container spec as a reference:

```
containers:
- name: darwin
    image: CentOS
    resources:
        requests:
            memory: “256Mi”
            cpu: “300m”
        limits:
            memory: “1024Mi”
            cpu: “900m”
```

The above container spec essentially allocates resource requests 256 MiB of memory and 300 mCPU, while limiting a maximum of 900 mCPU and 1024 MiB to the **darwin** container.

5. Use readinessProbe & livenessProbe

Leverage **Kubernetes Check Probes** to proactively avoid pod failures:

- With **readinessProbe** for production apps, Kubernetes checks if the application is ready to start serving traffic before allowing traffic to a pod. This essentially indicates whether a pod is available to accept traffic and respond to requests.
- Through a **livenessProbe**, Kubernetes performs a health check to ensure the application is responsive and running as intended. In the event the **livenessProbe** fails, the **kubelet** default policy restarts the container to bring it back-up.

6. Deploy RBAC for security

Role-based Access Controls (RBAC) help administer access policies to define who can do what on the Kubernetes cluster. To set RBAC permissions on Kubernetes resources, Kubernetes provides these parameters:

- **Role** for a namespaced resource
- **ClusterRole** for a non-namespaced resource

Here is an example where **ClusterRole** is used to administer **read access** to services on all namespaces:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
 name: reader
rules:
- apiGroups:
 resources:
 verbs:
```

Additionally, Kubernetes also allows **RoleBinding** and **ClusterRoleBinding** by referencing roles already administered on a user, group, or service account. More details on Kubernetes RBAC policies can be found here.

7. Autoscale

It is strongly recommended that you leverage benefits from Kubernetes' autoscaling mechanisms to automatically scale cluster services with a surge in resource consumption.

With **Horizontal Pod Autoscaler** and **Cluster Autoscaler**, node and pod volumes get adjusted dynamically in real-time, thereby maintaining load to the optimum level and avoiding capacity downtimes.

8. Monitor control plane components

Don't make this common mistake: forgetting to monitor the Kubernetes cluster's brain—the control plane. The control plane includes:

- the Kubernetes API Service
- **kubelet**
- **controller-manager**
- **etcd**
- **kube-proxy**
- **kube-dns**

Monitoring the control plane helps identify issues/threats within the cluster and increase its latency. It is also recommended to use automated monitoring tools (Dynatrace, Datadog, Sysdig, etc.) rather than manually managing the alerts.

Keeping an eye on your Kubernetes' control plane components essentially monitors workload and resource consumption to help mitigate issues with your cluster health.

9. Adopt Git-based workflow

Use GitOps, a Git-based workflow, as the preferred model to use Git as the single source of truth for all automation, including CI/CD pipelines. Adopting a GitOps framework helps improve productivity by:

- Bringing down deployment times
- Enhancing error traceability
- Automating CI/CD workflows

Leveraging GitOps on a Kubernetes cluster helps you achieve unified management of the cluster as well as sped-up application development.

10. Watch for high disk usage

High-disk usage is a common issue that impacts cluster performance. As a regular practice, you should monitor:

- The root file system
- All disk volumes associated with the cluster

Frequently, you would also encounter high-disk utilization alerts for unknown reasons; such cases usually tend to get tricky to fix due to their obscure nature of the root cause. Keeping alert monitoring in place helps take corrective actions either by scaling or freeing disk space at the right time.

11. Regularly audit policy logs

Underrated as a best practice, all logs stored at **/var/log/audit.log** must be audited regularly to:

- Identify threats
- Monitor resource consumption
- Capture key event heartbeats of the Kubernetes cluster

The default policies of the Kubernetes cluster are defined in the **/etc/kubernetes/audit-policy.yaml** file and can be customized for specific requirements. Additionally, you could also use Fluentd as an open-source tool to maintain a centralized logging layer for your containers.

Custom approach for K8s

If you are already using Kubernetes or are getting production-ready, a custom approach to configuring your cluster goes a long way. These best practices have been repeatedly suggested by professionals who have already gone through the pain of learning the hard way.

You may also often realize a few practices that may suit your cluster architecture, as every application build would require a completely different approach to fine-tune efficiency. The key to success would rely, no matter what, on how optimized your containerized framework is.

Additional resources

For related reading, explore these articles:

- BMC DevOps Blog
- Kubernetes Guide, a series of articles and tutorials
- 3 Kubernetes Patterns for Cloud Native Applications
- State of Containers in 2020
- Containerized Machine Learning: An Intro to ML in Containers