



THE JFROG JOURNEY TO KUBERNETES:

BEST PRACTICES FOR TAKING YOUR
CONTAINERS ALL THE WAY
TO PRODUCTION

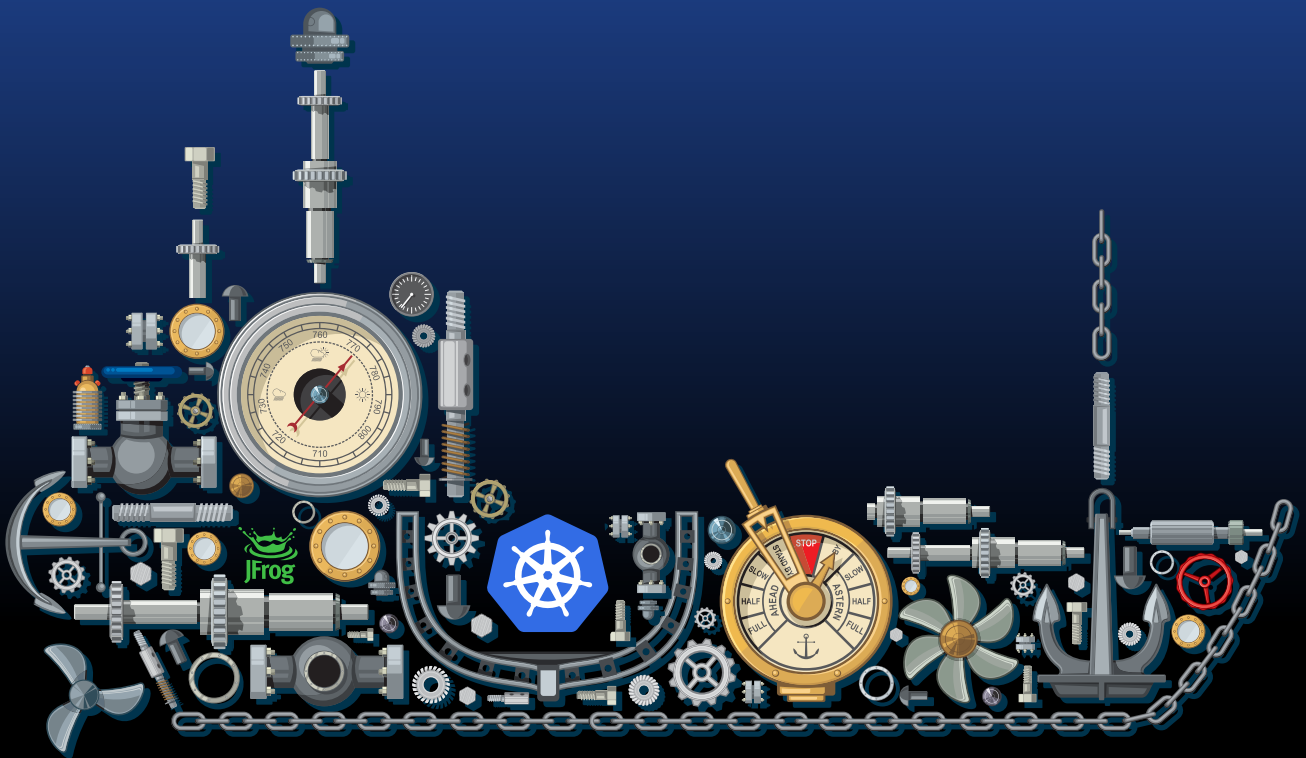


TABLE OF CONTENT

Table of Content	1
Executive Summary	2
Introduction	3
Kubernetes Cluster	3
CI/CD Pipeline	3
Kubernetes Registry	3
1. Getting Your Application Ready for Kubernetes	4
Questionnaire Checklist: Preparing Your App for K8S	4
2. Gaining Flexibility and Universality in Kubernetes	5
Deploying Artifactory as Your Kubernetes Registry	5
3. Automating Deployment to Kubernetes	6
Deploying Artifactory as Your Helm Chart Repository	6
Best Practices for Deploying Apps in your CI/CD Pipeline	6
4. Building Reliable and Scalable Environments in Kubernetes	7
Deploying Artifactory HA in Kubernetes	7
Storage and Scalability of Your Kubernetes Cluster	7
5. Visibility and Security: Protecting Your Apps in Kubernetes	8
Gaining Visibility into Your Containers in K8S	8
Scanning and Detecting Vulnerabilities in Containers	8
Protecting Your Open Source Projects in K8S	8
Preventing Unauthorized Access Using Tillerless in Helm 2	9
Setting Kubernetes with RBAC	9
6. Logging, Monitoring and Debugging Your Apps in K8S	10
Best Practices for Logging Your Apps in Kubernetes	10
Continuous Monitoring of Your Microservices in K8S	10
7. Deploying Your App to Production in K8S	11
10 Tips for Smoothly Embarking on Your Journey to Production	11
Conclusion	12

EXECUTIVE SUMMARY

“When we moved to microservices in Kubernetes, we went from 12 releases a year to doing 2,200 releases with a much lower failure rate.”

Sarah Wells, Tech Dir. for Operations and Reliability, Financial Times.

Kubernetes has become the de facto leading orchestration tool in the market and not only for technology companies but for all companies as it allows you to quickly and predictably deploy your applications, scale them on the fly, seamlessly roll out new features while efficiently utilizing your hardware resources.

The JFrog journey with Kubernetes started when we were seeking a suitable container orchestration solution to spin up a fully functional environment for internal purposes. Our developers needed to test our very complex environments including JFrog Artifactory and other products. In parallel to that, we needed to provide the program and product managers with a fully functional environment for demoing the **JFrog Enterprise+ platform** to our customers.

To meet our needs, each product required an independent CI/CD development environment allowing for testing the individual branches in isolation from others while testing the interaction between the branches.

As we gained confidence in Kubernetes, we acknowledged the value of distributing the JFrog products to Kubernetes while having the ability to run the applications across different staging, development and production environments. Kubernetes also allowed us to better utilize our resources as we no longer were required to spin up a single VM for deploying each product separately. In this white paper, we share our best practices, tips and lessons learned when taking containerized applications all the way to production with Kubernetes.

INTRODUCTION

Kubernetes lets you create containerized apps and deploy them side-by-side without being concerned about compatibility between the various services and components. The benefit of containerizing an app and running it in Kubernetes is that you get to develop your product within a vibrant community which makes it easier to create [scalable microservices apps](#). The downside to that is that when you have an entire team working on the various components, it becomes rather complex rather quickly. Adding to all that is the fact that your containerized apps can contain multiple component types depending on the operating system, language, and framework(s) you are using.

Let's begin with the three essential components required for running your application with Kubernetes:

KUBERNETES CLUSTER

The Kubernetes cluster is the orchestration infrastructure where your containerized application runs. You need to decide if you want to manage it yourself or not and if you want to host it using a cloud provider. This component is not discussed in this scope of the document.

CI/CD PIPELINE

The [CI/CD pipeline runs in Kubernetes and automates the process](#), starting from the source code and external packages to deploying your application in a Kubernetes cluster. Kubernetes pipelines are 'application aware', meaning they are natively capable of dynamically provisioning a full containerized application stack (generally composed of multiple services, deployments, replica sets, secrets, configmap, etc.). Every change to the application context, whether code, base-layer, image, or configuration changes, will in turn trigger a pipeline.

KUBERNETES REGISTRY







Your production clusters should use a single, managed and trusted source of truth that stores and tracks all pieces making up your applications and dependencies. [Using the Kubernetes registry](#), you can run multiple application stacks side-by-side in a pod without conflict and without caring about the internal dependencies of each app. This separates the concerns between maintaining a running cluster, scaling applications up and down, developing new versions, and debugging application specific issues.

1. GETTING YOUR APPLICATION READY FOR KUBERNETES

Your application is the heart of your service/solution. You need to plan and prepare your application before you can run it in Kubernetes.

QUESTIONNAIRE CHECKLIST: PREPARING YOUR APP FOR K8S

The following table shows the application-related tasks and questions you must ask before you prepare your application for Kubernetes.

TASKS	QUESTIONS	
Logging	<ul style="list-style-type: none">■ How is your application logging set up?■ Where will the logs be saved?■ Do you need logs files or perhaps using STDOUT/STDERR is sufficient?■ How will you handle multiple log files?	 <div>K8S GURU TIP Consider turning your logs to softlinks by setting <code>/dev/stdout</code> or <code>/dev/stderr</code> thereby ensuring all your logs are part of the container log.</div>
Data Persistency	<ul style="list-style-type: none">■ Is your application stateful?■ Does it require data persistence?■ What part of your data needs persistency?	 <div>K8S GURU TIP Don't store all your data on a persistent storage. Store only persistent data.</div>
Termination Signals	<ul style="list-style-type: none">■ How do you handle termination signals?	 <div>K8S GURU TIP Use <code>trap</code> in your container bash entrypoint to catch termination signals and handle them properly.</div>
Application Restart	<ul style="list-style-type: none">■ How will you survive a restart?■ What happens if you kill the pod?■ What happens if you crash the process in the pod?■ What happens if the K8S node crashes?■ How does the application behave?	 <div>K8S GURU TIP A great way to test your application recovery is to kill the pods or, kill the nodes, and see what happens?</div>
High Availability	<ul style="list-style-type: none">■ How should I set up my nodes and load balancer to achieve zero service unavailability of my application/service?	 <div>K8S GURU TIP Plan for zero service unavailability allowing for pod scheduling when performing cluster scaling (down) and planned node maintenance.</div>
Probes	<ul style="list-style-type: none">■ Do your applications have endpoints that can be used to check health and readiness using the Liveness and Readiness Probes?	 <div>K8S GURU TIP Proper use of probes can help you implement a great "auto-healing" process for your applications and will save your engineers many sleepless nights.</div>

For more information on the basics of building your software as a service, see [The Twelve-Factor App](#).

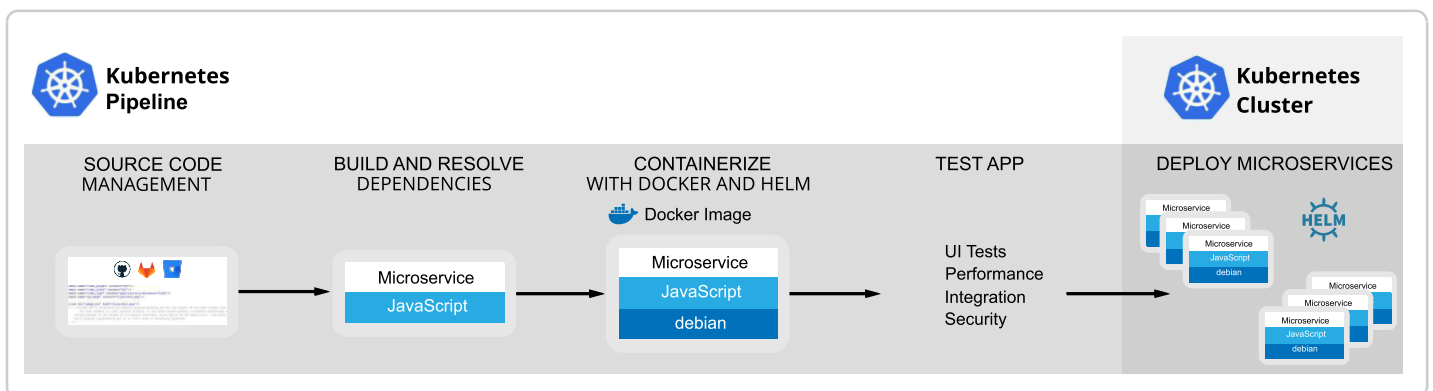
2. GAINING FLEXIBILITY AND UNIVERSALITY IN KUBERNETES

Polyglot programming and multiple disparate tools and technologies provide a multitude of possibilities. You can pick and choose the ones that best meet your business needs, but each technology may have a different interface, REST API and its own package format. The only way to support these tools is to be versatile by going universal in how you manage your artifact lifecycle from creation to deployment.

DEPLOYING ARTIFACTORY AS YOUR KUBERNETES REGISTRY

You can gain flexibility and universality by using Artifactory as your “Kubernetes Registry” as it lets you gain insight into your code-to-cluster process while relating to each layer for each application and act as your single source of trusted truth. [Artifactory supports 25+ different technologies](#) in one system with one metadata model, one promotion flow, and strong inter-artifact relationships.

Artifactory allows you to deploy containerized microservices to the Kubernetes cluster as it serves as a [universal repository manager](#) for all your CI/CD needs, regardless of where they are running in your organization. Once you check in your App package, you can proceed to propagate and perform the build, test, promote and finally deploy to Kubernetes. To easily deploy Artifactory (and other JFrog products) to Kubernetes, refer to our official JFrog helm charts in the [Helm hub](#).



3. BUILDING AUTOMATING DEPLOYMENT TO KUBERNETES

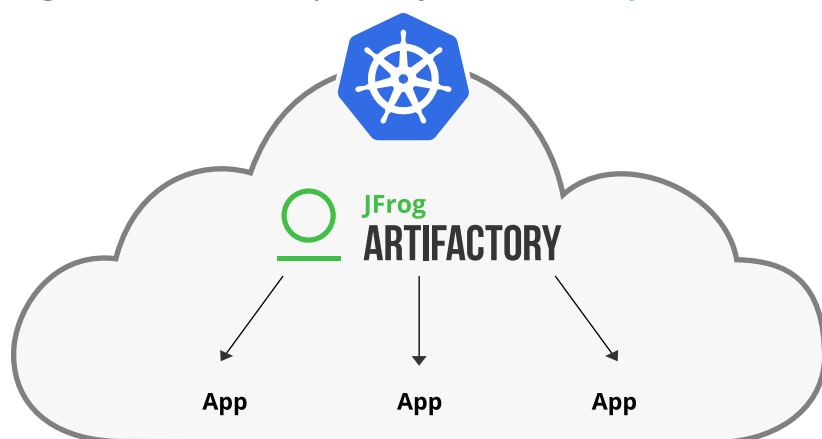
Facilitating the deployment of applications reliably at scale without the need for human intervention at every stage of the CI/CD pipeline is the main reason for orchestration.

But how do you get your code into the cluster in a repeatable, reliable way? And how do you make sure that only the right version of your app makes it to production?

To accomplish this, we propose deploying Artifactory as your repository manager, to play a vital role in your CI/CD pipeline by bridging the gap between development and operations.

DEPLOYING ARTIFACTORY AS YOUR HELM CHART REPOSITORY

Artifactory natively supports [Helm repositories](#), giving you full control of your deployment process to Kubernetes. It provides secure, private, local Helm repositories to share Helm charts across your organization with fine-grained access control. Proxies and caches public Helm resources with remote repositories, and aggregates local and remote resources under a single virtual Helm repository to [access all your Helm charts from a single URL](#).



K8S GURU TIP

When using Artifactory as your Helm repository, we recommend:

- Separating your Stable and Incubator repositories.
- Using SemVer version 2 versions in your charts.
- Periodically recalculating the index.yaml file from scratch in Artifactory.

BEST PRACTICES FOR DEPLOYING APPS IN YOUR CI/CD PIPELINE

When deploying your applications in your CI/CD pipeline, we recommend:

- Using the same Helm chart for local, Staging, Testing and Production while using a separate values.yaml file for each environment. Each yaml needs to contain specific environment configuration values. For example: values-stg.yaml, value-prod.yaml.
- Managing the custom values in your VCS.
- Settings in the default values.yaml should be for dev or local, so the developer can use it locally without hassle.
- Using external charts for dependencies. Use the work already done by the community!
- For security purposes: Separate your secrets from your charts and reference them as external charts.

4. BUILDING RELIABLE AND SCALABLE ENVIRONMENTS IN KUBERNETES

Running multiple applications side-by-side in your Kubernetes cluster requires establishing continuous access to your artifacts, while supporting heavy load bursts with zero downtime.

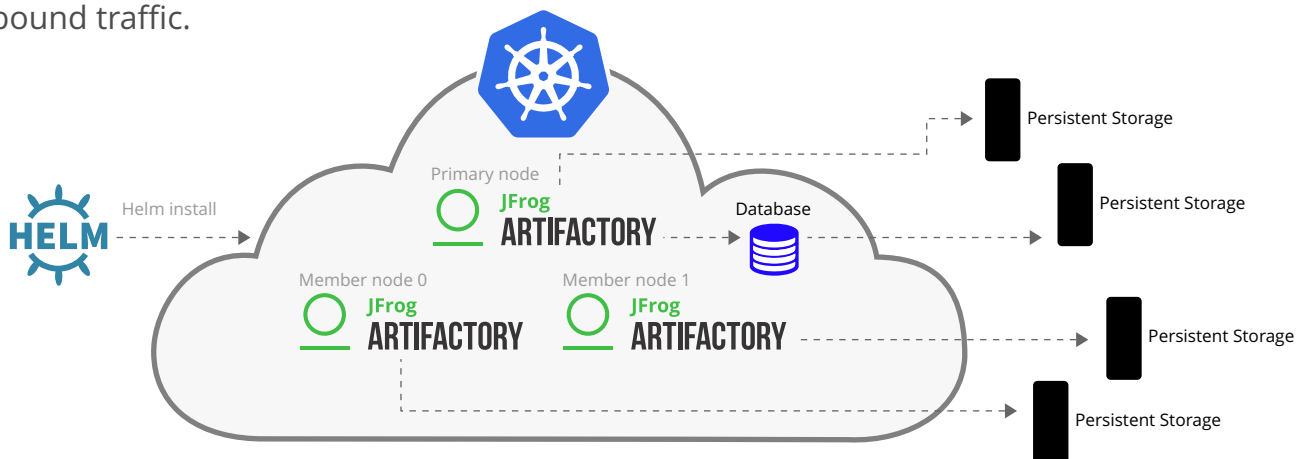
DEPLOYING ARTIFACTORY HA IN KUBERNETES

By deploying Artifactory HA in the Kubernetes cluster, you will experience zero service unavailability due to any type of pod scheduling when it comes to cluster scaling, if a pod is evicted or crashes, or in case of an unplanned outage of a node.

The benefits of deploying Artifactory HA in Kubernetes are:

- Accommodates larger load bursts with no compromise to performance.
- Provides horizontal server scalability, allowing you to easily increase your capacity to meet any load requirements as your organization grows.
- Supports performing most maintenance tasks with no system downtime.
- Supports rolling upgrades as newer versions can be installed on instances by replacing individual instances of your application with zero downtime.

In the following example, an Artifactory HA cluster is deployed using three nodes: a primary node and two member nodes. As load balancing is performed on the member nodes only. This leaves the primary node free to handle jobs and tasks and not be interrupted by inbound traffic.



You can use this predefined [Artifactory High Availability Helm chart](#) to create your own Artifactory HA environment.

STORAGE AND SCALABILITY OF YOUR KUBERNETES CLUSTER

Artifactory HA allows you to push the limits of your applications in Kubernetes as it supports a wide array of storage alternatives. For more information, see [Configuring the Filestore](#).

5. VISIBILITY AND SECURITY: PROTECTING YOUR APPS IN KUBERNETES

Cloud-native technologies like [Docker and Kubernetes](#) present a larger attack surface, with more potential entry points for malicious crypto mining, ransomware, and data theft.

Services running in your Kubernetes cluster are not totally isolated and may have access to other areas in your cluster.

Just for that reason, visibility into your cluster is crucial especially from a security perspective. You need to know what's running in your containers as your application rarely contains a single component but rather includes external dependencies such as OS packages, OSS libs, and 3rd party processes. This leads to the inevitable questions - Are they safe? Do they contain [security vulnerabilities](#)? Do they abide to FOSS license compliance?

GAINING VISIBILITY INTO YOUR CONTAINERS IN K8S

Artifactory gives you insight into the CI/CD process by providing auditability as it captures a substantial amount of valuable metadata that's emitted throughout the CI/CD process. You can trace the CI job responsible for producing the application tier that is part of the Docker image layer. It can also show build differences by allowing you to compare two builds, making it easy to trace which layer of your Docker image was generated to which build so you can track it down to the commit.

K8S GURU TIP

*Recommended Reading:
[9 Kubernetes Security Best Practices Everyone Must Follow](#)*



SCANNING AND DETECTING VULNERABILITIES IN CONTAINERS

JFrog Xray works with Artifactory to perform universal analysis of binary software artifacts at any stage of the application lifecycle. It runs a recursive scan of all of the layers in your container and helps identify vulnerabilities in all layers by [scanning and analyzing artifacts](#) and their metadata, recursively going through dependencies at any level.

[Policies can be set in Xray](#) to limit or prevent deployment of container images to Kubernetes according to the level of risk indicated by what Xray's scans find. In this way, vulnerable or non-compliant applications can be kept from running, or limited in what they are allowed to do when they are launched.

PROTECTING YOUR OPEN SOURCE PROJECTS IN K8S

Most applications rely heavily on dependencies from package managers and [open source repositories](#), and are therefore vulnerable to either malicious or insecure code from these sources. As part of [our initiative to support and contribute to the Open Source community](#), JFrog has developed KubeXray, an open source project that extends the security of Xray to the applications running (or about to run) in your Kubernetes pods. Using the metadata that Xray generates by scanning container images, KubeXray can enforce your policies on what has already been deployed.





KubeXray monitors all of your active Kubernetes pods to help you:

- Catch newly reported risks or vulnerabilities in applications that are currently running in all Kubernetes pods.
- Enforce your current policy on running applications, even after you have changed those policies.
- Enforce policy for running applications that have not been scanned by Xray, and whose risks are unknown.

PREVENTING UNAUTHORIZED ACCESS USING TILLERLESS IN HELM 2

Helm 2 includes a server-side component called “Tiller”. Tiller is an in-cluster server that interacts with the Helm client, and interfaces with the Kubernetes API server.

Tiller is definitely cool but it is important to be aware that there are security issues related to Tiller in Helm 2. This is because the Helm client is responsible for managing charts, and the server is responsible for managing the release. This poses a great risk as Tiller runs with root access and someone can get unauthorized access to your server.

Rimas Mocevicius, a Kubernaut at JFrog and co-founder of Helm proposes an innovative approach to addressing this situation by running Helm and Tiller on your workstation or in CI/CD pipelines without installing Tiller to your Kubernetes cluster. To get you up and running you can download and install the [Tillerless Helm v2 plugin](#).

SETTING KUBERNETES WITH RBAC

Setting RBAC (Role-based Access Control) as an administrative function for Kubernetes is a must as it allows you to define which user can administer the cluster and its granularity. In addition to defining which users and applications can be listed, while getting, creating or deleting pods and other Kubernetes objects. If you do not specify a service account, it automatically assigns it to the pod as the “default” service account in the same namespace.

We recommend not using the default which comes with the namespace. Always create a service account for your application as it will allow you to set your application limiting including namespace or cluster-wide actions, and totally disabling access to Kubernetes API.

K8S GURU TIP



A good practice is to disable the access to API by setting “`automountServiceAccountToken: false`” in the created service account for the application.

K8S GURU TIP



Adhere to the Least Privileges Principle by granting the least permissions to users when accessing your applications in Kubernetes!

6. LOGGING, MONITORING AND DEBUGGING YOUR APPS IN K8S

The number of microservices is growing together with increasing complexity and the question is how do you track and monitor them and what should you be monitoring. When it comes to microservices, you need to collect data for:

- Unexpected events: For example, a change of ownership executed in a database container.
- Typos causing a microservice to go down.
- Incorrect files selected in production causing chaos.
- A specific base OS version was not allowed.

BEST PRACTICES FOR LOGGING YOUR APPS IN KUBERNETES

Application and system logging is essential for troubleshooting your Kubernetes cluster activity.

Follow these best practices when logging your applications in Kubernetes:

- Limit direct access to the logs.
- When using the Kubernetes Dashboard (not recommended for production), set the dashboard as read-only with access rights. You can allow other members to perform troubleshooting, but refrain from providing full access to the dashboard as it can cause damage to your Kubernetes cluster.
- Make sure your logs are accessible in real-time and available for analysis at a later stage.
- Use a log collecting tool, such as the ELK/EFK stack (ElasticSearch, Logstash/Fluentd and Kibana), to collect and index all logs from your system and applications.
- Consider saving your logs in a separate cluster to consume the logs at a later stage. This is especially useful if your cluster goes down allowing you to gain access to the logs.

CONTINUOUS MONITORING OF YOUR MICROSERVICES IN K8S

The need for continuous monitoring of your system and application health is critical!

There are many free and commercial solutions for real-time monitoring of your Kubernetes cluster and the applications running in it. One of the popular solutions is the combination of [Prometheus](#) and [Grafana](#), which provide real-time monitoring which can be combined with alerting tools.

7. DEPLOYING YOUR APP TO PRODUCTION IN K8S

Based on our journey, we recommend you read these 10 tips before embarking on your journey to Kubernetes.

10 TIPS FOR SMOOTHLY EMBARKING ON YOUR JOURNEY TO PRODUCTION

1. For beginners, we recommend starting by reading [Kubernetes the hard way!](#)
2. Start small. Learn from examples and start with a small application (nginx), use existing demos, and try deploy your apps in Skin Kubernetes for Docker.
3. Get your application ready before jumping into K8S.
4. [Set a minimal goal for getting your application to run in K8S.](#)
5. Use managed K8S to free your work for example: AKS, ESK or GKE, which abstract lots of the complication for you.
6. Have one main container per POD.
7. We recommend trying the Managed GKE when selecting managed Kubernetes.
8. Determine where to store your database in or outside the Kubernetes cluster.
This is critical, as you need to plan for cluster recovery in case of cluster crashes.
Consider the following:
 - ☑ When K8S running On-Prem: Use your existing database on-prem as a stateless app in Kubernetes.
 - ☑ When running K8S on the cloud: Select a persistent database like PostgreSQL or MySQL operator that knows how to recover when a Kubernetes node goes down.
9. When deploying to cloud, separate your clusters for running your CI/CD pipeline.
Deploy from external CI/CD pipelines to the Kubernetes clusters.
10. Work with the community!

CONCLUSION

As described in this whitepaper, we showed how Kubernetes together with JFrog Artifactory allows you to reliably and predictably deploy your applications, scale them on the fly, seamlessly roll out new features and efficiently utilize hardware resources.

This hands-on-guide was intended to review the complexity and the challenges facing companies wanting to adopt Kubernetes as their container orchestration tool. We hope that the lessons learned, best practices, and tips we shared will help get you up and running on your voyage to Kubernetes.