BEST PRACTICES: HOW TO ARCHITECT APPLICATIONS FOR KUBERNETES



Architecting a production-ready application for Kubernetes is not an easy task, but guidelines and best practices are available to ensure your application meets all necessary requirements for a production environment.

The microservice pattern has helped evolve the way applications are built. Because of this change, the platforms that can run your app should be considered during development. Indeed, Kubernetes itself—an exceedingly popular container orchestration platform—has drastically changed how we build and deploy apps. So, here are a few tips to keep in mind before running your application on Kubernetes.

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

Your app must be a container image

Because K8s is a container orchestration engine, your application must be packaged into a container image for it to run in a k8s cluster. This is among the most basic requirements for building microservices, as it directly ties into how Kubernetes functions as a platform.

Ideally, you should keep your application image in a private repository for secure storage. For example, you can build a docker image and push that image to a private docker repository, where Kubernetes can then pull your image from.

By default, Kubernetes supports container runtimes in:

- Docker
- CRI-O
- containerd
- Frakti

Follow best practice for whichever runtime you use. The one for Docker can be found here.

Helm is your friend

If you have never used Helm charts, you are missing out. Helm is the de-facto way to deploy production-grade applications into Kubernetes. It allows you to describe your application in charts—so you can version, publish, share, and do other useful things with your application. It is Kubernetes yum or apt-get for simplicity.

A typical helm chart looks something like this tree:

```
YOUR-CHART-NAME/
 |
 |- .helmignore
 |
 |- Chart.yaml
 |
 |- values.yaml
 |
 |- charts/
 |
 |- templates/
```

Note: Using a CLI tool like kubectl should be used *only* in dev or for debugging—**never** for deploying into production.

Use liveness and readiness probes

An unhealthy application can result in revenue lose. Luckily, Kubernetes offers two key concepts to check the health (or failure) of your applications:

- Liveness probes. You are telling the kubelet to kill the container your application is running on if the probe fails.
- Readiness probes. You are telling the kubelet to make sure the service load balancer that fronts the application does not route traffic to this application if the probe fails.

Using liveness and readiness probes in your manifest is a great way to control and know your application health before and after it starts up in a cluster. Once your app passes the probe, the service will send traffic to the application.

12-factor, if you haven't

Deploying your application into a Kubernetes cluster is great. But, if your application is not built to be cloud-native, chances are that it won't play well with Kubernetes. A good rule of thumb is to follow

the 12-factor application methodology.

Some benefits of following the 12 factors:

- Scaling up is easy, with limited changes to tooling or architecture.
- Dev, stage, and prod are as close as possible because there are no config values hardcoded in the config files.
- Makes sure the service has a clean contract with OS. (Meaning it does not care which OS it is running on.)
- Services deploy easily into any public or private cloud.

Operator uses your expert knowledge

A main reason that Kubernetes is so powerful? The fact that you can extend it. In other words, you can apply your application domain knowledge to create your own custom controller and custom resources in Kubernetes that knows how your application should function and performs certain tasks when needed. That's exactly what the Kubernetes operator pattern is for.

Part of architecting your application for k8s is to perform enough planning to know what kind of automation your application needs—and whether you can leverage k8s operator for those automation tasks. The operator is especially useful when you have a stateful application that you need to perform some type of occasional admin task, like taking and restoring backups of that application's state. In this scenario, you'd have to create a custom controller and custom resource to perform this task because Kubernetes does not know how to perform this task out-of-the-box.

Architecting your application for Kubernetes is not easy, but by following these best practices, you can have a well-architectured application running in k8s cluster. Start with these application tips, and look to continually improve the architecture process.

Additional resources

For more on Kubernetes, check out these resources:

- Kubernetes Guide, offering 20+ articles and tutorials
- BMC DevOps Blog
- Bring Kubernetes to the Serverless Party