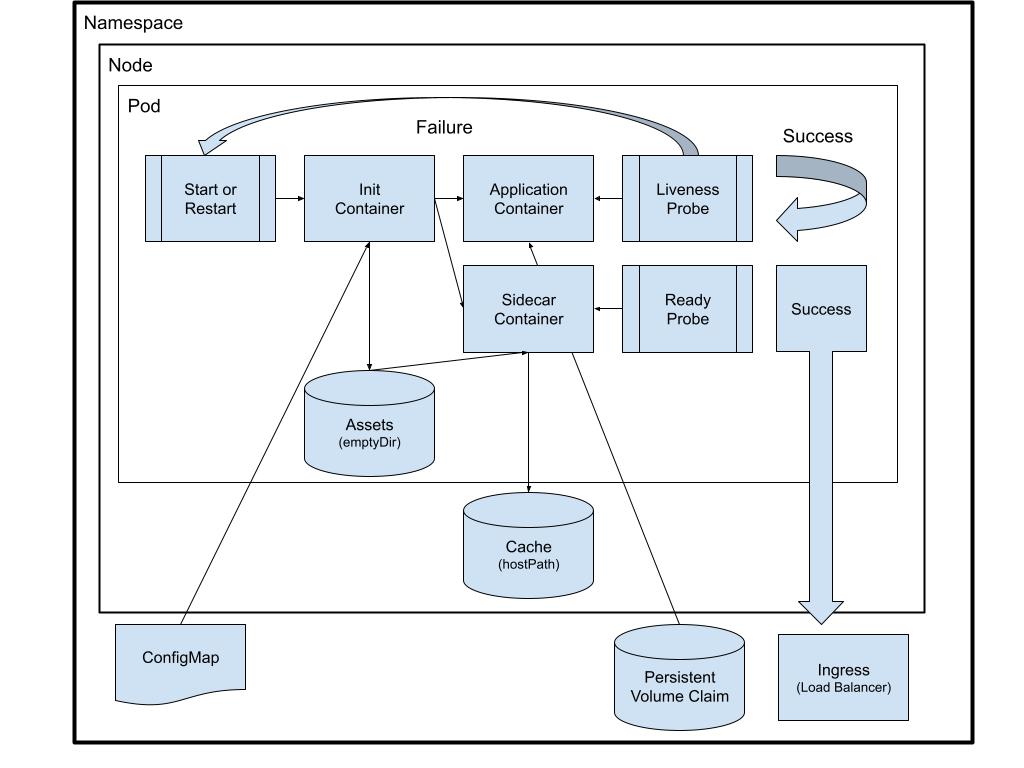
**Kubernetes Pods Advanced Concepts Explained**

In this blog post, we’ll investigate certain advanced concepts related to Kubernetes init containers, sidecars, config maps, and probes. We’ll show you how to implement these concepts in your own cluster, but more importantly how to apply these to your projects in [Release](https://releasehub.com/?__hstc=168857022.09f300a47e8621f16e70136af0222924.1696331955898.1696331955898.1696331955898.1&__hssc=168857022.1.1696331955898&__hsfp=4083444424) for both fun and profit.

We’ll start with a brief introduction to pods and containers in Kubernetes, and then show specific examples of each item listed above. Below you will find a drawing of these examples to keep yourself oriented during our bumpy ride ahead.



Key Kubernetes Pod Concepts

Before we begin, let’s get a brief overview of some key concepts.

Container

In Docker, a container is an image that bundles layered filesystems which can be deployed as a runnable bundle. This container is usually built with a Dockerfile and has a startup binary or executable command.

Sidecar Container

A [sidecar container](https://kubernetes.io/docs/concepts/workloads/pods/#how-pods-manage-multiple-containers) is simply a container that runs alongside other containers in the pod. There’s no official definition of a sidecar concept. The only thing that distinguishes a container as a sidecar container is that you consider it ancillary or secondary to the primary container. Running multiple sidecar containers does not scale well, but does have additional advantages of being able to reuse configuration files and container images. The reason sidecars do not scale well is that they may be overprisioned or wasteful based on the performance of the main application container. However, the tradeoffs can make sense in legacy applications or during migrations toward truly cloud-native designs.

Init Container

An [init container](https://kubernetes.io/docs/concepts/workloads/pods/init-containers/) is simply a container that runs before any other containers in the pod. You can have several init containers that run sequentially. As each container finishes and exits properly (with a zero!), the next container will start. If an init container exits with an error or if it does not finish completely, the pod could go into a [dreaded CrashLoopBackoff](https://releasehub.com/blog/kubernetes-how-to-debug-crashloopbackoff-in-a-container?__hstc=168857022.09f300a47e8621f16e70136af0222924.1696331955898.1696331955898.1696331955898.1&__hssc=168857022.1.1696331955898&__hsfp=4083444424). All of the containers share a filesystem, so the benefit here is that you can use or reuse container images to process, compile, or generate files or documents that can be picked up later by other containers.

Probes

Although the word “probes” may stir up visions of Alien tools used for discovery and investigation of humans, fear not. These probes will only make your services run better! Kubernetes has [several probes](https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/) for defining the health of containers inside a pod. A startup probe allows the scheduler to tolerate delays in a slow-startup container. A liveness probe allows Kubernetes to restart a faulty or stalled container. A readiness probe allows a container to receive traffic only when it is ready to do so.

Pod

You may harbour some fear in the back of your mind of “pod people” or vegetable clones grown to replace humanity with mindless zombies who hunt and destroy mankind. However, in Kubernetes, the smallest managed unit is the pod. But a pod could be composed of several containers that run in a single process space and filesystem. A pod is usually composed of one container that runs a single process as a service. However, there are several advanced usage examples we will go into that run multiple containers for expanded options and use cases.

Node

A Kubernetes node is ultimately a physical machine (which can have several layers of virtualisation) that runs the pod or pods, providing the critical CPU, memory, disk, and network resources. Multiple pods can be spread across multiple nodes, but a single pod is contained on a single node.

Volumes

Volumes are simply abstractions of filesystems that can be mounted inside containers. You cannot overlap or nest volume mounts. However, there are several mount types that might be very useful to your use case.

configMap

A [configMap](https://kubernetes.io/docs/concepts/storage/volumes/#configmap) is a so-called “blob” of information that can be mounted as a file inside your container. Remember, that this is not an evil, destructive blob out to devour our planet! It is a batch of text that is treated amorphously, like a… well… blob. The usual use case here is for a configuration file or secrets mount.

emptyDir

An [emptyDir](https://kubernetes.io/docs/concepts/storage/volumes/#emptydir) is an empty filesystem that can be written into and used by containers inside a pod. The usual use case here is for temporary storage or initialization files that can be shared.

hostPath

A [hostPath](https://kubernetes.io/docs/concepts/storage/volumes/#hostpath) is a filesystem that exists on the Kubernetes node directly and can be shared between containers in the pod. The usual use case here is to store cached files that could be primed from previous deployments if they are available.

Persistent Volume Claim (PVC)

A [persistent volume claim](https://kubernetes.io/docs/concepts/storage/persistent-volumes/) is a filesystem that lasts across nodes and pods inside a namespace. Data in a PVC are not erased or destroyed when a pod is removed, only when the namespace is removed. PVCs come in many underlying flavors of storage, depending on your cloud provider and infrastructure architecture.

Namespace

A Kubernetes namespace is a collection of resources that are grouped together and generally have access to one another. Multiple pods, deployments, and volume claims (to list a few) will run together, potentially across multiple nodes.

Sidecars and Init Containers

The first use case we will cover involves running several containers inside a single pod. Once again, a pod here refers to one or more containers grouped together in Kubernetes, not vegetable human clones grown for evil reasons. In the following scenario, we will examine how multiple containers can share a single process space, filesystem, and network stack.

Keep in mind that most docker and Kubernetes purists will tell you that running more than one process in a container, or having more than one container in a pod is not a good design and will inevitably lead to scalability and architectural issues down the road. These concerns are generally well founded. However, careful application of the following supported and recommended patterns will allow you to thrive either during your transition from a legacy stack to Kubernetes or once you are successfully running your application in a cluster.

One particular use case we encounter with customers is that their application has a backend container that requires a reverse proxy like Nginx to perform routing, static file serving, and so forth. The best method to achieve this objective would be to create a separate pod with Nginx (for example) and run the two service pods in a single namespace. This gives us the flexibility to scale the backend pods and Nginx pods separately as needed. However, typically the backend service or application needs to also serve static files that are located inside the container filesystem and would not be available across the pod boundary. We agree this is not a preferred pattern to use, but it is common enough with legacy applications that we see it happen.

In this scenario, we often recommend a sidecar container running Nginx which can be pulled directly from Docker Hub or a custom image can be created. We also recommend that customers reuse their backend application container as an init container that starts with a custom command for creating any initialization or other startup tasks that need to be completed before the application itself starts.

One feature of this multi-container setup is that the Nginx container can use the “localhost” loopback to communicate with the backend service. Of course the sidecar container might be a logging or monitoring agent, but the principle is the same: the containers can speak with each other over a private network that is potentially not available outside of the pod, unless you make it available. In our Nginx example, the backend could be isolated so that all communication traffic inbound to the service container must be routed to the Nginx proxy.

The other nice feature of this configuration is that the containers all share a common file system so that the Nginx container can access static files generated by (or stored on) the backend service container.

Here is a link to our documentation that shows an example of running [sidecar](https://docs.releasehub.com/reference-guide/application-settings/application-template?__hstc=168857022.09f300a47e8621f16e70136af0222924.1696331955898.1696331955898.1696331955898.1&__hssc=168857022.1.1696331955898&__hsfp=4083444424#sidecar-containers) and [init](https://docs.releasehub.com/reference-guide/application-settings/application-template?__hstc=168857022.09f300a47e8621f16e70136af0222924.1696331955898.1696331955898.1696331955898.1&__hssc=168857022.1.1696331955898&__hsfp=4083444424#init-containers) containers on Release.

Probes

As we have noted, probes are not just for Aliens! Kubernetes uses them to test your application stack and report on its health. Kubernetes will also take action based on these probes, just like an Alien might. There are several probes that are supported natively by Kubernetes. The main use cases we support for our customers are the liveness probe and readiness probe.

The liveness probe is a way to test whether a container is “alive” or not, and if it fails the probe, then Kubernetes will restart the container. We usually recommend that your application not freeze up or have memory leaks and so forth so that a liveness probe should not be necessary. This “reboot your app to fix the problems” philosophy is not generally considered good practice. However, perfect code is impossible and when services are running in a production container environment, we know that almost anything can (and will) happen.

The readiness probe is a way to test whether a container is capable of serving traffic or not, and if it fails the probe, then the service port will be removed from the ingress controller. Contrary to our stance on the liveness probe, we strongly encourage and recommend that customers implement a readiness probe on any service that receives inbound traffic. In some sense, we consider a readiness probe mandatory for your production services.

Here is a link to our documentation that shows an example of using a [liveness and readiness probe](https://docs.releasehub.com/reference-guide/application-settings/application-template?__hstc=168857022.09f300a47e8621f16e70136af0222924.1696331955898.1696331955898.1696331955898.1&__hssc=168857022.1.1696331955898&__hsfp=4083444424#readiness-and-liveness-probes) for services running in Release.

Volumes

This section gets a bit technical and tricky. Of course, no actual customer stacks would use every single type of volume, container, and probe listed in this article. But we do hope this overview shows all the features that are possible. You should carefully consider the use cases presented below and choose the one that best fits your use case.

Here is a link to our documentation that shows options for our [storage volume types](https://docs.releasehub.com/reference-guide/application-settings/application-template?__hstc=168857022.09f300a47e8621f16e70136af0222924.1696331955898.1696331955898.1696331955898.1&__hssc=168857022.1.1696331955898&__hsfp=4083444424#resources).

configMap (Just in Time File Mounts)

A configMap (purposely spelled in [camelCase](https://en.wikipedia.org/wiki/Camel_case#Programming_and_coding)) is not itself a volume in Kubernetes. Strictly speaking, a configMap is just a blob of text that can be stored in the [etcd key-value datastore](https://kubernetes.io/docs/concepts/overview/components/#etcd). However, one convenient use case Release supports is creating a container storage volume that is mounted inside a container as a file whose contents are the text blob stored in etcd. At Release, we call this customer helper function a [Just in Time File Mount](https://docs.releasehub.com/reference-guide/application-settings/file-mounts?__hstc=168857022.09f300a47e8621f16e70136af0222924.1696331955898.1696331955898.1696331955898.1&__hssc=168857022.1.1696331955898&__hsfp=4083444424). The common use case for a configMap at Release is being able to upload a file with configuration details. For example, in our previous example involving an Nginx sidecar, the [nginx.conf](https://www.nginx.com/resources/wiki/start/topics/examples/full/) file could be uploaded as a Just in Time File Mount. *“What do we want? File Mounts! When do we want them? Just in Time!”*

emptyDir (Scratch Volume)

An emptyDir volume is a native Kubernetes construct Release supports for containers in a pod to share empty space that can be mounted locally. This volume is erased as soon as the pod ends its life-cycle, and it is blank to begin with. Thus, the most common use case is for a scratch or temporary location to store files that only need to be stored during the lifetime of the pod.

hostPath (Intra-pod Cache or Shared Volume)

The next example is a native Kubernetes construct that Release supports for containers in a pod to share a filesystem path that stays on a node. The most common use case for a hostPath volume is to store cache or build data that can be generated and re-generated as needed inside a pod. Unlike an emptyDir volume that only lasts as long as the pod does, the hostPath can last as long as the application that deploys the pods. Thus, a container could generate (or compute) files, assets, or data that could be reused or incrementally updated with the next pod deployment on the same node. Release automatically sets the correct permissions and ensures that each namespace has unique files so that data are not leaked between customers.

PVC (Long Term Persistent Storage)

The final example of a volume mount that Release offers is the ability to store data on persistent storage that is available across nodes and pods in a namespace. This long term storage is persistent and does not disappear during pod or node life cycles. Release uses Amazon Web Services (AWS) Elastic File System (EFS), which is their cloud offering of Network File System (NFSv4) storage. This allows customers to store long term data that will persist between deployments, availability zones (AZs), and node failures, and can be shared between multiple pods. The most common use cases for persistent storage of this type are for pre-production databases that need long term storage between deployments.