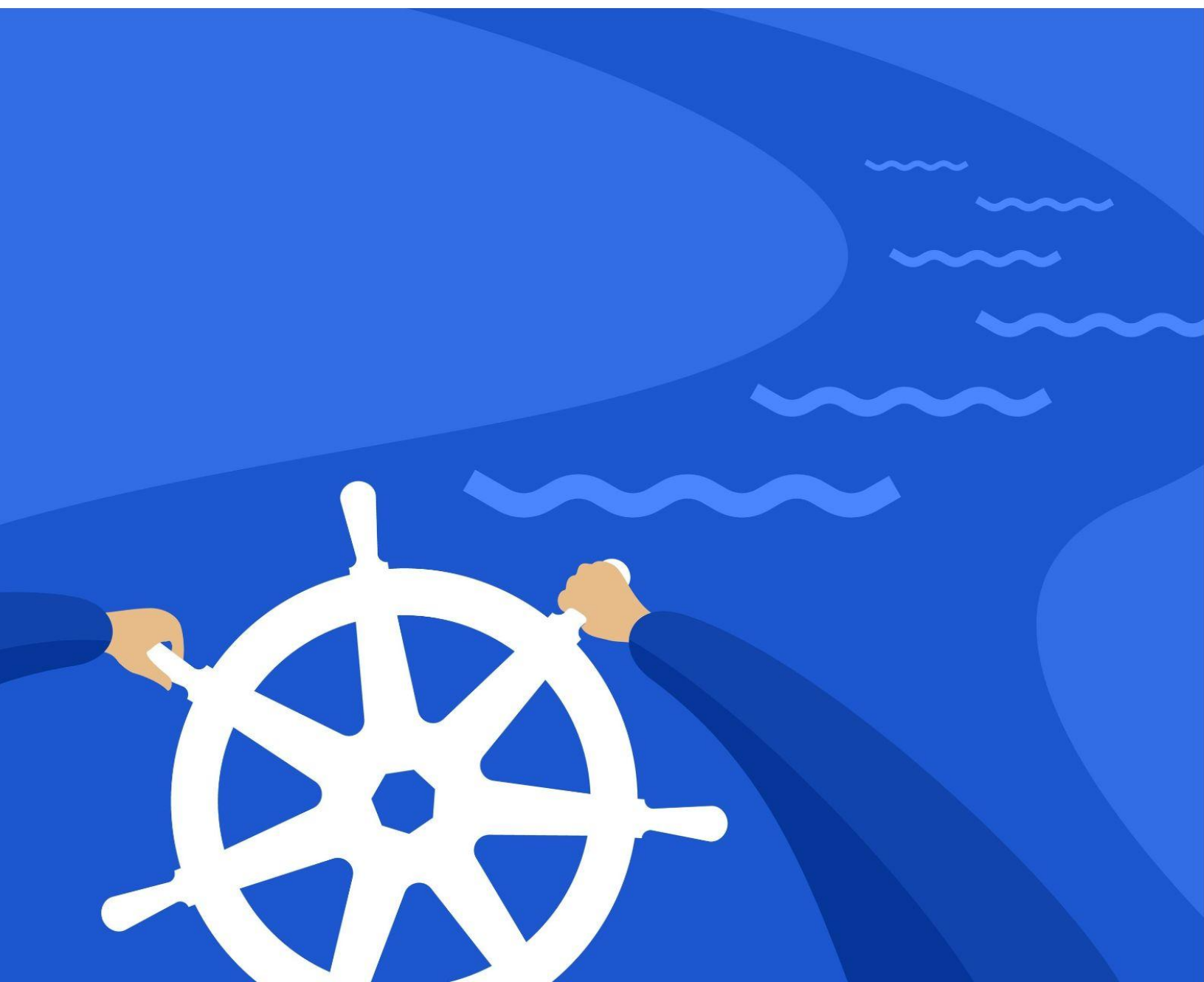


Kubernetes: A Guide for Developers and Ops Engineers



Overview	2
Introduction to Containers	3
Runtime Environments	4
Images	4
Registries	5
Introduction to Kubernetes	8
Architecture	8
Concepts & Terms	9
Nodes	9
Control Plane Nodes	9
Compute / Worker Nodes	10
Infrastructure Nodes (Optional)	10
Pods	10
Deployments	11
ReplicaSets	12
Services	12
Containerized Workload Scheduling	14
Kubernetes Scheduling Basics	14
Taints & Tolerations	14
Resource Requests & Limits	15
Affinity & Anti-Affinity Constraints	16
Liveliness & Readiness Probes	19
Workload Lifecycle & Redeployments	20
Workloads on Kubernetes vs Traditional VMs	22
Case Example: Web Applications deployed on VMs vs Kubernetes	22
Good Fit vs Bad Fit Use Cases	24
Building Applications to Survive Chaos	25
Cross Team Collaboration Needs	26
Case Example: Automated Pipeline Took Multiple Nodes Offline	26
Risks and Responsibilities of Team Members	27

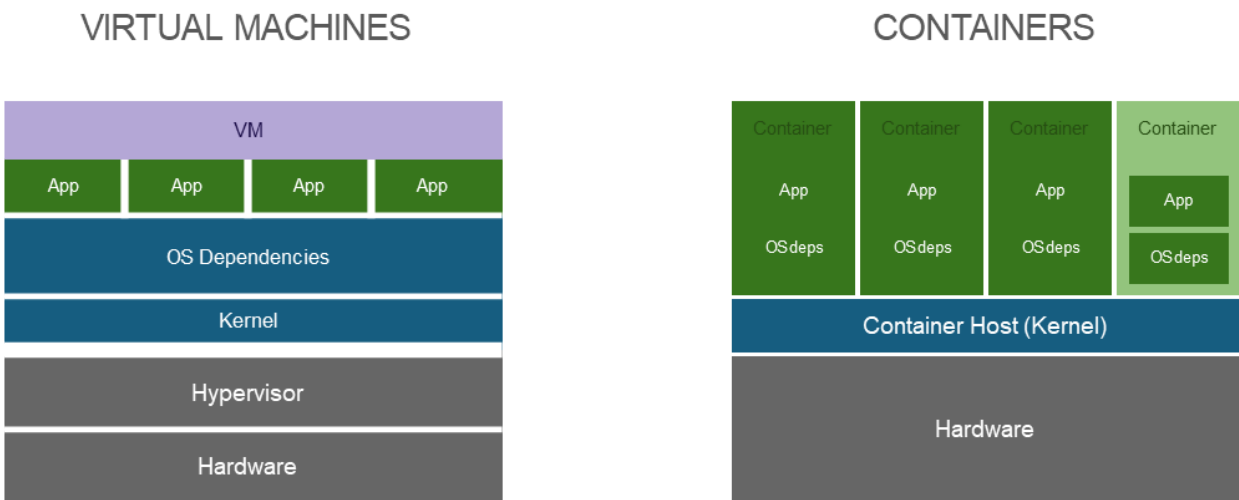
Overview

This guide serves to provide readers with Shadow-Soft's introduction to containers and Kubernetes. The intention of this guide is to highlight and establish core concepts, roles, responsibilities and design paradigms used in container orchestration environments. Having a full understanding of the operating environment and how workloads are scheduled compared to traditional infrastructure is crucial to ensuring software is developed, deployed and managed correctly. By providing real-world situational examples and use cases, readers will avoid common pitfalls and be set up for long term success.

Introduction to Containers

Containers are singular instances of an isolated executable process (ex. an instance of HTTPD, MariaDB, etc) running on a Linux-based host operating system. These processes (and all their underlying required packages) are bundled together in an easy to manage and archivable format so it can be deployed between disparate systems. This idea isn't ground breaking or new and has existed under different names (i.e. Solaris Zones) for many years. However, it wasn't until the Docker container runtime system was released in 2013 that containers became manageable enough for practical usage.

Unlike virtual machines, containers are built at the process-level. Virtual machines can be large, cumbersome and take a long time to execute in a rapid fashion; additionally, they require the overhead of a virtual hypervisor to emulate and isolate virtual machine instances from one another. Containers instead require only a shared linux kernel and a OCI compliant runtime (i.e. CRI-O, Docker, etc) to run the process on top of the underlying operating system. The isolation is then applied via the usage of Linux Namespaces and CGroups.

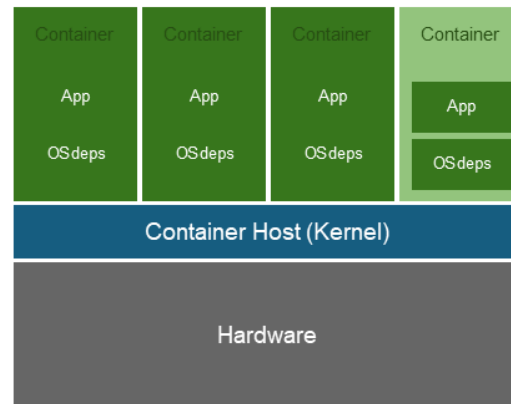


The process of managing and deploying containers has been standardized over the last several years making it safe, consistent and easy to use. Regarding the usage of Containers, three main concepts need to be understood.

- Runtime Environments
- Images
- Registries

Runtime Environments

CONTAINERS



Containers currently require Linux based operating systems to provide the necessary functionality for process isolation to be possible. To simplify the process of managing multiple isolated processes on a given host, Docker was created. Docker's main purpose initially was to provide the ability to create, run, retrieve and manage container images on a given local system.

Example Docker Commands

```
# docker build /location/of/image/content/ -t helloworld:v1
# docker run helloworld:v1 --name helloworld
# docker stop helloworld
# docker start helloworld
# docker rm helloworld
```

Eventually the image format and runtime were standardized creating the emergence of multiple easy-to-use implementations for container engines (i.e. Docker, Podman, CRI-O, etc).

Images

Container images are tar files bundled with an associated JSON file that is many times referred to as an Image Bundle. This image contains the running process, all of the required installed packages and any needed configuration details in a standardized OCI format. This format allows for containers to be easily ported across various systems regardless of the container runtime implementation (i.e. Docker, CRI-O, RKT, etc).

These images are portable to various distributions of Linux including Fedora, Debian, RHEL, CentOS, SUSE, etc. The only caveat to portability is the expected underlying linux kernel and system architecture. All container images require the running host to provide a compatible kernel (for API purposes) and system architecture (i.e. x64/x86, ARM, Power, etc). If the Linux kernel version variance is too great and/or the system architecture is not the same, the container image will not run in the target environment.

Despite this, it is important to be aware of the vendor supportability regarding image compilation and target environment. For example, Red Hat supports RHEL 6, UBI 7, and UBI 8 container images on both RHEL 7 and RHEL 8 container hosts (CoreOS is built from RHEL bits). Red Hat cannot guarantee that every permutation of a container image and host combination that exists will work. The same applies to any major vendor and should be considered when determining build and target running environments should supportability and security be concerned.

Building an image can be performed in various ways depending on your tool of choice. Most end-users leverage Docker to build container images by constructing a *Dockerfile* instance consisting of the instructions which will be used to construct the image.

Example Dockerfile

```
# vi /location/of/image/content/Dockerfile
FROM node:12-alpine
RUN apk add --no-cache python2 g++ make
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

Once a *Dockerfile* instance has been created, a single command is run to build and tag the corresponding image as is depicted below:

Example Docker Commands

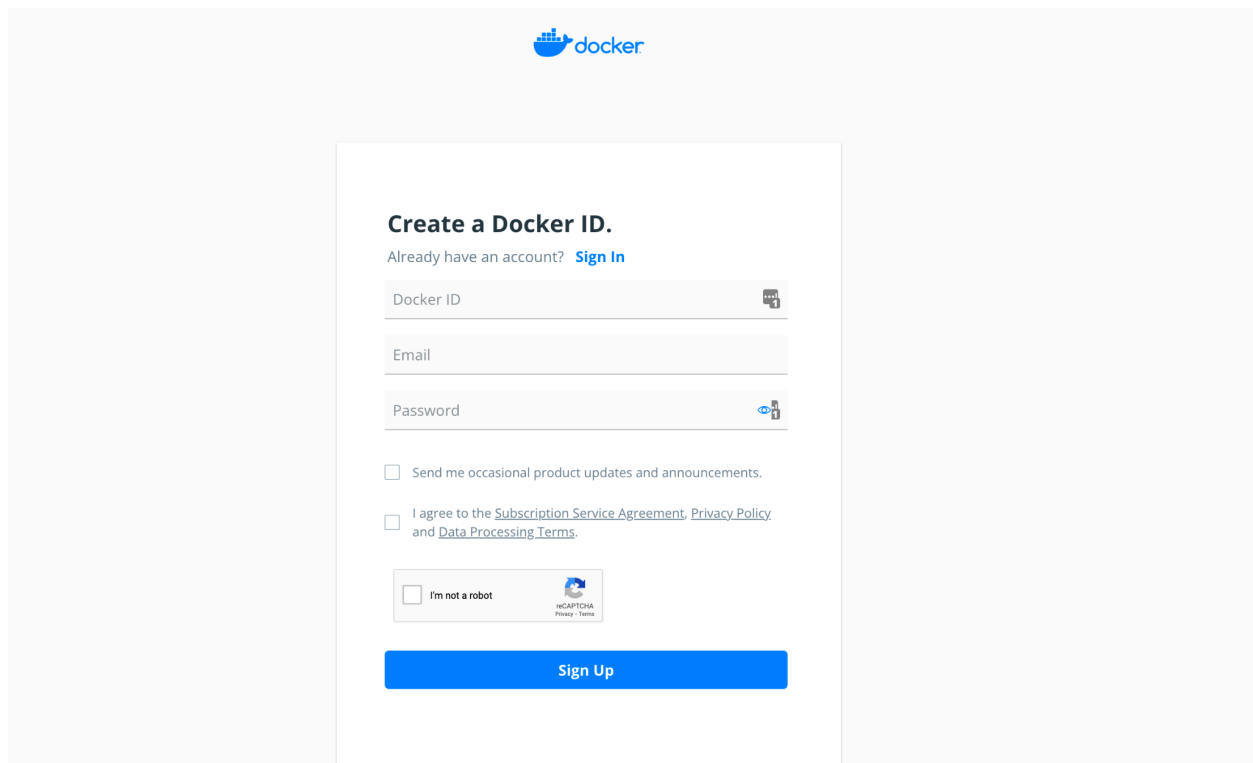
```
# docker build /location/of/image/content/ -t helloworld:v1
```

Registries

Container registries are standardized file servers that are designed to host and distribute container images across various systems. The registries offer the ability for end-users to authenticate to, authorize, upload, tag, and download images.

Many public registries exist from various product companies to host curated & standardized supported images of content provided by the given vendor (ex. Red Hat Container Catalog) . Additionally, multiple registries offerings exist privately and/or public host an users own container images (ex. Docker Hub). Curated registries are good for partners who want to deliver solutions together, while cloud-based registries are good for end users collaborating on work.

An instance of a container registry can be created by installing a series of software packages on a hosted system or by creating an account with a cloud hosting provider (ex. DockerHub).



The image shows the Docker Hub sign-up page. At the top is the Docker logo. The main heading is "Create a Docker ID." Below it, there is a link for "Already have an account? Sign In". The form contains three input fields: "Docker ID", "Email", and "Password". Below the fields are two checkboxes: "Send me occasional product updates and announcements." and "I agree to the Subscription Service Agreement, Privacy Policy and Data Processing Terms." There is also a CAPTCHA section with the text "I'm not a robot" and a CAPTCHA logo. At the bottom of the form is a blue "Sign Up" button.

Once a target registry has been selected and a container image has been locally created, it can be tagged for, uploaded to and downloaded from the given registry.

Example Docker Commands

```
# docker login registry-1.docker.io
# docker image tag helloworld:v1 registry-1.docker.io/myadmin/helloworld:v1
# docker image push registry-1.docker.io/myadmin/helloworld:v1
```

After a image has been uploaded to a given registry it can be downloaded by any container runtime that is compatible with the given image:

Example Podman & Docker Commands

```
# docker pull registry-1.docker.io/myadmin/helloworld:v1  
# podman pull registry-1.docker.io/myadmin/helloworld:v1
```


Introduction to Kubernetes

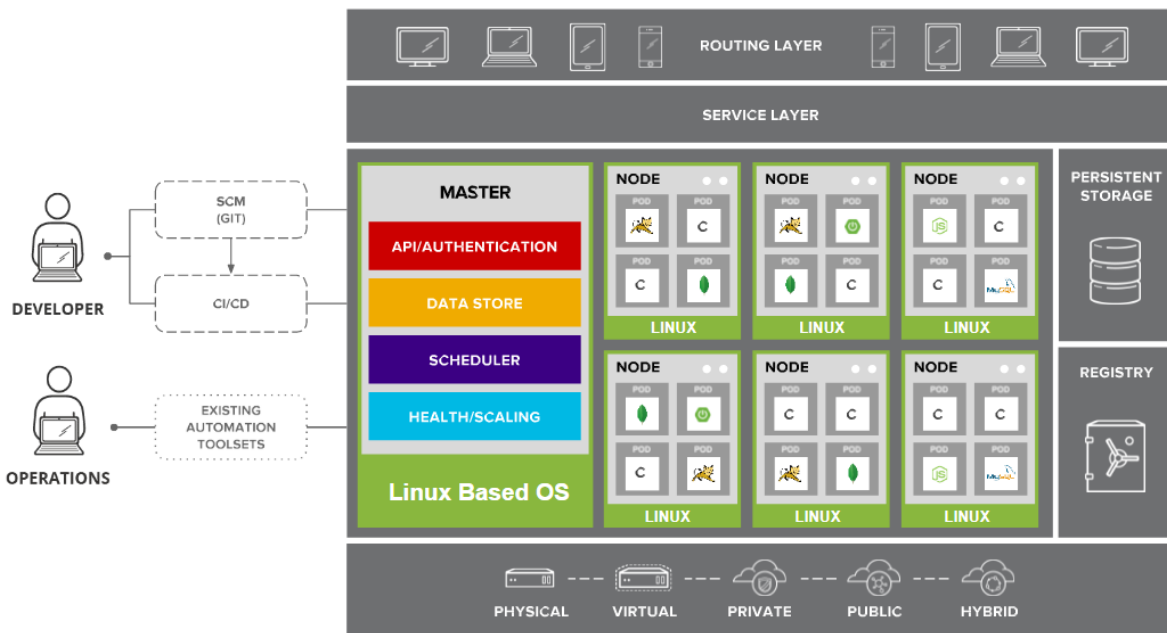
Despite containers providing a lightweight and highly distributable means of packaging and running workloads on individual systems, container runtimes (i.e. Docker, LXC, Podman, etc) by themselves lack the additional capabilities necessary to orchestrate workloads across various interconnected systems. These missing capabilities include virtual networking, storage coordination, remote management, and scheduling of workloads across various systems while maintaining operational awareness and interconnected system dependency management.

This need warranted the creation of various orchestration platforms over time including Kubernetes, Mesosphere, Docker Swarm and many home-grown solutions. Founded by Google and maintained by various large IT organizations (including Red Hat), Kubernetes has long since become the de facto standard for container orchestration.

Architecture

Abstracting away the underlying infrastructure, Kubernetes acts as a Software Defined Datacenter for containerized workloads. By providing a series of tightly coupled technology solutions, Kubernetes enables workloads to dynamically scale in a standardized way across various linux systems. Various distributions of Kubernetes have been created including Amazon EKS, Google GKE, Azure AKS, Red Hat OpenShift, SUSE Rancher and more; however, the baseline functionality, system constructs and API definitions have been standardized to create a consistent experience regardless of the selected distribution. The baseline functionality of Kubernetes includes (but is not limited to) the following:

- Container Workload Scheduling & Recovery
- Configuration State Management
- Remote Management API Interface
- User Authentication/Authorization
- Security Policy Management
- Software Defined Networking
- Standardized Storage Support
- Universal Plug-N-Play Support for Various Infrastructure Providers



The variations between distributions mostly affect the platform installation, upgrade, management experience and pre bundled add-on integrations frequently installed post deployment. These pre bundled add-ons vary between distributions but include solutions for functionality such as persistent storage, monitoring, service discovery, log aggregation, ingress networking, and more. These variations are similar to that of various distributions of Linux where the underlying kernel and baseline functions are the same, but the management experience can somewhat differ.

Concepts & Terms

Kubernetes has various standardized object constructs & terms, many of which go beyond the scope of this introduction. However, a brief understanding of a subset of these terms is essential to understanding how Kubernetes orchestrates workloads.

Nodes

Nodes refer to any host operating system instance (whether virtual or physical) included in the Kubernetes cluster capable of running a containerized workload. By default there are two types of Kubernetes Nodes included in every cluster which are responsible for different tasks (with more custom types being optional).

CONTROL PLANE NODES

Control Plane (formly "Master") nodes are responsible for performing scheduling operations, maintaining configuration state (via Etcd), authenticating/authorizing operations and hosting the remote management API interface. These nodes are

explicitly blocked by default from running user-defined workloads. As a standard operating practice, each cluster is normally deployed with 3 running instances of Control Plane Nodes (minimum) to ensure high-availability and reliability of its hosted components should a node failure occur.



All component services provided by Kubernetes run as a containerized linux process. In the event a specific Kubernetes provided function (i.e. etcd - the configuration state management software) on a local node ends unexpectedly, the given containerized process will automatically be redeployed by the scheduler. However, there are some underlying linux subsystems that these container workloads rely on (i.e. kubelet, podman, etc) which cannot be restored through the scheduler.

COMPUTE / WORKER NODES

Compute Nodes are all nodes existing within a cluster which have the distinct purpose of running user-defined workloads. These nodes can vary in memory/cpu footprint and can be configured to define unique attributes (ex. host information) about a given instance to ensure specific workloads are deployed where they can run successfully.

INFRASTRUCTURE NODES (OPTIONAL)

Infrastructure Nodes (originally coined by Red Hat) are optional but recommended specialized Compute Nodes which run targeted shared service workloads. Traditionally these nodes are designed to run services such as Ingress Routers, Log Monitoring, Storage, and more such that they are further isolated (by node) from user-defined workloads. This provides an additional layer of stability and security by ensuring there is no potential process leakage (i.e. memory and cpu usage) between a specific user-defined workload and those required to keep an entire cluster fully operational.

Pods

Pods are the smallest object construct that exists in Kubernetes. Traditionally, pods refer to a single containerized process running on a specified node in a Kubernetes cluster. However, instances also exist where multiple containerized processes run inside the same individual pod. The rationale for running multiple containers in the same pod is only specifically used in instances where two container workloads MUST run on the exact same node at all times. An example of this would be if every instance of a containerized workload required that a web proxy run on the same

host as the workload itself. In all other instances, a pod should be equivalent to one running container instance.

Deployments

Deployments are standardized Kubernetes constructs which outline the desired state of a user-defined workload. Deployment configurations provide a variety of options including the associated containerized images to deploy, references to backend persistent storage, number of replicas to run, network port access requirements, and more.

Example Deployment Configuration

```
# vi deployment-ex.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
              name: http-web-svc
```

```
# kubectl -f deployment-ex.yml
```

```
# kubectl get deployments
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3/3	3	3	18s

```
# kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-75675f5897-7ci7o	1/1	Running	0	18s
nginx-deployment-75675f5897-kzszj	1/1	Running	0	18s
nginx-deployment-75675f5897-qqcnn	1/1	Running	0	18s



All deployments are initially network isolated to the internal Kubernetes cluster software-defined network. Each Pod is provided its own internal individual IP address. The complexity of network isolation between various workloads greatly depends on the network policy configuration of the associated cluster and the underlying network provider selected (i.e. Flannel, Calico, OpenShiftSDN, etc).

ReplicaSets

ReplicaSets are a Kubernetes construct which represents the current running instance count of a given series of pods and the desired number that should be running. If at any point, the number of instances running varies from the desired count, ReplicaSets are responsible for correcting this by either increasing or decreasing the number of pods actively scheduled.

Command or Code Block description

```
# kubectl get replicaset
NAME                DESIRED  CURRENT  READY  AGE
nginx-deployment-75675f5897  3        3        3      18s
```

Services

Services are responsible for providing a means to expose a series of pods as an internal DNS name, IP address and port number which can be addressed globally within a cluster. The set of pods targeted by a Service is usually determined by a selector. This selector looks for pods with a particular label (Ex. "app: nginx" was used previously in the example above) and designates traffic to these downstream pods accordingly. Once deployed, a Service will proxy traffic from itself to one of the various backend selected pods. This is especially useful as pods will receive new IP addresses every time they are recreated and the Service endpoint will automatically and transparently be aware of all these network changes in real-time.

Example Service Configuration

```
# vi service-example.yml
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
```

```
selector:  
  app: nginx  
ports:  
- name: name-of-service-port  
  protocol: TCP  
  port: 80  
  targetPort: http-web-svc
```

```
# kubectl -f service-example.yml
```



The behavior of traffic distribution for a Service to its downstream pods vary by the configuration of the underlying kube-proxy itself. For more information on which configuration to use and how configuration changes can be made to the global kube-proxy, please review the following:

<https://kubernetes.io/docs/concepts/services-networking/service/#configuration>

Containerized Workload Scheduling

Kubernetes is a Software Defined Datacenter for containerized workloads providing a pool of shared compute resources for all deployed workloads. Many configurable parameters exist which can impact how/where workloads get scheduled as well as actions that trigger deployment. As such it is important to understand the basic mechanics of how Kubernetes allocates & schedules deployments to ensure workloads are designed in a resilient and organized manner.

Kubernetes Scheduling Basics

All deployments running in Kubernetes (including both user-defined and those provided) are orchestrated by the built-in scheduling system. The scheduler watches for new pod creation requests that have not been assigned a node. The scheduler then determines the best fit node for that pod to run on based on the pods individual different requirements. Nodes that meet the minimum scheduling requirements for a pod are considered "feasible" nodes which are scored against a set of predefined functions. Some of the factors taken into consideration include collective resource requirements, hardware / software / policy constraints, affinity / anti-affinity specifications, persistent storage availability, etc. If none of the nodes are considered suitable, the pod remains unscheduled until adequate placement is determined.

The following topic areas cover some of the most common considerations used during workload scheduling. Understanding these optional configuration settings aims to provide a greater deal of context for considerations when considering how workloads should be designed and deployed.



The topic areas presented are not an exhaustive list for all the optional scheduling considerations but are the most commonly used and core concepts leveraged.

Taints & Tolerations

Taints are properties placed on a node configuration to specify rules which will prevent a particular workload from being schedulable to the associated system. The purpose of this mechanism is great node segments which should only be for running a specific type of workload targeted for the given system.

Example tainting a given node

```
# kubectl taint nodes node1 key1=value1:NoSchedule
```

The above reference places a taint on node *node1*. The taint has a key of *key1*, a value of *value1*, and taint effect *NoSchedule*. This means that no pod will be able to schedule onto *node1* unless it has a toleration rule overriding this default behavior.

Example Toleration on Pod Definition

```
# vi pod-example.yml
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  tolerations:
  - key: "key1"
    operator: "Exists"
    effect: "NoSchedule"
```

Resource Requests & Limits

Unlike virtual machines, by default containerized workloads do not include any resource constraints to maintain the capacity necessary to keep a workload operational. Additionally, these workloads do not define by default any limitations regarding the quantity of resources which should not be exceeded to protect other workloads running in the same environment. As such, it is important that all workloads include these parameters when possible. All additional workloads which cannot have their resources parameterized in this way should be isolated from other workloads by targeting specific nodes designated for this type of risk. These settings will be taken into account during scheduling to ensure adequate system resources are available on the target node.

Example Requests & Limits Configuration

```
# vi pod-example.yml
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
```



```
containers:
- name: app
  image: images.my-company.example/app:v4
  resources:
    requests:
      memory: "64Mi"
      cpu: "250m"
    limits:
      memory: "128Mi"
      cpu: "500m"
- name: log-aggregator
  image: images.my-company.example/log-aggregator:v6
  resources:
    requests:
      memory: "64Mi"
      cpu: "250m"
    limits:
      memory: "128Mi"
      cpu: "500m"
```



It is important to note that limits and requests will help prevent issues with system resources being overcommitted or consumed on a target node. However, depending on the nature and language of the underlying containerized process it is still possible for memory leaks to still occur which can affect other workloads running on the same system.

Affinity & Anti-Affinity Constraints

Unlike taints which prevent workloads from running on a specified host, Node Affinity is a property that informs a workload to prefer (or hard require) that it be scheduled on a predefined set of nodes. Rules can be defined as *soft* or *hard* to inform the scheduler of the degree of the requirement including the *weight* of the given rule (versus other rules). These rules allow for a great deal of flexibility including which systems to target based on Node information or even which Nodes to target based on other concurrent workloads running on the given system; This provides users with the ability to define rules for which Pods can be co-located on a target node.

Example Affinity Rules Configuration

```
# vi pod-example.yml
---
apiVersion: v1
```

```

kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/os
                operator: In
                values:
                  - linux
            preferredDuringSchedulingIgnoredDuringExecution:
              - weight: 1
                preference:
                  matchExpressions:
                    - key: another-node-label-key
                      operator: In
                      values:
                        - another-node-label-value
    containers:
      - name: with-node-affinity
        image: k8s.gcr.io/pause:2.0

```



Affinity rules are not the only method for targeting a selected system. The most simple and basic method for targeting a specific system (as a hard requirement) is using the concept of node selectors which include the ability to target systems based on optional parameters (ex. if the target node is running on SSDs). For more information on node selectors, please review the following:

<https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes/#create-a-pod-that-gets-scheduled-to-your-chosen-node>

In addition to Affinity rules, Kubernetes provides the concept of Inter-Pod Affinity & Anti-Affinity. These rules take the form "this Pod should (or, in the case of anti-affinity, should not) run in an *X environment* if *X* is already running one or more Pods that meet rule *Y*. The uniqueness of these rules is that *X environment* can be any topology domain including a specific node, rack, cloud provider availability zone or region.

Example Anti-Affinity Rules Configuration

```

# vi pod-example.yml
---
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - S1
        topologyKey: topology.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security
              operator: In
              values:
              - S2
          topologyKey: topology.kubernetes.io/zone
  containers:
  - name: with-pod-affinity
    image: k8s.gcr.io/pause:2.0

```

Without these rules, it's possible that for a given user-defined workload, all pods (or at least a subset) will potentially end up running on one individual node rather than being distributed across various nodes. This can be troublesome & increase downtime during node update/failure workloads since multiple instances of a user-defined workload will be taken offline at the same time.



The number and description for all configuration options possible for Affinity & Anti-Affinity rules goes well beyond the scope of this document. For more information, please review the following:

<https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#affinity-and-anti-affinity>

Liveliness & Readiness Probes

Definable in a pod definition, once deployed, liveness and readiness probes are checked by the local running nodes kubelet process to ensure the pod is running and ready to accept traffic. By default, once all of the containers (within a given pod) main process is fully running, the pod is considered functional and will start accepting traffic. However, in many instances a process could be running and deadlocked or malfunctioning. Therefore it is important to define these checks such that redeployment of containerized services can operate as expected should an issue occur.

Example Liveliness & Readiness Probe Configuration

```
# vi deployment-example.yml
---
...

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
readinessProbe:
  exec:
    command:
      - cat
      - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

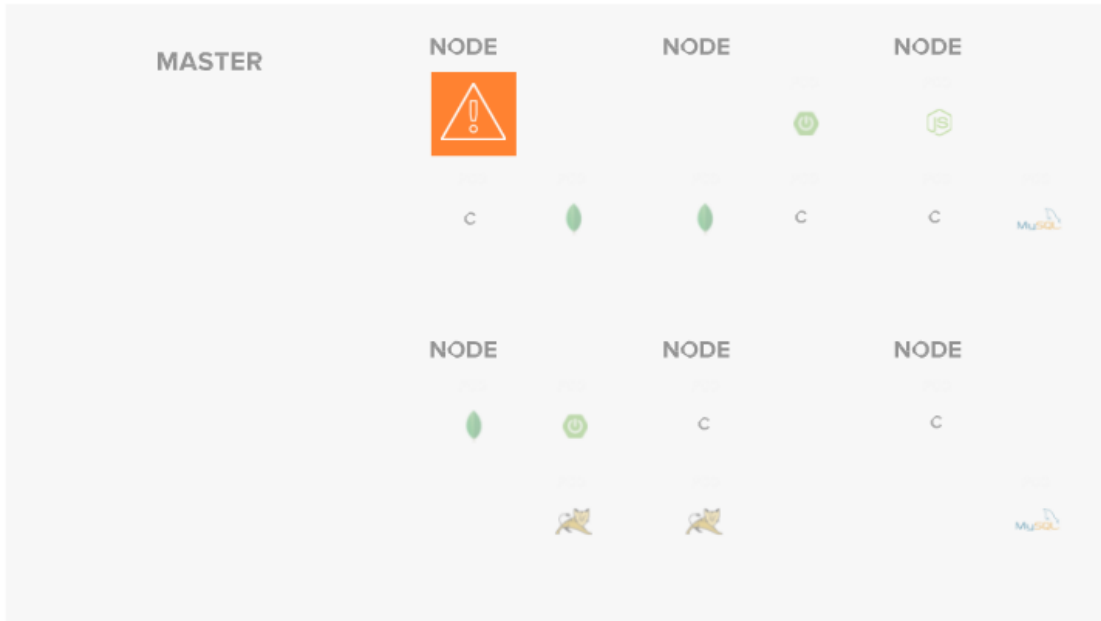


For more information regarding how to configure probes and the purpose of each probe individually, please refer to the following:

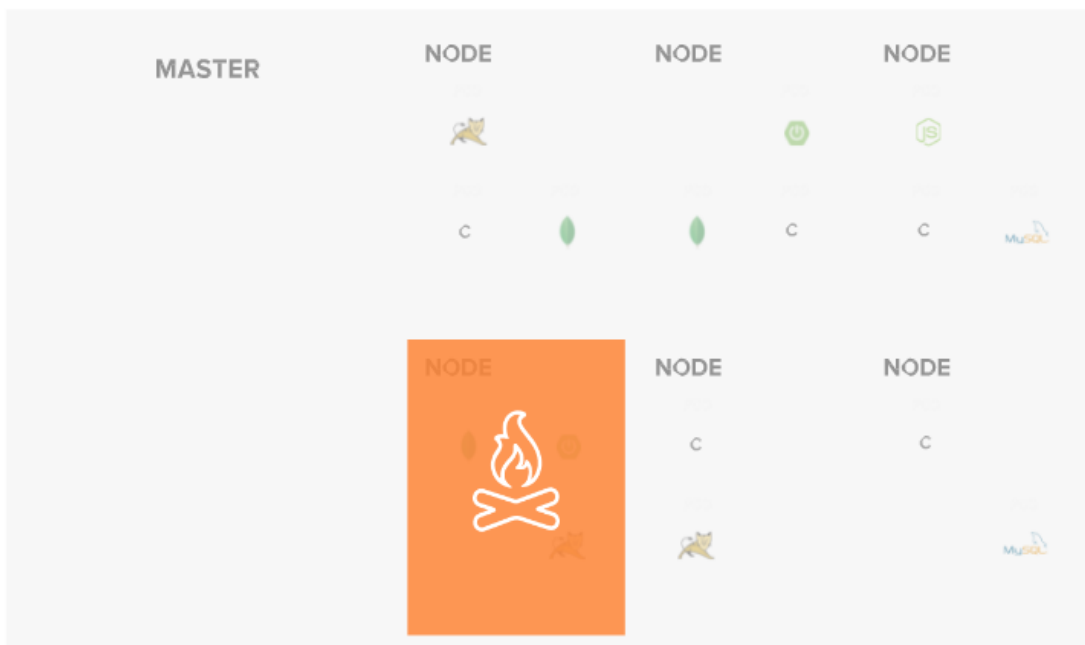
<https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>

Workload Lifecycle & Redeployments

Many instances exist where workloads actively deployed will change. If a liveness probe fails, a containerized workload will be considered unhealthy. At this point, the running pod will be terminated and a new pod will be scheduled to replace the current instance. This new pod will go through the same scheduling process as a new workload which means the instance will likely be deployed to a different node in the cluster.



Regardless of where the new instance is deployed, the new pod will be provisioned with a new IP address. This happens transparently and is automatically orchestrated behind any defined service object associated with the given deployment. This same behavior will automatically occur should any node go offline whether due to a restart, failure or update event.



Furthermore, pod autoscaling exists which can cause the number of pods to increase/decrease based on various metrics (i.e. mem, cpu, etc). It is important to

understand that these changes can occur theoretically at any time to ensure deployments are designed to operate in this type of environment.

Workloads on Kubernetes vs Traditional VMs

Kubernetes deployments have very distinct behavior characteristics which need to be considered during software development and deployment alike. Unlike traditional virtual machine workloads which are long-lived and traditionally point-solution specific, Kubernetes deployments are subject to the following:

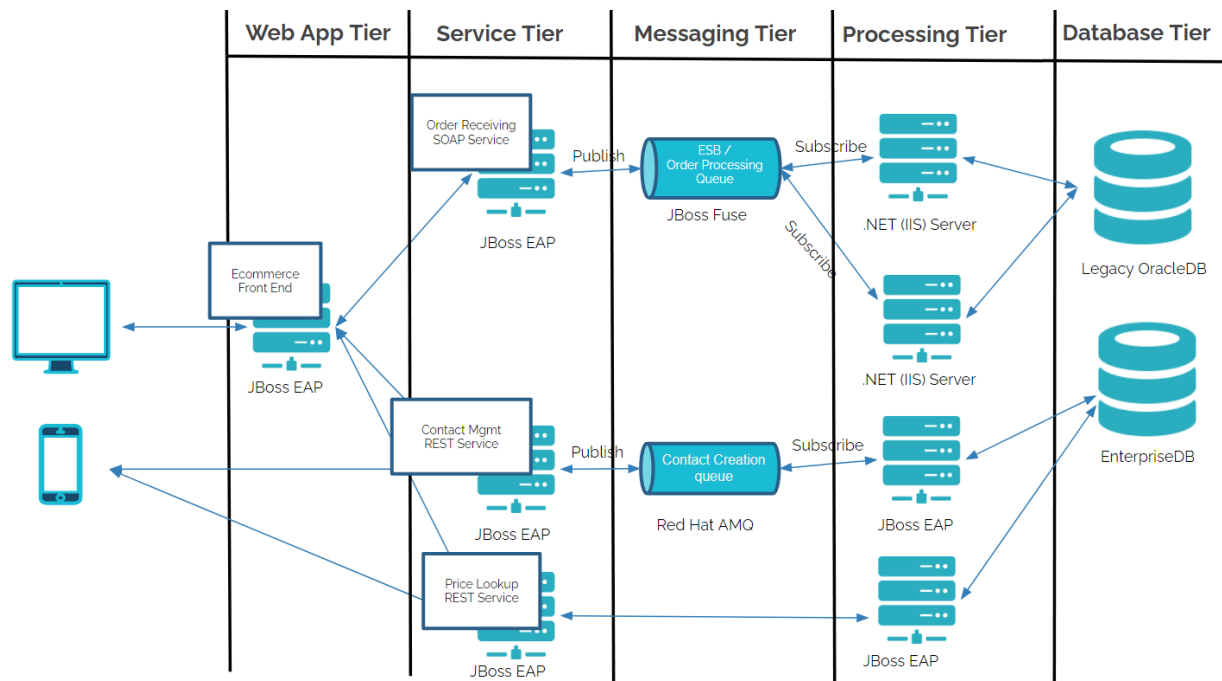
1. Shared host operating system resources (i.e. Memory, CPU, Filesystem Resources)
2. Shared cluster resources (i.e. Persistent storage, IP allocation, cluster-wide mem/cpu, etc)
3. Frequent workload horizontal scaling events
4. Frequent node horizontal scaling events
5. Frequent changes to IP addresses
6. Frequent process reload / restart events
7. Ephemeral-based by default (including system logging and local storage)

Due to these variance in characteristics, user-defined workloads normally require partial or complete redevelopment to functionally and efficiently operate within a Kubernetes cluster.

Case Example: Web Applications deployed on VMs vs Kubernetes

Many organizations have the impression that deploying a workload into Kubernetes without any changes will automatically increase their ability to scale to meet the demand of their customers. Though this is possible, the method of scaling, deploying and orchestrating workloads is so vastly different from traditional environments that frequently Kubernetes will bring to the surface existing issues arising from software development patterns that may have remained unknown previously.

Like many customer instances, a 10 year old web application and series of backend web services actively deployed in JBoss EAP 7.2 were set to be migrated to Kubernetes. These various components had gone through various iterations over the years but the overall architecture remained the same. The front-end application required the usage of sticky sessions from a front-end load balancer to not interrupt user connections and many of the backend services required stateful data sharing.



Without making any changes to the underlying application and backend services, migration began. Each component (i.e. front end application, backend services, messaging brokers, databases, etc) was converted into individual Kubernetes deployments with their own series of pods. Immediate challenges were noticed as user traffic would rotate to different deployed pods in the environment and no shared caching mechanism was being leveraged. This led to inconsistent behavior (frequent request retries and re-logins required) when user requests were attempted against the frontend application. Furthermore, when an instance of a pod died or additional pods were added, transactional information was frequently lost.

This additionally became problematic with backend SQL databases when deployment changes occurred as these systems were never designed to have the data remain consistent when frequent environment changes occurred. As such, as scaling events occurred rapidly, data was frequently lost due to synchronization challenges.

Eventually the entire architecture was rebuilt on stateless microservices. Additionally, front-end shared caching mechanisms were implemented for authentication purposes and the databases were swapped for systems built for the ground up to run in Kubernetes (i.e. CockroachDB). A Kubernetes-based persistent storage solution (i.e. Portworx) was added to the environment and a backup solution (i.e. Kasten K10) was implemented to ensure workloads could be safely redeployed in the event of a total cluster failure. This is just one example of why it is crucial that the technology selected and application design be considered prior to beginning any migration efforts to this style of environment.

Good Fit vs Bad Fit Use Cases

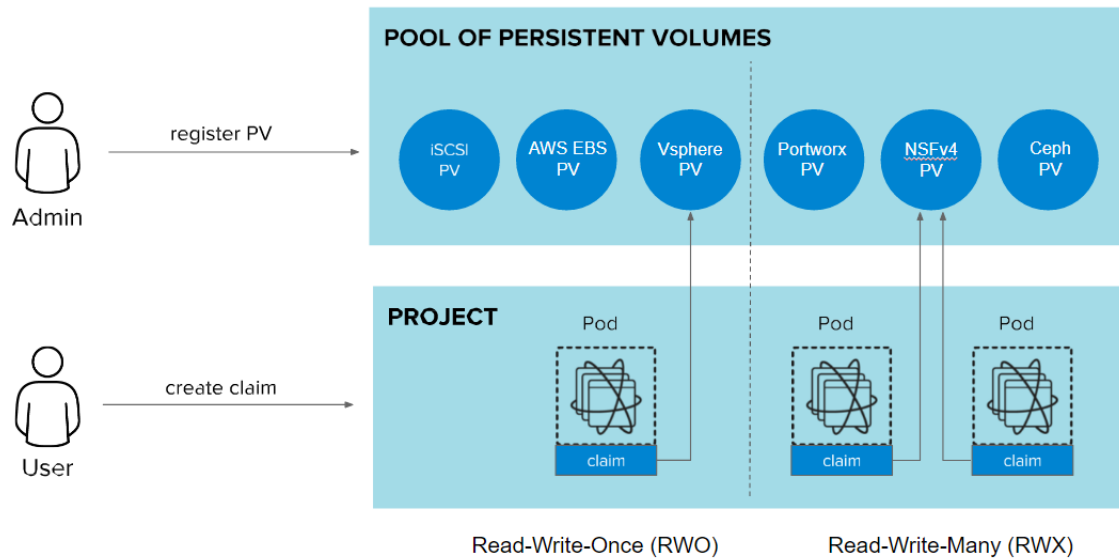
Various types of user-defined workloads can be deployed within Kubernetes. However, as we have seen in our previous examples, it is important to understand what is a good fit, bad fit and where design considerations need to be made. The types of workloads that are traditionally a good fit include:

- Stateless Web Application
- Web Applications with Distributed Caching Mechanisms
- Stateless Web Services
- Non-relational databases
- Backend Processing Jobs
- Specialized Messaging Brokers / Queues (i.e. KubeMQ)

In comparison, various workload types exist which need additional considerations prior to migrating to Kubernetes. The workload types include:

- Stateful Web Applications
- SQL Databases
- Traditional Message Brokers / Queues
- Anything requiring persistent storage

Truly anything requiring persistent storage needs to be thoroughly tested and evaluated prior to being considered production-ready in Kubernetes. Unlike a traditional hosting environment where storage usage is relatively static and relative to the running host, Kubernetes workloads scale and shift between hosts frequently. This means the underlying storage provider selected must be well understood to ensure that it can seamlessly migrate from one node to another instantaneously and that it is capable of real-time replication to ensure data consistency across all deployment instances.



Additionally, design considerations must be made considering how well these more traditional solutions function during scaling and deployment change events. If for example a SQL instance has a sudden outage or scale event occurs, how is data integrity guaranteed and does the data remain consistent across instances? Furthermore, in the event an entire cluster outage occurs, how will the workload be redeployed to a new environment? During these types of windows, will any data loss occur that is acceptable to the business?

Building Applications to Survive Chaos

Since Kubernetes workloads are subject to scaling events, node updates, networking changes and more, the lifecycle of a given containerized workload is usually short-lived lasting anywhere from minutes to hours. Additionally, despite system isolation, due to the unique nature of how containerized workloads are orchestrated, these shared resources can become overcommitted and potentially impact the stability of other workloads running on the same nodes.

For example, if workloads are deployed without the usage of requests/limits or a given workload has a major memory leak, there is a potential that other workloads running on the same system could run out of available resources and become unresponsive. As such, it is important that applications and services deployed on Kubernetes be designed from the ground up to be resilient to frequent system changes.

Cross Team Collaboration Needs

Because of the complexity of Kubernetes and its various capabilities, using Kubernetes effectively at scale requires a clear line of communication between various teams and a great deal of system governance through automation to reduce the potential for human error when running disparate workloads in production. Disappointingly, there is no one-size-fits-all approach to effectively implementing a governance strategy as the designed methodology will be greatly impacted by a number of factors including:

1. Cultural / Organization Standards & Maturity
2. Workloads Variations (i.e. Data Science/Machine Learning, Databases, Web Applications, Web Services, etc)
3. Underlying Infrastructure Features (Node Autoscalers, Availability Zones, etc)
4. Persistent Storage Requirements
5. Security Requirements
6. Service Level Agreements
7. Available Automation Solutions

Case Example: Automated Pipeline Took Multiple Nodes Offline

When moving to Kubernetes, many organizations not only migrate old workloads without first optimizing their design but continue to operate in very siloed working groups where Development and Operations are completely separated. However, operating in a Kubernetes environment without a clear governance strategy and well-defined communication patterns can often result in catastrophe.

Frequently, developers are given access to Kubernetes environments to administrate workloads they have built and maintained. On the surface, this may not seem like an issue but because of the way Kubernetes orchestrates deployments, it is possible that systems can become unstable when proper operating procedures aren't followed.

One situation that frequently comes up is around the usage of CICD as it relates to Kubernetes. In most organizations, the CICD solution (ex. Jenkins) is maintained by Operations where the pipeline generated is built and maintained by Development teams. In a particular instance, an organization had a Kubernetes cluster running in AWS with Jenkins contributing CICD jobs to the environment. The Development team created a pipeline which was responsible for deploying a workload, running a series of integration tests and then cleaning up the workload when completed. Unfortunately, this pipeline took numerous nodes in a given cluster offline causing the cluster to be unstable.

The pipeline began by creating a temporary NFS instance, deploying the workload to be tested, mounting the storage and then running the integration test. The problem in this situation was the clean up job which deleted the temporary NFS instance prior to deleting the workload. Since Kubernetes nodes are linux instances, when the NFS server got removed, the nodes running the pipeline's application workload had their file systems locked-up in perpetuity waiting for the NFS share to reconnect. The only way to fix the situation was to either login in as root on the host and forcibly remove the mount or restart the nodes.

The worst problem in this entire scenario, is it took weeks to debug the issue because the Operations team was unaware this pipeline job was running in the first place to make the correlation between the job and the environment instability. This created a chaotic situation which crippled workloads at seemingly random times.

Risks and Responsibilities of Team Members

Regardless, a Kubernetes strategy should be derived from the perspective that the platform operates as a Software Defined Datacenter. Like all datacenters, in the event a system failure or major security events occurs, there is a large risk that all data and services deployed within the Kubernetes cluster will be lost (especially when persistent storage is a concern). As such, the organizational usage and governance strategy around Kubernetes should have a well thought and designed plan to ensure system-wide & individual workload stability alike. Additionally, a well defined security posture and disaster recovery plan for individual workloads is critical.

Considering individual team responsibilities, user-defined workloads are co-located and scheduled amongst a series of shared cluster resources. It is crucial that limits and requests exist on all workloads and that affinity/anti-affinity rules are in place to ensure workloads operate in true HA at all times. This is frequently enforced by ensuring end-users do not have direct access to Production or Staging clusters beyond read access and that configuration state of workloads is enforced through the usage of Continuous Delivery solutions (i.e. ArgoCD, Fleet, etc) via Service Accounts. These Continuous Delivery jobs should include lint testing rules for the required safety parameters and a change review board should oversee every code commit to a release branch before it is accepted and deployed to an environment. Additionally, identical Staging environments should exist for testing purposes to not only test the functionality of the user-defined workload but to further understand the impact of these workloads against other workloads running within the same cluster.

Lower-end environments (i.e. Dev/Test environments) can potentially have more flexibility to their usage and governance depending on the organization needs. However, it is important that shared lower environments maintain operational readiness to not interrupt development efforts. As such, it is frequently recommended that users are provisioned limited access to segmented Kubernetes namespaces and that workloads are isolated enough from other end-users to prevent memory leak and thread usage issues from causing system wide stability problems.

About Shadow-Soft

Shadow-Soft is an award-winning Kubernetes systems integrator, specializing in helping companies simplify and optimize their IT environments. With a team of experienced consultants and proprietary modernization frameworks, Shadow-Soft is the partner of choice for manufacturing, logistics, retail, and finance companies looking to leverage their infrastructure and applications to Make Optimal Possible©.

Visit shadow-soft.com to learn more about what we do and how we help clients.