**Kubernetes Overview**



**Kubernetes** has quickly become the de facto standard for container orchestration and management. As more organizations adopt Kubernetes, there is a growing need for Kubernetes skills and expertise. This will take you from Kubernetes fundamentals all the way to advanced management, security, and governance.

**1. Understanding Kubernetes Fundamentals**

To start, it’s important to understand what Kubernetes is and why it has become so popular.

**Kubernetes** is an **open-source** c**ontainer orchestration engine** for automating deployment, scaling, and management of containerized applications. It helps with containerized application lifecycle management.

**Some key benefits of Kubernetes include:**

* **Automated rollouts and rollbacks** — Kubernetes progressively rolls out changes to applications via declaritive configuration, while monitoring application health to ensure it doesn’t kill all instances if something goes wrong. Rollbacks can quickly undo changes.
* **Self-healing** — Restarts containers automatically if they fail, replaces and reschedules containers when nodes die, and doesn’t advertise dead containers to clients until they are revived. This increases availability.
* **Horizontal scaling** — Scale applications up or down easily through a simple command, UI, or auto-scaling. Kubernetes can scale to handle extremely large clusters.
* **Service discovery and load balancing** — No need to modify application code to use an unfamiliar service discovery mechanism. Kubernetes provides automatic load balancing for containerized applications using simple abstractions.
* **Storage orchestration** — Automatically mount storage systems and make storage available to containers.
* **Automated rollouts and rollbacks** — Roll out and roll back application changes easily through the Kubernetes API. Automatically check for pods running the current version and stop rollout if issues emerge.
* **Secret and configuration management** — Deploy and update secrets and application configuration without rebuilding container images and without exposing secrets in config files.

Clearly, **Kubernetes** provides extremely useful abstractions that help with many complex tasks involved in managing containerized infrastructure and applications. But to take full advantage of what Kubernetes offers, you need to understand some key concepts.

**Kubernetes Architecture**

Kubernetes follows a client-server architecture:

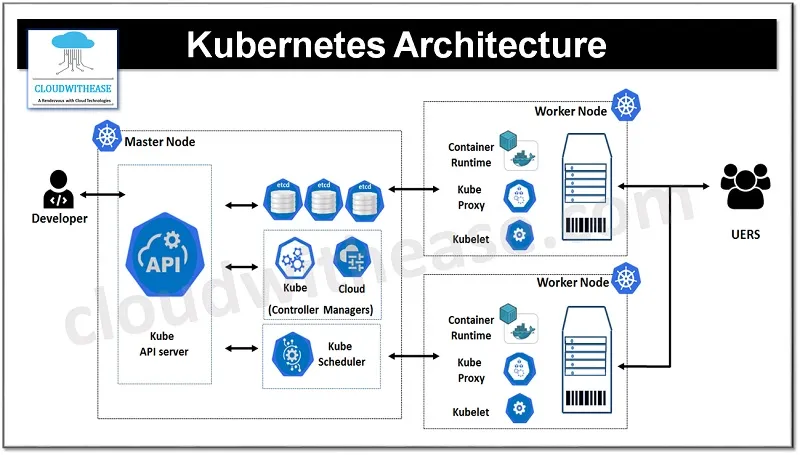


Image:- [https://cloudwithease.com](https://cloudwithease.com/what-is-kubernetes/)

* **Kubernetes Master** — The brains behind Kubernetes cluster responsible for maintaining desired state. Includes these components:
* **kube-apiserver** — Frontend of the control plane, exposes Kubernetes API.
* **etcd** — Consistent and highly-available key value store used to store all Kubernetes data.
* **kube-scheduler** — Watches for newly created pods and selects a node for them to run on.
* **kube-controller-manager** — Runs controller processes like replication controller and endpoint controller.
* **Kubernetes Nodes** — The workers that run containerized applications. Includes:
* **kubelet** — Agent that runs on each node to receive pod specs via API server and ensure containers described are running and healthy.
* **kube-proxy** — Network proxy and load balancer for Kubernetes networking service implemented using IP tables rules.
* **Container Runtime** — Software to run containers like Docker, containerd, CRI-O etc.

This architecture allows Kubernetes to scale horizontally while maintaining high availability. If any master component fails, a new one can be spun up and state is preserved in etcd.

**Kubernetes Building Blocks**

**Kubernetes** uses some key abstractions to represent applications and infrastructure:

* **Pods** — The smallest deployable units that hold one or more tightly coupled containers that share resources like storage and networking. Containers in a pod also have access to shared volumes for persistent data.
* **Services** — Provides named abstraction to allow loose coupling between dependent pods, with options for load balancing and service discovery.
* **Volumes** — Allows data to survive container restarts and provides data persistence.
* **Namespaces** — Provides isolation for teams and applications via virtual clusters backed by the same physical cluster.

These building blocks come together to provide patterns for running various types of workloads, including stateless apps, stateful apps, data processing jobs, and more.

Now that you understand Kubernetes basics, let’s move on to core concepts…

**2. Kubernetes Core Concepts**

To leverage the full power of Kubernetes, you need to become familiar with some key concepts. This section will provide an in-depth overview.

**Pods**

Pods represent a logical application and hold one or more tightly coupled containers. Containers within a pod share an IP address, IPC namespace, hostname, and other resources. Pods abstract away management of containerized applications and facilitate horizontal scaling.

Pods have a lifecycle and go through phases like Pending, Running, Succeeded, Failed, and Unknown. The Kubernetes control plane manages pod lifecycles end-to-end.

Pods provide two shared resources to their containers — networking and storage. Containers within pods share the same IP address and port space. And containers can mount shared storage volumes.

Pods are designed for disposability and do not provide guarantees around availability and persistence. So even though pods share resources and dependencies, it is antibhetical to Kubernetes design to have multiple tightly coupled processes together in one pod.

Instead, related pods should be grouped using higher-level abstractions like services and replicasets.

**Services**

Services provide named abstractions that allow loose coupling between dependent pods. They integrate with service discovery mechanisms to provide dynamic networking.

Services automatically load balance across pods based on labels. This provides flexible networking without needing to manage names or IPs.

There are several types of Kubernetes services with differing networking models:

* **ClusterIP** — The default type that exposes pods only within the cluster based on an internal IP address. This IP stays constant regardless of pod lifecycles.
* **NodePort** — Exposes pods across cluster nodes using NAT and a static port. Allows calling the service from outside the cluster via NodeIP:NodePort.
* **LoadBalancer** — Creates a cloud load balancer to access pods. This provides a static IP exposed externally. Only supported on some clouds.
* **ExternalName** — Exposes pods using an arbitrary DNS name by returning a CNAME record pointing at the external DNS address.

Services enable loose coupling between pods and provide flexibility around networking requirements.

**ReplicaSets**

ReplicaSets maintain a stable set of replica pods running at any given time. They help guarantee the availability of pods.

They use pod templates that specify the pod properties, along with a label selector that determines which pods belong to the replica set. They ensure specified number of pods match the selector continuously run.

Replica sets provide self-healing capabilities by creating or removing pods to maintain desired state, providing resilience. They also seamlessly support horizontal scaling of pods.

While replica sets manage pod replicas, deployments manage replica sets and provide additional capabilities like graceful rolling updates to applications.

**Deployments**

Deployments provide declarative updates to pods and replica sets via rolling updates and rollbacks. This allows deploying new versions of applications gradually while retaining availability.

Deployments consist of configurations like:

* Pod templates specifying container images, ports, volumes etc.
* Label selector identifying pods the deployment manages
* Replication strategy

They monitor rollout status and health to ensure availability. Deployments integrate seamlessly with horizontal pod auto-scalers that adjust replicas automatically based on CPU usage or custom metrics.

Together, pods, replica sets, and deployments provide complementary building blocks for scalable, resilient application deployment and management.

**Namespaces**

**Namespaces** partition Kubernetes clusters into virtual sub-clusters and provide isolation for teams and applications. Resources created in one namespace are hidden from other namespaces.

Namespaces allow using the same names for resources like pods or services in different namespaces. And users and access policies can differ across namespaces.

Some use cases for namespaces include:

* **Partition development, test and production** — Prevents intermingling of resources between environments.
* **Multi-tenancy** — Allows allocating resources between multiple teams or applications.
* **Access control policies** — Isolate access to resources across organization groups.

Namespaces become essential in large clusters with multiple teams and varied workloads sharing Kubernetes.

**Storage**

Storage management is a key consideration in running stateful applications. Kubernetes provides multiple storage abstraction objects.

**Volumes** allow mounting storage filesystems into containers. Pods can access the volumes as normal filesystems regardless of the backend storage provider.

**PersistentVolumes (PVs)** provision storage at cluster level instead of pod level, enabling lifecycle independent of any pod. Multiple pods can access the PV.

A **PersistentVolumeClaim (PVC)** allows a user to consume a PV without knowing implementation details. Pod definitions can request specific storage capacity via PVC.

Using PVCs and PVs enables storage orchestration without apps needing to interact directly with storage APIs. It also facilitates on-demand dynamic provisioning from cloud storage pools since PVs can integrate with public cloud storage providers.

Additionally, **StorageClasses** define “classes” of storage with different properties like performance. This simplifies PVC configuration by auto-provisioning based on classes instead of needing to specify full requirements every time.

Together, these abstractions provide powerful storage orchestration facilities as an integrated part of the Kubernetes environment.

**Configuration**

For maximum portability across environments, Kubernetes aims to de couple configuration artifacts from container images. This allows changing configuration without rebuilding images.

The **ConfigMap** API resource provides injection of config data like settings, license keys etc into pods. Pods reference config data values through environment variables or config files. ConfigMaps don’t provide confidentiality as they reside unencrypted in etcd — secrets solve this.

The **Secret** resource objects let you encode confidential data like passwords or keys and consume them in pods through mounted files or environment variables without exposing the values permanently. Kubernetes automatically encodes secrets, but encryption at rest depends on the backend etcd store.

These configuration mechanisms facilitate loosely coupled microservices architectures on Kubernetes.

**Kubernetes Controllers**

Kubernetes uses “controller” processes constantly running in control loops to converge current state towards desired state. Resource controllers included cover deployments, replica sets, namespace lifecycle, node lifecycle, endpoints etc.

Custom controllers build on the patterns composable API resources exposed by Kubernetes. Operator frameworks like the Kubernetes Operator SDK accelerate developing specialized controllers with CRDs tailored to specific apps or infrastructure components like databases. These provide simplified management for complex apps.

That covers core Kubernetes concepts! Next we will explore…

**3. Kubernetes Application Management**

Now you understand Kubernetes basics — let’s explore best practices and processes around application management.

**Application Deployment**

The most common way to deploy applications on Kubernetes is by using workload resources like deployments.

Deployments provide features like:

* Declarative configuration for desired application state
* Automated rollouts and rollbacks
* Revision history and audit logs
* Horizontal auto-scaling

Additionally, Kubernetes facilitates practices like blue-green, canary, or A/B testing deployments that reduce risk. For example, different versions run simultaneously and traffic shifts gradually.

**Application Observability**

Observability is crucial for maintaining availability and diagnosing issues through data like metrics, logs, and traces.

The **kubelet** provides basic health checking via readiness probes. Additionally many Kubernetes-native monitoring tools provided enhanced observability:

* **Prometheus** for storing and querying metrics
* **Grafana** for visualizing monitoring data
* **Jaeger** for distributed tracing
* **Fluentd** for log aggregation

These integrate seamlessly with Kubernetes APIs.

**Application Configuration**

Kubernetes promotes immutable infrastructure principles — container images bundled with app code are redeployed rather than changed in-place.

So application configuration should remain externalized from container images using options like:

* **ConfigMaps** — Externalize non-confidential config
* **Secrets** — Externalize confidential data such as credentials
* **Environment variables** — Set through Kubernetes manifests

This allows updating configuration separately from code changes.

**Application Security**

Kubernetes provides various application security options:

* **Namespace isolation** — Limit damage from vulnerabilities
* **RBAC policies** — Control access granularly
* **mTLS auth** — Secure Kubernetes API communication
* **Network policies** — Restrict communication between pods
* **Admission webhooks** — Validate resources during creation

Additionally, tools like **Falco** or **Sysdig Falco** can monitor and audit runtime application behavior and activity for threat detection.

**Storage and Data Management**

For stateful apps, structured data storage is required beyond ephemeral storage in pods. Kubernetes provides abstractions like:

* **Volumes** — Ephemeral per-pod storage
* **Persistent Volumes** — Long-term cluster storage

And mechanisms like **StatefulSets** to manage stateful apps and data.

Higher level databases or caches should run in containers alongside applications. But critical databases may run best on dedicated IaaS virtual machines.

**CI/CD Integration**

Kubernetes declarative model integrates nicely with infrastructure-as-code tooling like Ansible, Terraform, and Pulumi for provisioning and managing Kubernetes cluster resources.

And declarative application configs allow seamless promotion of applications through CI/CD pipelines across lower and higher level environments. No manual intervention necessary for app upgrades.

Source control systems like Git track changes to Kubernetes manifests and other configs and enforce version control and audit trails.

**Infrastructure Management**

In terms of managing underlying infrastructure for a Kubernetes cluster:

* Use infrastructure-as-code tools to provision the cluster
* Configure high availability (HA) for Kubernetes control plane
* Choose certified conformant tooling and distro
* Plan networking carefully for performance and security

Managed services like **Amazon EKS**, **Azure Kubernetes Service (AKS)**, **Google Kubernetes Engine (GKE)** reduce the burden of tasks like upgrading Kubernetes versions.

Next let’s explore advanced Kubernetes capabilities…

**4. Advanced Kubernetes**

You now know the critical basics — but Kubernetes has many additional powerful capabilities:

**Scheduling**

The **kube-scheduler** assigns pods to cluster nodes balancing resource utilization and additional policies:

* **Affinity/anti-affinity** — Ensure workloads run alongside or away from other pods
* **Taints/tolerations** — Dedicate nodes to pods and control which pods run where
* **Priority** — Define scheduling priority for certain pods
* **Preemption** — Reclaim resources by removing lower priority pods

These facilities provide very fine-grained control over workload placement.

**Autoscaling**

Kubernetes supports autoscaling pods horizontally by adding or removing replicas automatically based on metrics like CPU:

* **HorizontalPodAutoscaler (HPA)** — Automatically scales pods within one namespace
* **ClusterAutoscaler** — Automatically scales worker nodes in the whole cluster

Additionally, vertical pod autoscaling automatically adjusts resource requests and limits based on historical utilization to right size pods.

These automation capabilities reduce administrative burden significantly.

**Batch Workloads**

In additional to services, Kubernetes supports short-lived batch workloads via **Jobs** which run pods to completion.

**CronJobs** build on jobs and provide time-based scheduled execution, like cron.

These abstractions expand the types of workloads Kubernetes can automate beyond stateless long-running apps and services.

**Serverless Computing**

**Kubernetes Events** provide an event streaming mechanism that automatically trigger custom resources known as **EventResources** in response to events happening across cluster. This enables event-driven automation.

The **Knative** framework leverages this along with abstraction resources like Knative Services, Builds etc to enable a serverless execution model on top of Kubernetes. This facilitates finer-grained autoscaling and eventing capabilities.

Serverless computing patterns in Kubernetes continue gaining traction for lighter-weight workloads.

**5. Kubernetes Networking**

Given networking is intrinsic to how distributed systems communicate, understanding Kubernetes networking deeply is critical.

**Cluster Networking**

All pods can communicate with all other pods across cluster nodes without NAT, thanks to a **pod network**. This relies on native VPC-CNI plugins or overlay networks from projects like Flannel, Calico, Cilium etc.

Pods get their own IP addresses from this flat pod network along with:

* Port mapping from containers to pods
* Iptables/kube-proxy for distributed load balancing to pods

This forms the foundation for all communications.

**Ingress Controllers**

Ingress provides externally reachable URLs, load balancing, SSL termination and name-based virtual hosting for services within the cluster. Widely used ingress controllers include:

* NGINX
* Contour
* HAProxy
* Traefik
* Istio ingress gateway

These negotiate external traffic to cluster services and provide critical edge routing and management.

**Service Mesh**

Service meshes like Linkerd and Istio build on basic Kubernetes networking to provide:

* Traffic management — Split testing, circuit breaking, timeouts, retries
* Observability — Metrics, logs, traces are automatically captured
* Security — mTLS encryption between services

They work by injecting an extra container proxy throughout pod’s data path for cross-cutting capabilities.

Service mesh adoption is growing given increased microservices complexity.

**Container Network Interface (CNI)**

CNI consists of specifications and libraries to write plugins that configures networking for pods within Kubernetes environments consistently. Many CNI plugins exist for integration with VPCs, on-prem networks, BGP etc.

**Network Policies**

You can use **NetworkPolicies** to restrict communication between pods through rules specifying allowed inbound and outbound connectivity. Policies get implemented by the pod network.

Fine-grained network controls help reduce potential attack surface.

As you can see, Kubernetes provides very advanced networking capabilities — now let’s talk production management next.

**6. Kubernetes in Production**

Running Kubernetes reliably in production requires following sound operational patterns and processes.

**Release Management**

A **GitOps** based approach for managing infrastructure and application definitions via declarative configs stored and version controlled in Git provides excellent release management.

It also enables promoting apps across environments like so:

* Developer edits files locally
* Submits PR to trigger review then merge to trunk
* CD pipelines execute on commit against trunk
* Changing parameters in configs controls deployments

No manual intervention needed. Review process also reduces surprises.

**Cluster Lifecycle**

Provisioning Kubernetes clusters should follow infrastructure-as-code principles using tools like Terraform, Ansible, or Pulumi. This ensures repeatability and enables treating clusters as cattle.

Perform regular Kubernetes version upgrades to stay reasonably current and benefit from new features and security fixes. Managed services like EKS, AKS and GKE reduce the upgrade burden.

Use Kubernetes conformance test suite to validate cluster implementation conforms to API specifications. This reduces nasty surprises down the line.

**Monitoring, Logging and Tracing**

Essential operational aspects include:

* Resource Monitoring — Track cluster component and infrastructure health. Popular tools include Prometheus operator, Datadog.
* Application Monitoring — Instrument apps for metrics, logs collection and traces. Tools include Prometheus, Grafana, Jaeger.
* Vertical Pod Autoscaling — Automatically tune resource requests and limits for efficient resource usage.
* Service Mesh Telemetry — Tools like Istio automatically gather rich telemetry data across services.
* Log Aggregation — Use tools like Fluentd, Loki and Elastic to aggregate logs.
* Tracing — Monitor distributed request flows across microservices with Jaeger or Zipkin or OpenTelemetry.

Observability is impossible to forego in production.

**Security and Compliance**

Running secure and compliant Kubernetes clusters in production mandates:

* Hardening host OS, container runtimes, and networking
* TLS for control plane (API server) communication
* Role based access control (RBAC) for least privilege
* Security context constraints
* Network policies restricting unauthorized communications
* Kubesec and Falco to continuously audit configs and runtime
* Regular scans of images for CVEs with tools like Trivy
* Integrating with corporate authentication systems

This defense-in-depth approach limits the attack surface.

Additionally, certifications like CIS benchmarks provide best practice configs to validate. Regulated organizations commonly desire features like multi-tenancy as well.

**7. Kubernetes Ecosystem**

Beyond just the core project, a rich ecosystem of tools integrates with and extends Kubernetes. Let’s discuss key players.

**Helm**

Helm provides a package manager for deploying applications packaged as charts — a bundle of YAML templates modeling resources required. Benefits include:

* Repeatability
* Parameterization
* Versioning
* Dependency management
* Marketplace of ready apps at Artifact Hub

Helm streamlines deploying applications on Kubernetes.

**Kubernetes Operators**

Operators build on Kubernetes extensibility via CRDs and controllers to automate complex stateful applications like databases in Kubernetes. Benefits include:

* Encapsulate domain expertise required to managed complicated apps
* Provide cloud-native lifecycle management for traditional apps
* Robust error handling logic
* Near 1-click managed install

Popular examples manage MySQL, PostgreSQL, Redis etc automatically.

**Container Registries**

Container registries store and distribute container images needed by Kubernetes like:

* Quay
* Docker Hub
* Elastic Container Registry (ECR)
* Google Container Registry (GCR)

They provide optimized storage and image distribution networked with Kubernetes.

**CNCF Landscape**

The Cloud Native Computing Foundation serves as the hub for Kubernetes and many adjacent projects constituting critical cloud native technologies — prominently including service meshes and CI/CD pipelines.

Exploring the CNCF landscape provides insight into the extended tooling ecosystem powering modern software delivery.

**Managed Kubernetes Services**

Providers like AWS, Azure, and GCP offer managed Kubernetes clusters called EKS, AKS and GKE. These reduce operational burden and provide:

* Automatic upgrades
* Integrated monitoring and logging
* Secure by default configurations
* Serverless integrations on the same VPC

As Kubernetes grows in complexity, managed services gain appeal to offload undifferentiated heavy lifting.

**Kubernetes Distributions**

Many vendors offer packaged Kubernetes platform distributions with batteries included:

* Red Hat OpenShift
* VMware Tanzu
* Canonical Charmed Kubernetes
* Rancher Kubernetes Engine (RKE)
* Mirantis Kubernetes Engine

These integrate complementary tools, extensions and support contracts for enterprise production use.

The extensive and expanding ecosystem around Kubernetes multiplies what it can accomplish.

**8. Advanced Security and Governance**

Now that we’ve covered the extensive Kubernetes ecosystem, let’s focus on advanced security and governance capabilities required in highly regulated industries.

**Multi-tenancy**

Kubernetes provides multiple isolation mechanisms for safely sharing clusters between untrusted teams:

**Namespaces** — Logical isolation of resources  
**Network policies** — Isolate pod network communications  
**Resource quotas** — Limit resource consumption by namespace  
**RBAC access policies** — Restrict access to resources in namespaces

Additional security measures like **PSP policies** also constraint pods.

Together these building blocks provide strong multi-tenancy.

**Identity Federation**

Integrating Kubernetes identity and authentication with existing IAM systems is crucial for unified access policies.

Standards like **OIDC** facilitate federation with systems like AD, LDAP etc so Kubernetes inherits identities and associated privileges.

**Secrets Encryption**

By default Kubernetes secrets get base64 encoded but remain unencrypted. For security, enable **envelope encryption** provided by tools like spiffe/spire, kms providers to encrypt secrets at rest.

**Compliance Scans**

Since organizations need to validate Kubernetes configuration and security controls against compliance benchmarks continuously, tools like **kube-bench** and **kube-score** programmatically check settings against CIS policies to maintain compliant clusters.