Building Kubernetes Cluster using Terraform

Version 1.1.0
July 2021

Prepared by: himanshu_dave@persistent.com

Plan Kubernetes Cluster Deployment: Identify, define, and agree with the scope for customization of many aspects of the Kubernetes cluster deployment.

- Choice deployment mode: kubeadm or non-kubeadm
- Cluster Configuration
- Infrastructure Requirements
- HA Considerations
- CNI (networking) plugins / Network Policy Providers
- DNS configuration
- Choice of control plane: native/binary or containerized
- Component versions
- Component runtime options

    a) Docker
    b) containerd
    c) CRI-O

- Certificate generation methods
- Storage Classes
- Logging
- Testing
- Security

    1) Cloud Native Security
    2) Pod Security
    3) Kubernetes API Security
    4) Network policies for Pods
    5) Controlling Access to the Kubernetes API
    6) Securing your cluster
    7) Data encryption in transit for the control plane
    8) Data encryption at rest
    9) Secrets in Kubernetes
    10) Runtime class

Choice for deployment mode:

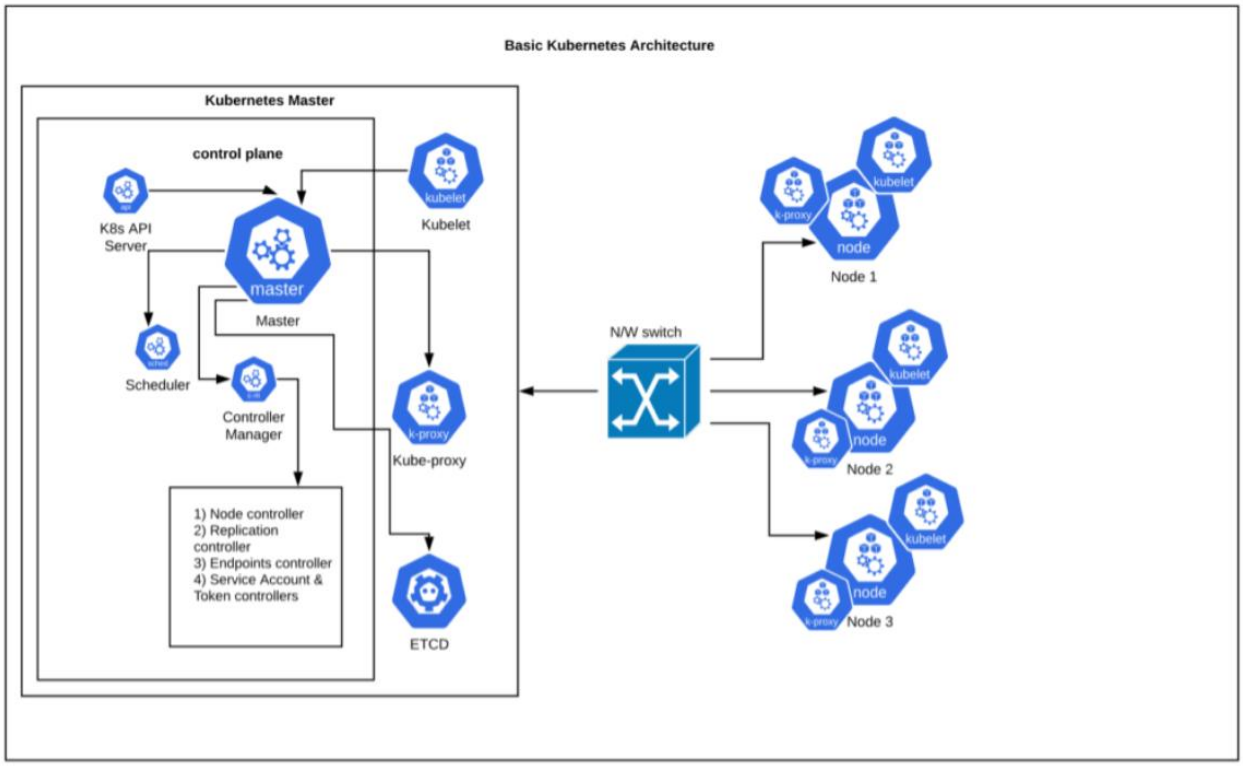
1) Using Kubeadm
   https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/
2) Non-kubeadm
   a) Kubespray
      https://kubernetes.io/docs/setup/production-environment/tools/kubespray/
   b) Kops
      https://kubernetes.io/docs/setup/production-environment/tools/kops/

Kubernetes Cluster Configurations:

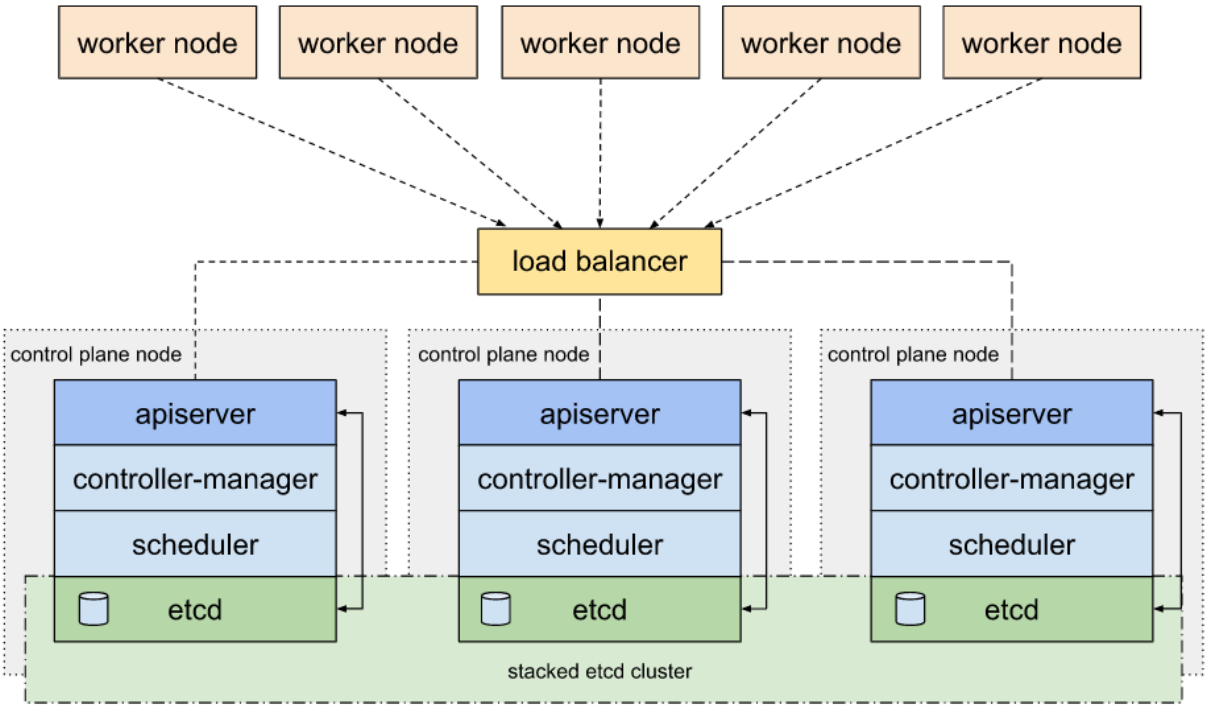
1. create a minimum viable Kubernetes cluster
   (Install a single control-plane Kubernetes cluster)
2. configuring highly available (HA) Kubernetes clusters.
   a. With stacked control plane nodes
      (Where etcd nodes are colocated with control plane nodes)
   b. With external etcd nodes
      (Where etcd runs on separate nodes from the control plane)

Design:

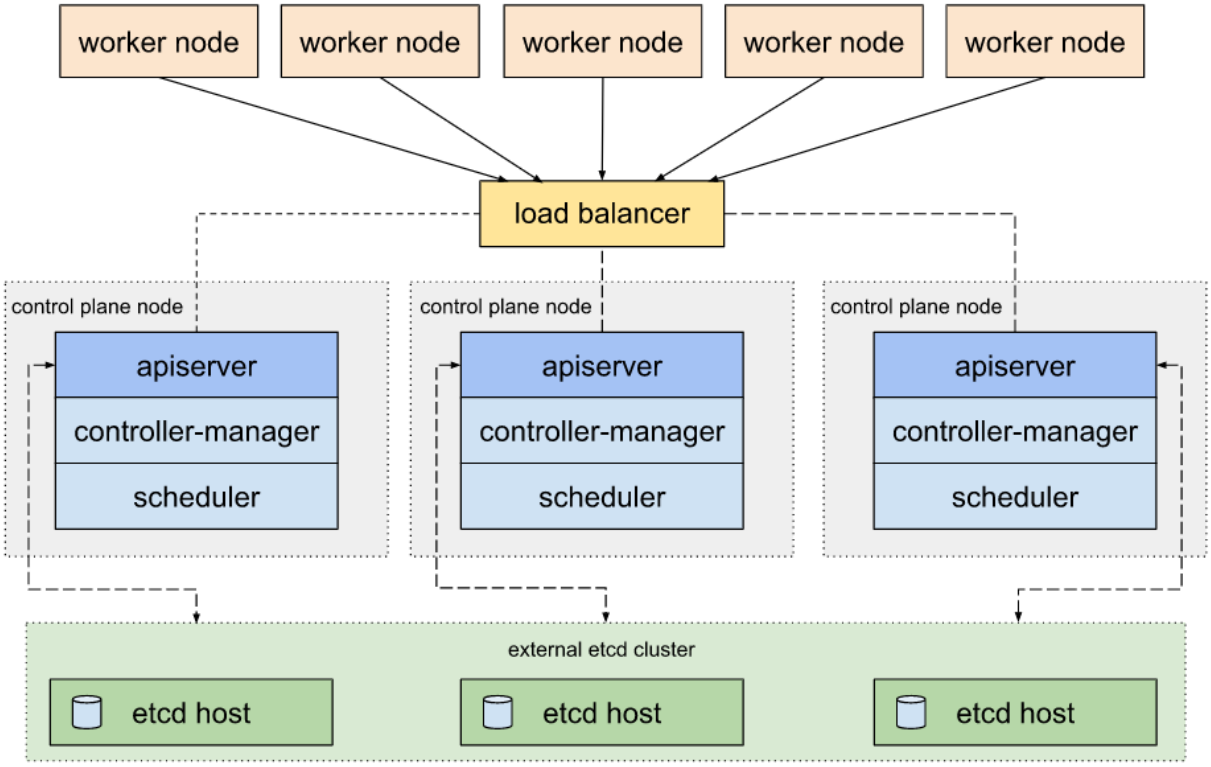
Single Control-Plane Kubernetes Cluster



kubeadm HA topology - stacked etcd



kubeadm HA topology - external etcd



Infra requirements for Option 1: A minimum viable Kubernetes cluster

a) One or more machines running a deb/rpm-compatible Linux OS; for example: Ubuntu or CentOS.
b) 2 GiB or more of RAM per machine--any less leaves little room for your apps.
c) At least 2 CPUs on the machine that you use as a control-plane node.
d) Full network connectivity among all machines in the cluster. You can use either a public or a private network.

You also need to use a version of kubeadm that can deploy the version of Kubernetes that you want to use in your new cluster

Infra requirements for Option 2: Highly available (HA) Kubernetes clusters.

For both methods 2a and 2b we need this infrastructure:

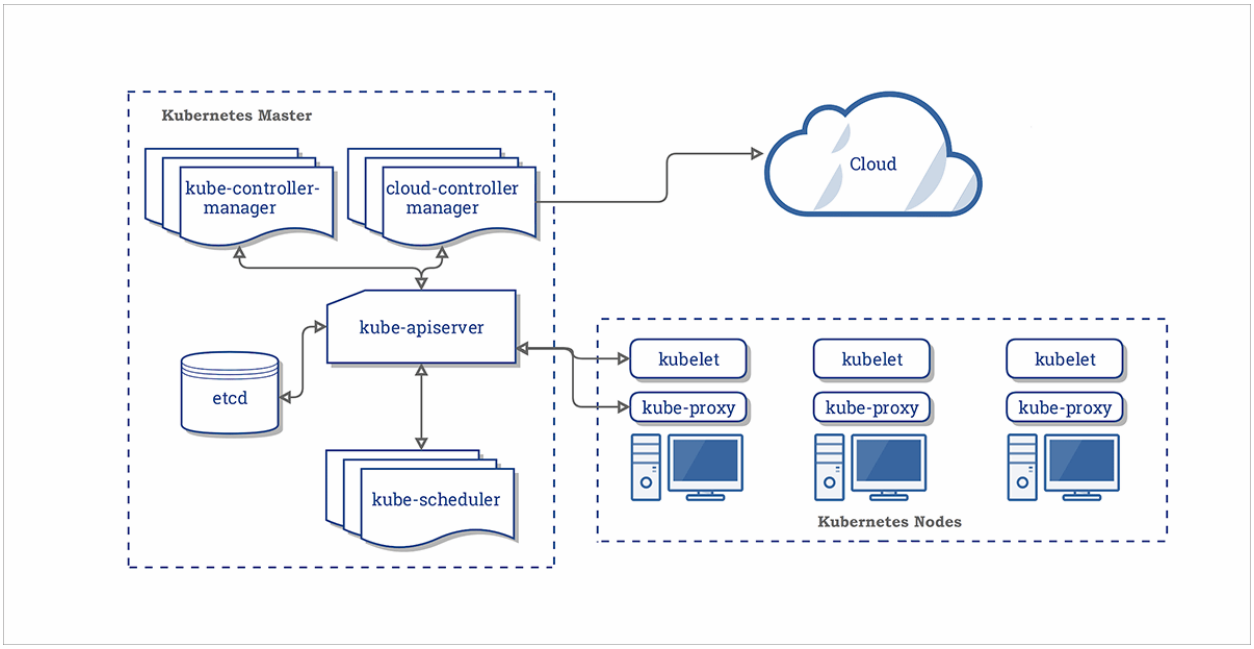
- 3 machines that meet kubeadm's minimum requirements for the control-plane nodes
- 3 machines that meet kubeadm's minimum requirements for the workers
- Full network connectivity between all machines in the cluster (public or private network)
- sudo privileges on all machines
- SSH access from one device to all nodes in the system
- kubeadm and kubelet installed on all machines. kubectl is optional.

For option 2b- the external etcd cluster only, we also need:

- Three additional machines for etcd members

Components and its versions should be compatible with each other

https://kubernetes.io/docs/concepts/overview/components/#node-components



Kubernetes master components include:

- **Kube-apiserver:** The front end of the control plane that exposes Kubernetes APIs to cluster nodes and applications.
- **Etcd:** The Kubernetes data plane, in the form of a key-value store that manages cluster-specific but not application data.
- **Kube-scheduler:** Monitors resource usage on a cluster and assigns workloads, in the form of Kubernetes pods, to one or more worker nodes based on specified policies about hardware usage, node-pod affinity, security and workload priority.
- **Kube-controller-manager:** Runs the controller processes responsible for node monitoring, replication, container deployment and security policy enforcement.
- **Cloud-controller-manager:** A feature that primarily service providers use to run cloud-specific control processes.

Kubernetes worker node components include:

- **Kubelet:** An agent that runs on each worker node.
- **Kube-proxy:** Manages network communication between cluster nodes.
- **Container runtime:** The engine that runs containers and maintains workload isolation within the OS.

High Availability Considerations:
https://github.com/kubernetes/kubeadm/blob/master/docs/ha-considerations.md#options-for-software-load-balancing

When setting up a production cluster, high availability (the cluster's ability to remain operational even if some control plane or worker nodes fail) is usually a requirement. For worker nodes, assuming that there are enough of them, this is part of the very cluster functionality. However, redundancy of control plane nodes and etcd instances needs to be catered for when planning and setting up a cluster.

Option 1: Use keepalived and haproxy
For providing load balancing from a virtual IP the combination keepalived and haproxy can be considered well-known and well-tested:

The **keepalived** service provides a virtual IP managed by a configurable health check. Due to the way the virtual IP is implemented, all the hosts between which the virtual IP is negotiated need to be in the same IP subnet.

The **haproxy** service can be configured for simple stream-based load balancing thus allowing TLS termination to be handled by the API Server instances behind it.

This combination can be run either as services on the operating system or as static pods on the control plane hosts.

Option 2: Use kube-vip

kube-vip implements both management of a virtual IP and load balancing in one service and it will be run as a static pod on the control plane nodes.

Choice of control plane: native/binary or containerized

CNI (networking) plugins: Cluster Networking

https://kubernetes.io/docs/concepts/cluster-administration/networking/#how-to-implement-the-kubernetes-networking-model

Network Policy Providers:
https://kubernetes.io/docs/tasks/administer-cluster/network-policy-provider/

1. Antrea
2. Calico
3. Cilium
4. Kube-router
5. Romana
6. Weave Net

DNS configuration for service discovery

https://kubernetes.io/docs/tasks/administer-cluster/dns-custom-nameservers/

Component runtime options:
https://kubernetes.io/docs/setup/production-environment/container-runtimes/

we need to install a container runtime into each node in the cluster so that Pods can run there. This page outlines what is involved and describes related tasks for setting up nodes.

Following is a list of common container runtimes with Kubernetes, on Linux:

- containerd
- CRI-O
- Docker

data encryption:
https://kubernetes.io/docs/tasks/administer-cluster/kms-provider/

Certificate generation methods
https://kubernetes.io/docs/setup/best-practices/certificates/

Kubernetes requires PKI certificates for authentication over TLS. If you install Kubernetes with kubeadm, the certificates that your cluster requires are automatically generated. You can also generate your own certificates -- for example, to keep your private keys more secure by not storing them on the API server. This page explains the certificates that your cluster requires.

Kubernetes requires PKI for the following operations:

- Client certificates for the kubelet to authenticate to the API server
- Server certificate for the API server endpoint
- Client certificates for administrators of the cluster to authenticate to the API server
- Client certificates for the API server to talk to the kubelets
- Client certificate for the API server to talk to etcd
- Client certificate/kubeconfig for the controller manager to talk to the API server
- Client certificate/kubeconfig for the scheduler to talk to the API server.
- Client and server certificates for the front-proxy

Storage Classes:
https://kubernetes.io/docs/concepts/storage/storage-classes/

Volume Plugin	Config Example
AWSElasticBlockStore	AWS EBS
AzureFile	Azure File
AzureDisk	Azure Disk
CephFS	-
Cinder	OpenStack Cinder
FC	-
FlexVolume	-
Flocker	-
GCEPersistentDisk	GCE PD
Glusterfs	Glusterfs

Volume Plugin	Config Example
iSCSI	-
Quobyte	Quobyte
NFS	-
RBD	Ceph RBD
VsphereVolume	vSphere
PortworxVolume	Portworx Volume
ScaleIO	ScaleIO
StorageOS	StorageOS
Local	

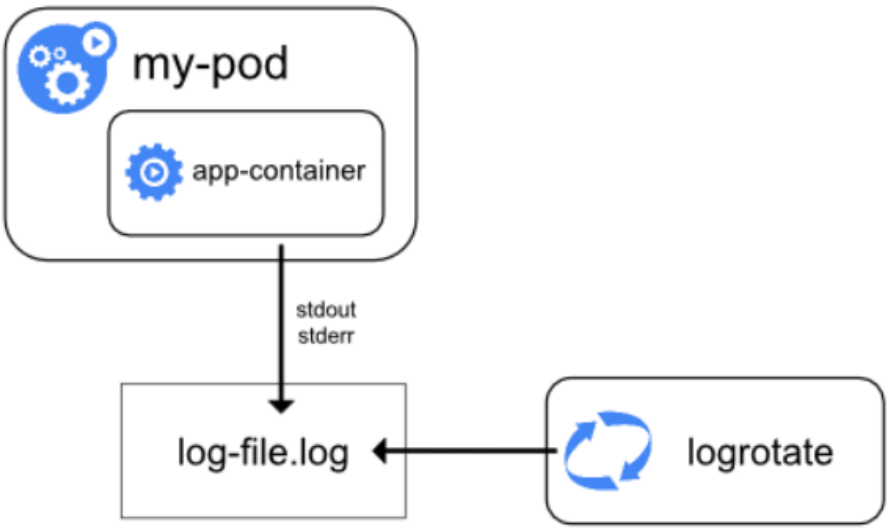
Logging

https://kubernetes.io/docs/concepts/cluster-administration/logging/

Application logs can help you understand what is happening inside your application. The logs are particularly useful for debugging problems and monitoring cluster activity.

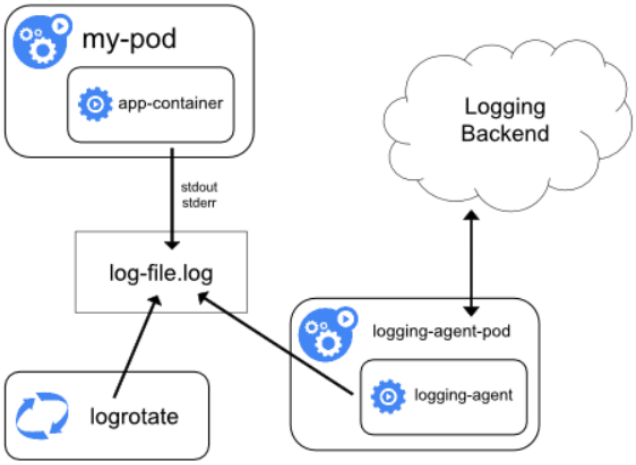
The easiest and most adopted logging method for containerized applications is writing to standard output and standard error streams.

In a cluster, logs should have a separate storage and lifecycle independent of nodes, pods, or containers. This concept is called cluster-level logging.

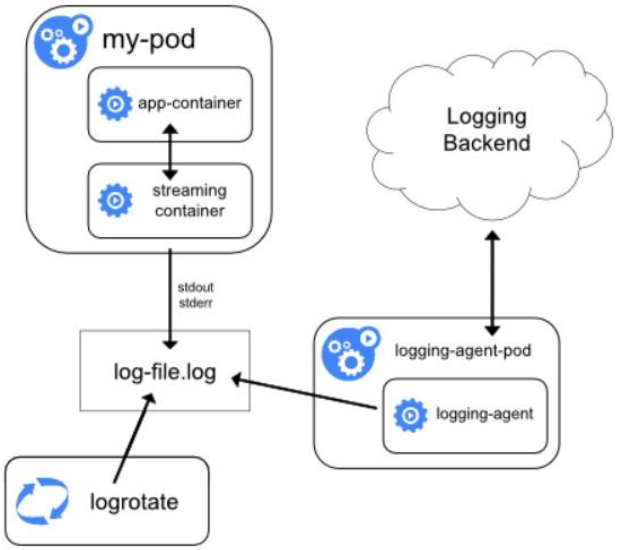


Cluster-level logging architectures require a separate backend to store, analyze, and query logs. Kubernetes does not provide a native storage solution for log data. Instead, there are many logging solutions that integrate with Kubernetes.
The following sections describe how to handle and store logs on nodes

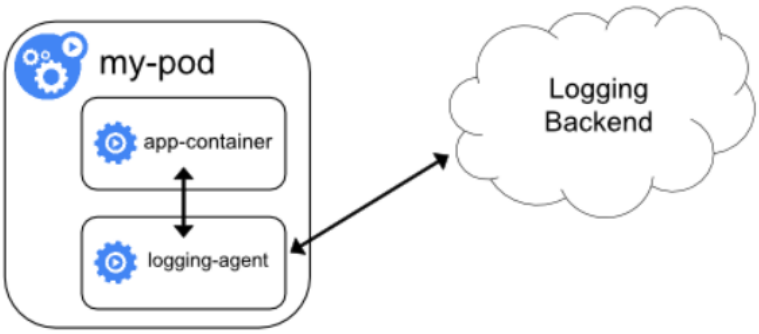
a) Use a node-level logging agent that runs on every node.



b) Include a dedicated sidecar container for logging in an application pod.



c) Push logs directly to a backend from within an application.



Testing / Validation:

1) Validate node setup
   https://kubernetes.io/docs/setup/best-practices/node-conformance/

   Node conformance test is a containerized test framework that provides a system verification and functionality test for a node. The test validates whether the node meets the minimum requirements for Kubernetes; a node that passes the test is qualified to join a Kubernetes cluster.

2) Verify that your cluster is running properly with Sonobuoy.

   Sonobuoy is a diagnostic tool that makes it easier to understand the state of a Kubernetes cluster by running a set of plugins (including Kubernetes conformance tests) in an accessible and non-destructive manner. It is a customizable, extendable, and cluster-agnostic way to generate clear, informative reports about your cluster.

   Its selective data dumps of Kubernetes resource objects and cluster nodes allow for the following use cases:

   - Integrated end-to-end (e2e) conformance-testing
   - Workload debugging
   - Custom data collection via extensible plugins

Security

- Cloud Native Security - think about security in layers.
  https://kubernetes.io/docs/concepts/security/overview/

  The 4C's of Cloud Native security are Cloud, Clusters, Containers, and Code.

  Each layer of the Cloud Native security model builds upon the next outermost layer. The Code layer benefits from strong base (Cloud, Cluster, Container) security layers

  ### Cloud

  In many ways, the Cloud (or co-located servers, or the corporate datacenter) is the trusted computing base of a Kubernetes cluster. If the Cloud layer is vulnerable (or configured in a vulnerable way) then there is no guarantee that the components built on top of this base are secure. Each cloud provider makes security recommendations for running workloads securely in their environment

  ### Cluster

  There are two areas of concern for securing Kubernetes:

  1) Securing the cluster components that are configurable
  2) Securing the applications which run in the cluster

  ### Container

  Container security is outside the scope of this guide. Here are general recommendations and links to explore this topic:

Area of Concern for Containers	Recommendation
Container Vulnerability Scanning and OS Dependency Security	As part of an image build step, you should scan your containers for known vulnerabilities.
Image Signing and Enforcement	Sign container images to maintain a system of trust for the content of your containers.
Disallow privileged users	When constructing containers, consult your documentation for how to create users inside of the containers that have the least level of operating system privilege necessary in order to carry out the goal of the container.
Use container runtime with stronger isolation	Select container runtime classes that provider

Code

Application code is one of the primary attack surfaces over which you have the most control. While securing application code is outside of the Kubernetes security topic, here are recommendations to protect application code

Area of Concern for Code	Recommendation
Access over TLS only	If your code needs to communicate by TCP, perform a TLS handshake with the client ahead of time. With the exception of a few cases, encrypt everything in transit. Going one step further, it's a good idea to encrypt network traffic between services. This can be done through a process known as mutual or mTLS which performs a two sided verification of communication between two certificate holding services.
Limiting port ranges of communication	This recommendation may be a bit self-explanatory, but wherever possible you should only expose the ports on your service that are absolutely essential for communication or metric gathering.
3rd Party Dependency Security	It is a good practice to regularly scan your application's third party libraries for known security vulnerabilities. Each programming language has a tool for performing this check automatically.
Static Code Analysis	Most languages provide a way for a snippet of code to be analyzed for any potentially unsafe coding practices. Whenever possible you should perform checks using automated tooling that can scan codebases for common security errors. Some of the tools can be found at: https://owasp.org/www-community/Source_Code_Analysis_Tools
Dynamic probing attacks	There are a few automated tools that you can run against your service to try some of the well known service attacks. These include SQL injection, CSRF, and XSS. One of the most popular dynamic analysis tools is the OWASP Zed Attack proxy tool

Production environment

Create a production-quality Kubernetes cluster

A production-quality Kubernetes cluster requires planning and preparation. If your Kubernetes cluster is to run critical workloads, it must be configured to be resilient. This page explains steps you can take to set up a production-ready cluster, or to uprate an existing cluster for production use. If you're already familiar with production setup and want the links.

Production considerations

Typically, a production Kubernetes cluster environment has more requirements than a personal learning, development, or test environment Kubernetes. A production environment may require secure access by many users, consistent availability, and the resources to adapt to changing demands.

As you decide where you want your production Kubernetes environment to live (on premises or in a cloud) and the amount of management you want to take on or hand to others, consider how your requirements for a Kubernetes cluster are influenced by the following issues:

- *Availability*. A single-machine Kubernetes learning environment has a single point of failure. Creating a highly available cluster means considering:
    - Separating the control plane from the worker nodes.
    - Replicating the control plane components on multiple nodes.
    - Load balancing traffic to the cluster's API server.
    - Having enough worker nodes available, or able to quickly become available, as changing workloads warrant it.
- *Scale*. If you expect your production Kubernetes environment to receive a stable amount of demand, you might be able to set up for the capacity you need and be done. However, if you expect demand to grow over time or change dramatically based on things like season or special events, you need to plan how to scale to relieve increased pressure from more requests to the control plane and worker nodes or scale down to reduce unused resources.
- *Security and access management*. You have full admin privileges on your own Kubernetes learning cluster. But shared clusters with important workloads, and more than one or two users, require a more refined approach to who and what can access cluster resources. You can use role-based access control (RBAC) and other security mechanisms to make sure that users and workloads can get access to the resources they need, while keeping workloads, and the cluster itself, secure. You can

set limits on the resources that users and workloads can access by managing policies and container resources.

Before building a Kubernetes production environment on your own, consider handing off some or all of this job to Turnkey Cloud Solutions providers or other Kubernetes Partners. Options include:

- ***Serverless***: Just run workloads on third-party equipment without managing a cluster at all. You will be charged for things like CPU usage, memory, and disk requests.
- ***Managed control plane***: Let the provider manage the scale and availability of the cluster's control plane, as well as handle patches and upgrades.
- ***Managed worker nodes***: Configure pools of nodes to meet your needs, then the provider makes sure those nodes are available and ready to implement upgrades when needed.
- ***Integration***: There are providers that integrate Kubernetes with other services you may need, such as storage, container registries, authentication methods, and development tools.

Whether you build a production Kubernetes cluster yourself or work with partners, review the following sections to evaluate your needs as they relate to your cluster's *control plane*, *worker nodes*, *user access*, and *workload resources*.

Production cluster setup

In a production-quality Kubernetes cluster, the control plane manages the cluster from services that can be spread across multiple computers in different ways. Each worker node, however, represents a single entity that is configured to run Kubernetes pods.

Production control plane

The simplest Kubernetes cluster has the entire control plane and worker node services running on the same machine. You can grow that environment by adding worker nodes, as reflected in the diagram illustrated in Kubernetes Components. If the cluster is meant to be available for a short period of time, or can be discarded if something goes seriously wrong, this might meet your needs.

If you need a more permanent, highly available cluster, however, you should consider ways of extending the control plane. By design, one-machine control plane services running on a single machine are not highly available. If keeping the cluster up and running and ensuring that it can be repaired if something goes wrong is important, consider these steps:

- ***Choose deployment tools***: You can deploy a control plane using tools such as kubeadm, kops, and kubespray. See Installing Kubernetes with deployment tools to learn tips for production-quality deployments using each of those deployment methods. Different Container Runtimes are available to use with your deployments.
- ***Manage certificates***: Secure communications between control plane services are implemented using certificates. Certificates are automatically generated during deployment or you can generate them using your own certificate authority. See PKI certificates and requirements for details.
- ***Configure load balancer for apiserver***: Configure a load balancer to distribute external API requests to the apiserver service instances running on different nodes. See Create an External Load Balancer for details.
- ***Separate and backup etcd service***: The etcd services can either run on the same machines as other control plane services or run on separate machines, for extra security and availability. Because etcd stores cluster configuration data, backing up the etcd database should be done regularly to ensure that you can repair that database if needed. See the etcd FAQ for details on configuring and using etcd. See Operating etcd clusters for Kubernetes and Set up a High Availability etcd cluster with kubeadm for details.
- ***Create multiple control plane systems***: For high availability, the control plane should not be limited to a single machine. If the control plane services are run by an init service (such as systemd), each service should run on at least three machines. However, running control plane services as pods in Kubernetes ensures that the replicated number of services that you request will always be available. The scheduler should be fault tolerant, but not highly available. Some deployment tools

set up Raft consensus algorithm to do leader election of Kubernetes services. If the primary goes away, another service elects itself and take over.

- *Span multiple zones*: If keeping your cluster available at all times is critical, consider creating a cluster that runs across multiple data centers, referred to as zones in cloud environments. Groups of zones are referred to as regions. By spreading a cluster across multiple zones in the same region, it can improve the chances that your cluster will continue to function even if one zone becomes unavailable. See Running in multiple zones for details.
- *Manage on-going features*: If you plan to keep your cluster over time, there are tasks you need to do to maintain its health and security. For example, if you installed with kubeadm, there are instructions to help you with Certificate Management and Upgrading kubeadm clusters. See Administer a Cluster for a longer list of Kubernetes administrative tasks.

To learn about available options when you run control plane services, see kube-apiserver, kube-controller-manager, and kube-scheduler component pages. For highly available control plane examples, see Options for Highly Available topology, Creating Highly Available clusters with kubeadm, and Operating etcd clusters for Kubernetes. See Backing up an etcd cluster for information on making an etcd backup plan.

Production worker nodes

Production-quality workloads need to be resilient and anything they rely on needs to be resilient (such as CoreDNS). Whether you manage your own control plane or have a cloud provider do it for you, you still need to consider how you want to manage your worker nodes (also referred to simply as *nodes*).

- ***Configure nodes***: Nodes can be physical or virtual machines. If you want to create and manage your own nodes, you can install a supported operating system, then add and run the appropriate Node services. Consider:
    - The demands of your workloads when you set up nodes by having appropriate memory, CPU, and disk speed and storage capacity available.
    - Whether generic computer systems will do or you have workloads that need GPU processors, Windows nodes, or VM isolation.
- ***Validate nodes***: See Valid node setup for information on how to ensure that a node meets the requirements to join a Kubernetes cluster.
- ***Add nodes to the cluster***: If you are managing your own cluster you can add nodes by setting up your own machines and either adding them manually or having them register themselves to the cluster's apiserver. See the Nodes section for information on how to set up Kubernetes to add nodes in these ways.
- ***Add Windows nodes to the cluster***: Kubernetes offers support for Windows worker nodes, allowing you to run workloads implemented in Windows containers. See Windows in Kubernetes for details.
- ***Scale nodes***: Have a plan for expanding the capacity your cluster will eventually need. See Considerations for large clusters to help determine how many nodes you need, based on the number of pods and containers you need to run. If you are managing nodes yourself, this can mean purchasing and installing your own physical equipment.
- ***Autoscale nodes***: Most cloud providers support Cluster Autoscaler to replace unhealthy nodes or grow and shrink the number of nodes as demand requires. See the Frequently Asked Questions for how the autoscaler works and Deployment for how it is implemented by different cloud providers. For on-premises, there are some virtualization platforms that can be scripted to spin up new nodes based on demand.
- ***Set up node health checks***: For important workloads, you want to make sure that the nodes and pods running on those nodes are healthy. Using the Node Problem Detector daemon, you can ensure your nodes are healthy.

Production user management

In production, you may be moving from a model where you or a small group of people are accessing the cluster to where there may potentially be dozens or hundreds of people. In a learning environment or platform prototype, you might have a single administrative account for everything you do. In production, you will want more accounts with different levels of access to different namespaces.

Taking on a production-quality cluster means deciding how you want to selectively allow access by other users. In particular, you need to select strategies for validating the identities of those who try to access your cluster (authentication) and deciding if they have permissions to do what they are asking (authorization):

- ***Authentication***: The apiserver can authenticate users using client certificates, bearer tokens, an authenticating proxy, or HTTP basic auth. You can choose which authentication methods you want to use. Using plugins, the apiserver can leverage your organization's existing authentication methods, such as LDAP or Kerberos. See Authentication for a description of these different methods of authenticating Kubernetes users.
- ***Authorization***: When you set out to authorize your regular users, you will probably choose between RBAC and ABAC authorization. See Authorization Overview to review different modes for authorizing user accounts (as well as service account access to your cluster):
    - *Role-based access control* (RBAC): Lets you assign access to your cluster by allowing specific sets of permissions to authenticated users. Permissions can be assigned for a specific namespace (Role) or across the entire cluster (ClusterRole). Then using RoleBindings and ClusterRoleBindings, those permissions can be attached to particular users.
    - *Attribute-based access control* (ABAC): Lets you create policies based on resource attributes in the cluster and will allow or deny access based on those attributes. Each line of a policy file identifies versioning properties (apiVersion and kind) and a map of spec properties to match the subject (user or group), resource property, non-resource property (/version or /apis), and readonly. See Examples for details.

As someone setting up authentication and authorization on your production Kubernetes cluster, here are some things to consider:

- ***Set the authorization mode***: When the Kubernetes API server (kube-apiserver) starts, the supported authentication modes must be set using the *--authorization-mode* flag. For example, that flag in the *kube-adminserver.yaml* file (in */etc/kubernetes/manifests*) could be set to Node,RBAC. This would allow Node and RBAC authorization for authenticated requests.

- *Create user certificates and role bindings (RBAC)*: If you are using RBAC authorization, users can create a CertificateSigningRequest (CSR) that can be signed by the cluster CA. Then you can bind Roles and ClusterRoles to each user. See Certificate Signing Requests for details.
- *Create policies that combine attributes (ABAC)*: If you are using ABAC authorization, you can assign combinations of attributes to form policies to authorize selected users or groups to access particular resources (such as a pod), namespace, or apiGroup. For more information, see Examples.
- *Consider Admission Controllers*: Additional forms of authorization for requests that can come in through the API server include Webhook Token Authentication. Webhooks and other special authorization types need to be enabled by adding Admission Controllers to the API server.

Set limits on workload resources

Demands from production workloads can cause pressure both inside and outside of the Kubernetes control plane. Consider these items when setting up for the needs of your cluster's workloads:

- ***Set namespace limits***: Set per-namespace quotas on things like memory and CPU. See Manage Memory, CPU, and API Resources for details. You can also set Hierarchical Namespaces for inheriting limits.
- ***Prepare for DNS demand***: If you expect workloads to massively scale up, your DNS service must be ready to scale up as well. See Autoscale the DNS service in a Cluster.
- ***Create additional service accounts***: User accounts determine what users can do on a cluster, while a service account defines pod access within a particular namespace. By default, a pod takes on the default service account from its namespace. See Managing Service Accounts for information on creating a new service account. For example, you might want to:
    - Add secrets that a pod could use to pull images from a particular container registry. See Configure Service Accounts for Pods for an example.
    - Assign RBAC permissions to a service account. See ServiceAccount permissions for details.

Use Terraform - infrastructure as a code Platform

Terraform provisions, updates, and destroys infrastructure resources such as physical machines, VMs, network switches, containers, and more.

Configurations are code written for Terraform, using the human readable HashiCorp Configuration Language (HCL) to describe the desired state of infrastructure resources.

Providers are the plugins that Terraform uses to manage those resources. Every supported service or infrastructure platform has a provider that defines which resources are available and performs API calls to manage those resources.

Modules are reusable Terraform configurations that can be called and configured by other configurations. Most modules manage a few closely related resources from a single provider.

The Terraform Registry makes it easy to use any provider or module. To use a provider or module from this registry, just add it to your configuration; when you run `terraform init`, Terraform will automatically download everything it needs.

Workspace:

- A workspace is used to deploy and manage logical sub-divisions of your infrastructure, such as networking, database, K8 clusters, application instances etc.
- Workspace state can be shared to enable the use of shared infrastructure.
- This is where a Terraform run is performed and a state file (and its history) are stored.
- Variables (Terraform and shell) are assigned values to be passed into a run including provider credentials, possibly set at higher scopes.
- A workspace is where all objects related to Terraform managed resource are stored and managed. This is where you can find state, update the Terraform version, manage variables, see run history and much more.
- Workspaces can be as simple as a single resource or can be a complex monolithic configuration, it all depends on your use case and the directories that the workspace is linked to.
- A workspace can be integrated with your existing workflow whether
    - GitOps approach through a VCS provider - workspaces can be linked to Terraform configurations held in VCS repositories in order to automate pull request (PR) checks and deployments.
    - Using the native Terraform CLI, or
    - Deploying modules directly in the UI. - created by selecting a module from the private module registry. This is a good option for users who prefer a UI and Terraform resources that need to be consistently deployed using versioned modules that are made available by the account administrators. A good example of this is the deployment of a Kubernetes cluster that might have strict standards.

      There are three basic steps to get started with a module registry driven workspace:

        - Ensure a deployable module is available
        - Create the workspace
        - Set the variables (if necessary)

All workspace types follow the same pipeline.

Environment:

- A logical collection of workspaces that are typically used by the same team(s), belong to the same application/function and may have common provider credentials , governance policies, and module registry.
- In an environment you can define VCS provider connections and register modules that will be used across the workspaces.

Best Practices for Environment

- *Ensure environments (dev/UAT/stage/prod) have a proper level of separation having:*

  - Different system accounts are used for Terraform in these environments. Each Terraform system account has permissions only for its own environment.
  - Network connectivity is limited between resources across different environments.
  - (Optional) Only a designated agent or set of agents configured in a special virtual network is permitted to modify the infrastructure (i.e. run Terraform) and access sensitive resources (e.g. Terraform backend, key vaults etc). It is not possible to release to e.g. prod using a non-prod build agent.

- Ensure that Terraform configuration is as similar as possible between environments. (I.e. cannot forget about the whole module in PROD as compared to UAT)
- Terraform backend in higher environments (e.g. UAT) is not accessible from local machines (network + RBAC limitation). It can be accessed only from build machines and optionally from designated bastion hosts

System accounts for Terraform *- account is the administration area*

- In an account you
    - Manage team and user access to environments with RBAC.
    - Define and configure governance policies for all environments.
    - Register modules that are shared by all environments.
    - Optionally define provider credentials for each environment (published via shell variables).

- Different system accounts are used for:
    - Terraform
    - Kubernetes
    - Runtime application components
- Best Practices:

1) System accounts that are permitted to Terraform changes can be used only in designated CD pipelines.
2) It is not possible that I can use e.g. a production Terraform system account in a newly created pipeline without a special permission.
3) Access to use the Terraform system account is granted „just-in-time" for the release. Alternatively, the system account is granted permissions only for the time of deployment.
4) System accounts in higher environments have permissions limited to only what is required in order to perform actions.
5) Limit permissions to only the types of resources that are used.
6) Remove permissions for deleting critical resources (e.g. databases, storage) to avoid automated re-creation of these resources and losing data. On such occasions, special permissions should be granted „just-in-time".

Use cases for cluster deployment on AWS and Azure:

A: Using Terraform Scripts built by internal PSL Team

A1. For Minimal Viable Kubernetes Cluster:

1. Minimal Module - Simple Kubernetes deployment AWS – *completed*.
2. Complete Module - Modular based Kubernetes deployment on AWS – in-progress
3. Minimal Module - Simple Kubernetes deployment Azure - *Not started*
4. Complete Module - Modular based Kubernetes deployment on Azure - *Not started*

A2. For highly available (HA) Kubernetes cluster - With stacked control plane nodes

1. Minimal Module - Simple Kubernetes deployment AWS – *Not started*.
2. Complete Module - Modular based Kubernetes deployment on AWS – *Not started*
3. Minimal Module - Simple Kubernetes deployment Azure - *Not started*
4. Complete Module - Modular based Kubernetes deployment on Azure - *Not started*

A3. For highly available (HA) Kubernetes cluster - With external etcd nodes

1. Minimal Module - Simple Kubernetes deployment AWS – *Not started*.
2. Complete Module - Modular based Kubernetes deployment on AWS – *Not started*
3. Minimal Module - Simple Kubernetes deployment Azure - *Not started*
4. Complete Module - Modular based Kubernetes deployment on Azure - *Not started*

A4. For Managed Kubernetes Clusters on Cloud such as EKS and AKS

1. Minimal Module - Managed Kubernetes clusters on EKS – *Not started*.
2. Complete Module - Managed Kubernetes clusters on EKS – *Not started*
3. Minimal Module - Managed Kubernetes clusters on AKS - *Not started*
4. Complete Module - Managed Kubernetes clusters on AKS - *Not started*

B: Using Terraform Public Modules available from Terraform Repository

B1. Managed Kubernetes clusters on EKS
B2. Managed Kubernetes clusters on AKS