Contents

[Current Solution 2](#_Toc23408119)

[Agreed Solution 3](#_Toc23408120)

[What is a container 3](#_Toc23408121)

[Why Is Containerization of Applications Important for Deployment? 3](#_Toc23408122)

[So, what is Docker? 4](#_Toc23408123)

[Terminologies Docker 4](#_Toc23408124)

[Docker Images 4](#_Toc23408125)

[Dockerfile 5](#_Toc23408126)

[Build 5](#_Toc23408127)

[Docker Container 5](#_Toc23408128)

[Volumes 5](#_Toc23408129)

[Tag 5](#_Toc23408130)

[Docker Registry 5](#_Toc23408131)

[Compose 5](#_Toc23408132)

[Orchestrator 5](#_Toc23408133)

[Container Orchestration System 6](#_Toc23408134)

[Kubernetes 6](#_Toc23408135)

[Kubernetes Terminologies 7](#_Toc23408136)

[Kubernetes Pods 7](#_Toc23408137)

[Kubernetes Containers 7](#_Toc23408138)

[Kubernetes Nodes 7](#_Toc23408139)

[Kubernetes Deployments 7](#_Toc23408140)

[Kubernetes Ingress 7](#_Toc23408141)

[Our Kubernetes Architecture - Master & Worker (Node) 8](#_Toc23408142)

[Kubernetes Components Overview 9](#_Toc23408143)

[Scalability & Availability 9](#_Toc23408144)

[Scalability & Availability - Node Plane 9](#_Toc23408145)

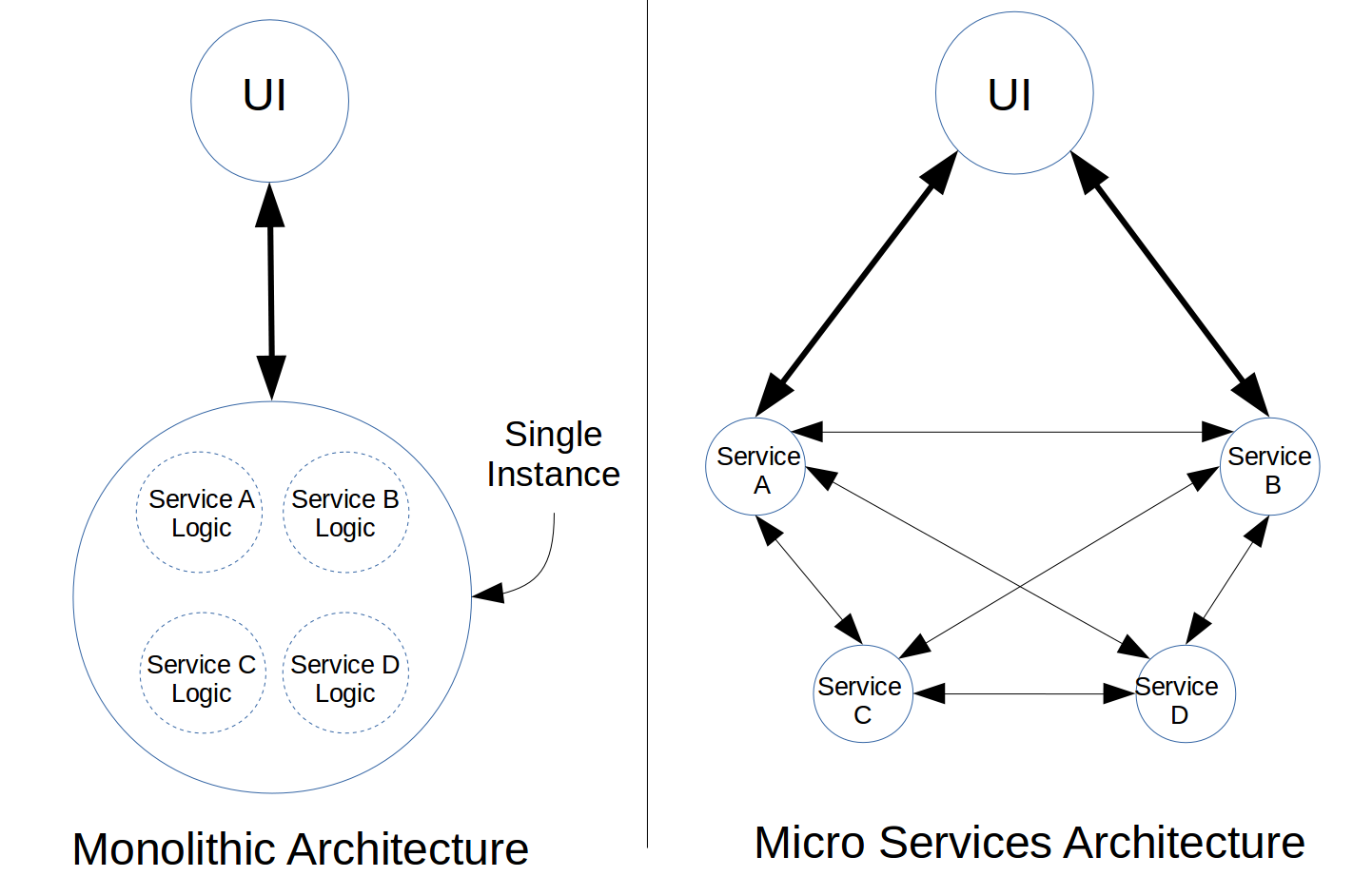
[Scalability & Availability - Control Plane 10](#_Toc23408146)

[Anatomy - HA Cluster 10](#_Toc23408147)

# Current Solution

Our current solution today is a monolith application (composed all in one piece). The current software application has different components combined into a single program from a single platform. This approach has drawbacks such as:

* The application is too large and complex to understand entirely, it is challenging to make changes quickly and correctly.
* The size of the application slows down the start-up time.
* Need to redeploy the entire application on each update.
* Challenging to scale when different modules have conflicting resource requirements.
* Bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug could impact the entire application.
* Difficulty in adopting new and advanced technologies. Since changes in languages or frameworks affect an entire application, it requires efforts to thoroughly work with the app details hence, it is costly considering both time and effort.



# Agreed Solution

The Agreed Solution is an approach to application development built as a suite of modular services (i.e. loosely coupled modules/components). Each module supports a specific business goal and uses a simple, well-defined interface to communicate with other sets of services.

Instead of sharing a single database as in Monolithic application, each microservice has its own database. Having a database per service is essential if we want to benefit from microservices, because it ensures loose coupling. Moreover, a service can use a type of database that is best suited to its needs.

There comes eventually a point where we decide for a scalable and robust architecture in terms of microservices. Furthermore, to fully exploit the advantages such as auto-scaling and cost-efficiency, a cloud-native application architecture is a prerequisite. The solution to handle the deployment of distributed applications on the cloud, was Containerization. Containerization is the practice of deploying applications by packaging them in a container.

# What is a container

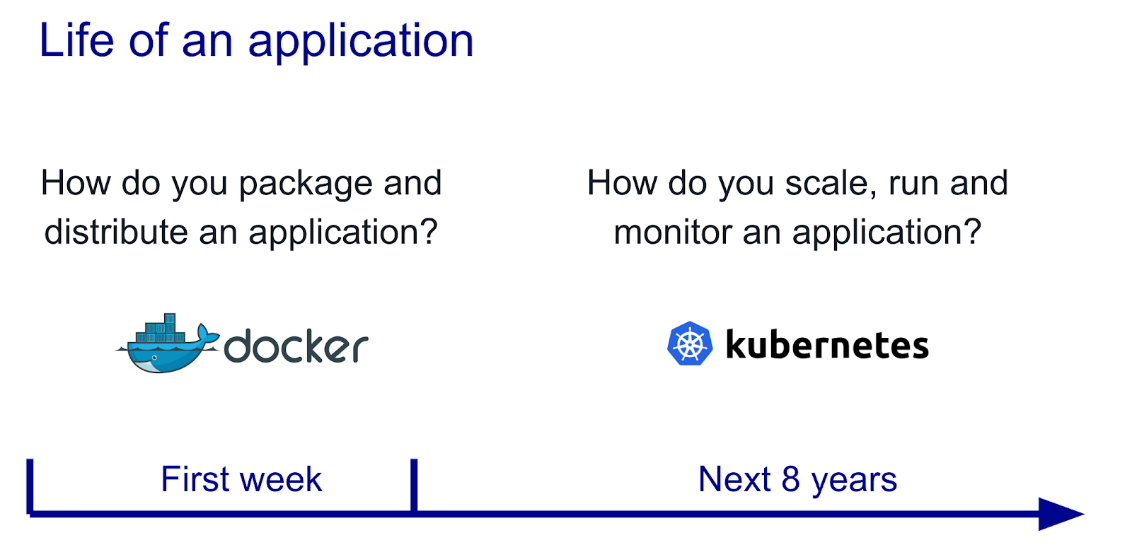
A container is an OS-level virtualization, in which the kernel allows the existence of multiple isolated user-space instances. In simpler terms, you could say that a container is a fully functional virtual computer, running inside another computer (host). Each container is isolated from other containers, and from the host. Containers have their own file systems. They can’t see each other's processes, and their computational resource usage can be bounded.

# Why Is Containerization of Applications Important for Deployment?

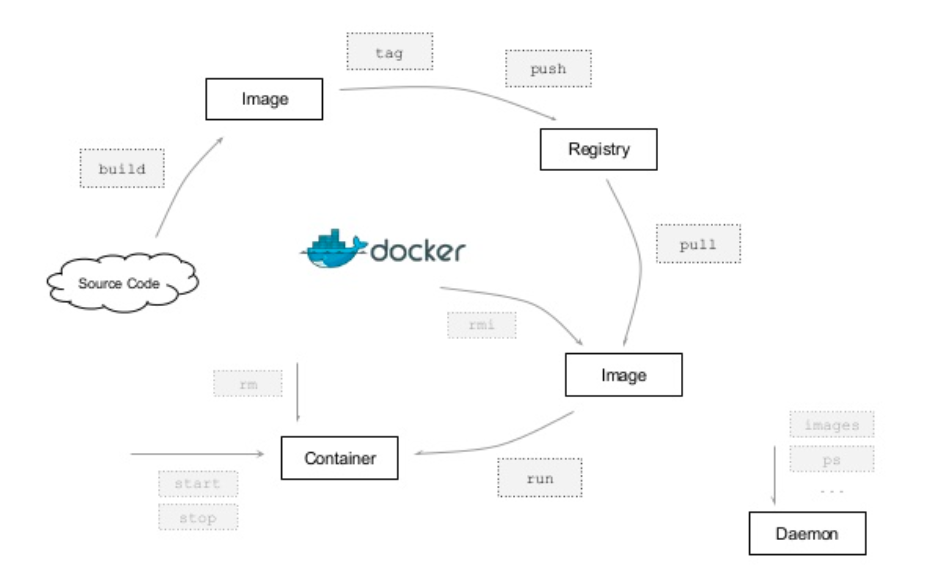
* **Decoupling of application and infrastructure**. Traditionally, software applications were deployed directly onto the host machine. This had the disadvantage of entangling the application’s executables, configuration, libraries, and lifecycles with each other and with the host OS. But this new approach to package each application and its dependencies into a self-sufficient isolated container, makes the application completely decoupled from the host OS and infrastructure. It allows for the deployment of an application on any host, without first worrying if the host has all the dependencies installed.
* **Isolation of resources**. Aside from decoupling an application from the host, an even more important advantage with containerization is the isolation of resources. If multiple containers are running on a host, a fatal error inside one container can’t affect other containers or, the host.
  + Compared to this, when all the applications are directly installed on the host, a fatal error can bring down, or corrupt, other applications and in some cases could crash the entire host. But there is no scope for such a scenario if the applications are deployed in containers. Containerization of applications adds great value and will simplify a great deal of complexity involved in developing, deploying, and maintaining distributed applications.
  + It wasn’t long before various engineering teams across the industry realized the advantages with containerization of applications and decided that every application that gets deployed within their organization must be containerized.

# So, what is Docker?

When most people talk about Docker they are talking about Docker Engine, the runtime that allows you to build and run containers. But before you can run a Docker container they must be built, starting with a Docker File. The Docker File defines everything needed to run the image including the OS network specifications, and file locations. Now that you have a Docker file, you can build a Docker Image which is the portable, static component that gets run on the Docker Engine. And if you don’t want to start from scratch Docker even has a service called Docker Hub, where you can store and share images.



# Terminologies Docker



## Docker Images

A package with all the dependencies and information needed to create a container. An image includes all the dependencies (such as frameworks) plus deployment and execution configuration to be used by a container runtime. Usually, an image derives from multiple base images that are layers stacked on top of each other to form the container's filesystem. An image is immutable once it has been created. Images are created from a Dockerfile with the docker build command.

## Dockerfile

A text file that contains instructions for how to build a Docker image. It's like a batch script, the first line states the base image to begin with and then follow the instructions to install required programs, copy files and so on, until you get the working environment you need.

## Build

The action of building a container image based on the information and context provided by its Dockerfile, plus additional files in the folder where the image is built.

## Docker Container

Containers are created from images with the docker run command i.e. an instance of a Docker image. A container represents the execution of a single application, process, or service. It consists of the contents of a Docker image, an execution environment, and a standard set of instructions.

## Volumes

Volumes offer a writable filesystem that the container can use. Since images are read-only but most programs need to write to the filesystem, volumes add a writable layer on top of the container image so the programs have access to a writable filesystem. The program doesn't know it is accessing a layered filesystem, it is just the filesystem as usual. Volumes live in the host system and are managed by Docker.

## Tag

A mark or label you can apply to images so that different images or versions of the same image (depending on the version number or the target environment) can be identified.

## Docker Registry

Images are stored in a Docker registry, such as Docker Hub and can be downloaded with the docker pull command

## Compose

A command-line tool and YAML file format with metadata for defining and running multi-container applications. You define a single application based on multiple images with one or more .yml files that can override values depending on the environment. After you have created the definitions, you can deploy the whole multi-container application with a single command (docker-compose up) that creates a container per image on the Docker host.

Orchestrator

A tool that simplifies management of clusters and Docker hosts. Orchestrators enable you to manage their images, containers, and hosts through a command line interface (CLI) or a graphical UI. You can manage container networking, configurations, load balancing, service discovery, high availability, Docker host configuration, and more. Typically, orchestrator products are the same products that provide cluster infrastructure, like Kubernetes and Azure Service Fabric, among other offerings in the market.

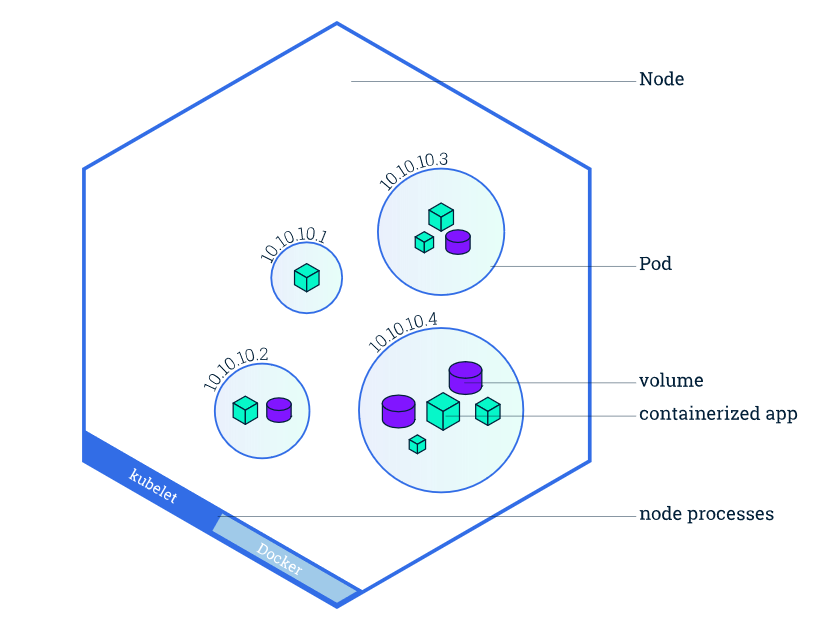
# Container Orchestration System

As soon as the industry embraced containerization, the need arose for the container orchestration system. On a high level, a container orchestration system allows the user to effectively manage the deployment of containerized applications. A robust and well-engineered container orchestration system could make the system resilient, efficient, and completely abstract the complexity of deployment operations from the user.

# Kubernetes

Kubernetes (K8s) is an open source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. Kubernetes is a container orchestration system developed by Google and is designed to effectively solve many of the issues we outlined earlier.

* Kubernetes can run containerized applications of any scale without any downtime.
* Kubernetes can self-heal containerized applications, making them resilient to unexpected failures.
* Kubernetes can auto-scale containerized applications as per the workload and ensure optimal utilization of cloud resources.
* Kubernetes greatly simplifies the process of deployment operations. With Kubernetes, however complex an operation is, it could be performed reliably by executing a couple of commands, at most.



# Kubernetes Terminologies

## Kubernetes Pods

When you created a deployment, Kubernetes created a Pod to host your application instance. A Pod is a Kubernetes abstraction that represents a group of one or more application containers, and some shared resources for those containers. Those resources may include:

* Shared storage, as Volumes
* Networking, as a unique cluster IP address
* Information about how to run each container, such as the container image version or specific ports to use

## Kubernetes Containers

Programs running on Kubernetes are packaged as containers. Containers are a widely accepted standard, so there are already many pre-built images that can be deployed on Kubernetes.

Containerization allows you to create self-contained execution environments. Any program and all its dependencies can be bundled up into a single file and then shared on the internet. Anyone can download the container and deploy it on their infrastructure with very little setup required.

## Kubernetes Nodes

A Pod always runs on a Node. A Node is a worker machine in Kubernetes and may be either a virtual or a physical machine, depending on the cluster. Each Node is managed by the Master. A Node can have multiple pods, and the Kubernetes master automatically handles scheduling the pods across the Nodes in the cluster.

## Kubernetes Deployments

Although pods are the basic unit of computation in Kubernetes, they are not typically directly launched on a cluster. Instead, pods are usually managed by one more layer of abstraction: the deployment.

A deployment’s primary purpose is to declare how many replicas of a pod should be running at a time. When a deployment is added to the cluster, it will automatically spin up the requested number of pods, and then monitor them. If a pod dies, the deployment will automatically re-create it.

## Kubernetes Ingress

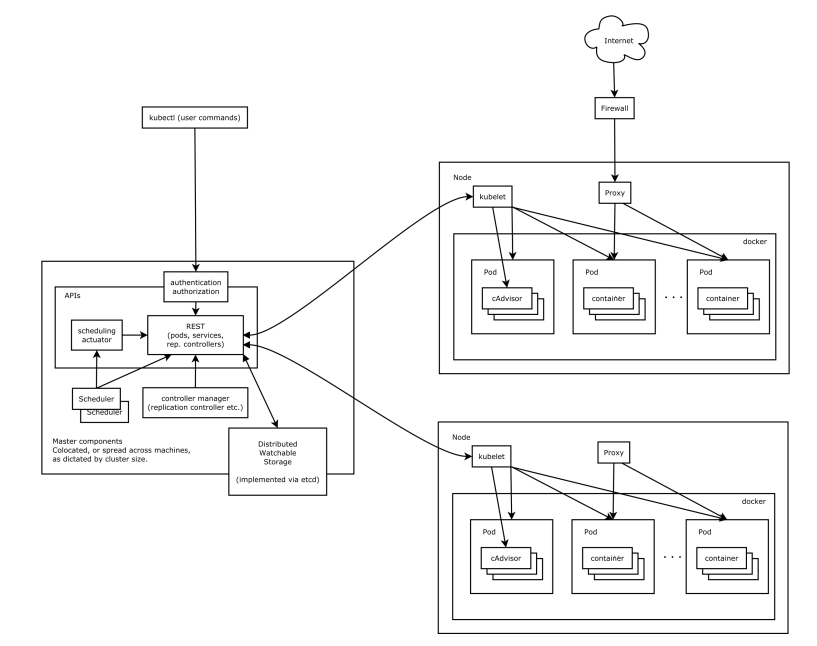
If you want to communicate with a service running in a pod, you must open up a channel for communication. This is referred to as ingress.

There are multiple ways to add ingress to your cluster. The most common ways are by adding either an Ingress controller, or a LoadBalancer.

# Our Kubernetes Architecture - Master & Worker (Node)

Kubernetes is really a master-slave type of architecture with certain components (master components) calling the shots in the cluster, and other components (node components) executing application workloads (containers) as decided by the master components.

In this system architecture diagram, services can be divided into services running on worker nodes and services constituting cluster-level control nodes. Kubernetes nodes have the necessary services to run application containers, which are controlled by master’s. Each node runs docker, which is responsible for all specific image downloads and container runs.



# Kubernetes Components Overview

Kubernetes consists of the following core components:

* etcd saves the state of the whole cluster.
* apiserver provides the only access to resource operation, and provides authentication, authorization, access control, API registration and discovery mechanisms.
* controller manager is responsible for maintaining the status of the cluster, such as fault detection, automatic expansion, rolling updates, etc.
* Scheduler is responsible for resource scheduling, scheduling Pod to the corresponding machine according to the scheduled scheduling strategy.
* kubelet is responsible for maintaining the life cycle of containers, as well as managing Volume (CVI) and Network (CNI).
* Container runtime is responsible for image management and the real operation of Pod and container (CRI);
* kube-proxy is responsible for providing service discovery and load balancing within the cluster for Service.

In addition to core components, there are also some recommended components:

* kube-dns is responsible for providing DNS services for the entire cluster
* Ingress Controller Provides Extranet Access for Services
* Heapster provides resource monitoring
* Dashboard provides GUI
* Federation provides clusters across available zones
* Fluentd-elastic search provides cluster log collection, storage and query

# Scalability & Availability

A common requirement is for a Kubernetes cluster to both scale to accommodate increasing workloads and to be fault-tolerant, remaining available even in the presence of failures (datacenter outages, machine failures, network partitions).

## Scalability & Availability - Node Plane

A cluster can be scaled by adding worker nodes, which increases the workload capacity of the cluster, giving Kubernetes more room to schedule containers. Kubernetes is self-healing in that it keeps track of its nodes and, when a node is deemed missing (no longer passing heartbeat status messages to the master), the control plane is clever enough to re-schedule the pods from the missing node onto other (still reachable) nodes. Adding more nodes to the cluster therefore also makes the cluster more fault-tolerant, as it gives Kubernetes more freedom in rescheduling pods from failed nodes onto new nodes.

Adding nodes to a cluster is commonly carried out manually when a cluster administrator detects that the cluster is heavily loaded or cannot fit additional pods. Monitoring and managing the cluster size manually is tedious. Using autoscaling, this task can be automated, the Kubernetes cluster-autoscaler is one such solution.

## Scalability & Availability - Control Plane

Adding more workers does not make the cluster resilient to all sorts of failures. For example, if the master API server goes down (for example, due to its machine failing or a network partition cutting it off from the rest of the cluster) it will no longer be possible to control the cluster (via kubectl).

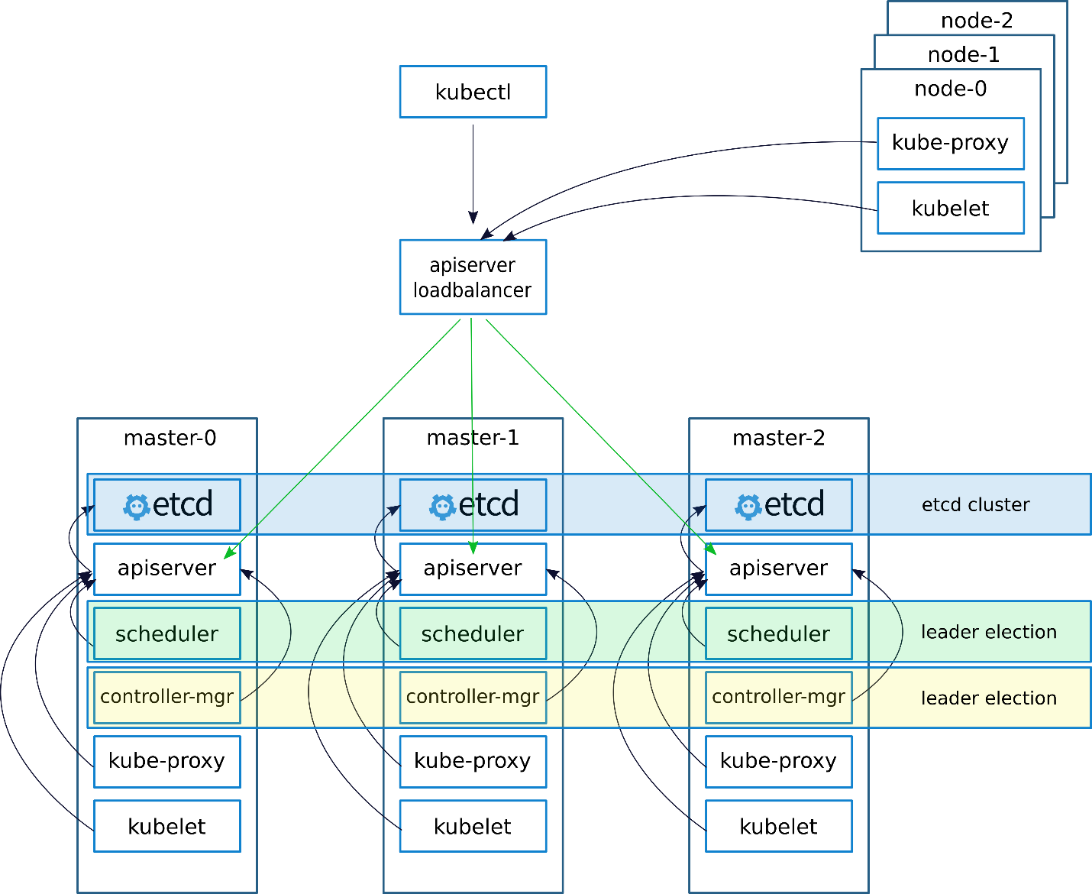
For a true highly available cluster, we also need to replicate the control plane components. Such a control plane can remain reachable and functional even in the face of failures of one or a few nodes, depending on the replication factor.

A HA control plane setup requires at least three masters to withstand the loss of one master, since etcd needs to be able to form a quorum (a majority) to continue operating.

# Anatomy - HA Cluster

* We need a replicated, distributed etcd storage layer.
* We need to replicate the apiserver across several machines and front them with a load-balancer.
* We need to replicate the controllers and schedulers and set them up for leader-election.
* We need to configure the (worker) nodes’ kubelet and kube-proxy to access the apiserver through the load-balancer.

With three sets of master components the cluster can tolerate a failure of one master node since, in that case, etcd requires two live members to be able to form a quorum (a node majority) and continue working.



Some further considerations:

* etcd replicates the cluster state to all master nodes. Therefore, to lose all data, all three nodes must experience simultaneous disk failures. Although unlikely, one may want to make the storage layer even more reliable by using a separate disk that is decoupled from the lifecycle of the machine/VM (for example, an AWS EBS volume). You can also use a RAID setup to mirror disks, and finally set up etcd to take periodical backups.
* The etcd replicas can be placed on separate, dedicated machines to isolate them and give them dedicated machine resources for improved performance.
* The load-balancer must monitor the health of its apiservers and only forward traffic to live servers.
* Spread masters across data centers to increase the overall uptime of the cluster. If the masters are placed in different zones, the cluster can tolerate the outage of an entire availability zone.
* (Worker) node kubelets and kube-proxys must access the apiserver via the load-balancer to not tie them to a particular master instance.
* The loadbalancer must not become a single point of failure. Most cloud providers can offer fault-tolerant loadbalancer services. For on-premise setups, one can make use of an active/passive nginx/HAProxy setup with a virtual/floating IP that is re-assigned by keep alived when failover is required.