Building Scalable Applications using Docker and Kubernetes

ΓΑΛΑΝΗΣ ΑΧΙΛΛΕΑΣ ΑΛΕΞΑΝΔΡΟΣ ΒΑΣΙΛΕΙΟΣ - 02941 and ΓΑΛΑΝΗΣ ΚΩΝΣΤΑΝΤΙΝΟΣ ΟΡΕΣΤΗΣ ΒΑΣΙΛΕΙΟΣ - 03074

Ειδικό Θέμα 23-24
Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Πανεπιστήμιο Θεσσαλίας, Βόλος
{acgalanis, kogalanis}@e-ce.uth.gr

Περιεχόμενα

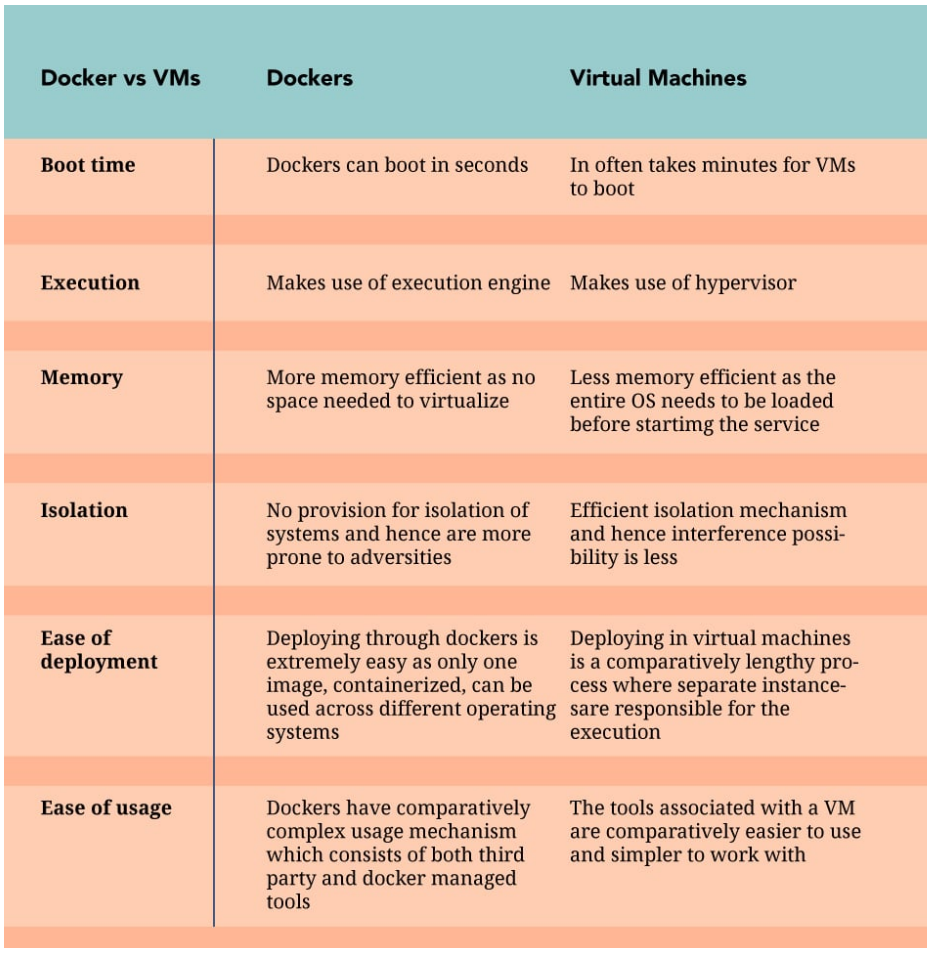
1 What is Docker

– **Docker Overview**

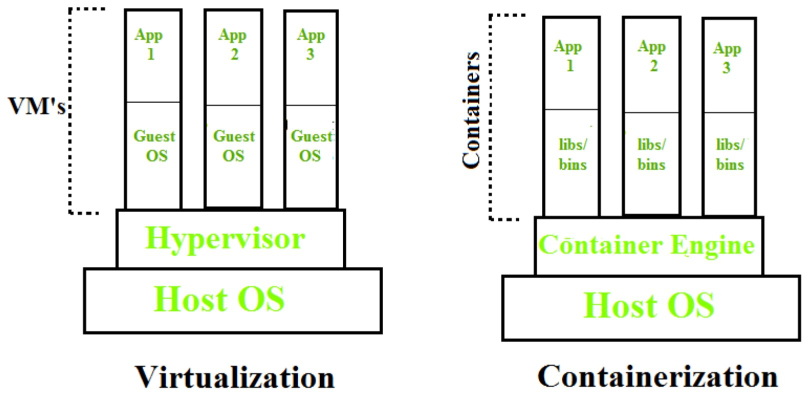
Docker is a containerization platform, written in Go that packages your application and all its dependencies together in the form of a docker container. It's a set of platform-as-a-service products designed to solve the many challenges created by the growing DevOps trend. DevOps is a blend of development and operations practices aimed at unifying software development (Dev) and software operation (Ops), focusing on collaboration, automation, and continuous integration/delivery to improve the speed and quality of software deployment. By utilizing **containers**, Docker simplifies the process of developing, deploying, and operating applications.

– **Containers**

Containers are a key reason why Docker is so attractive to contemporary developers. While the current industry standard involves using Virtual Machines (VMs) for running software applications, VMs operate within a guest Operating System on virtual hardware, supported by the server's host OS. This method, however, incurs a significant computational overhead due to the virtualization of hardware for the guest OS.
In contrast, containers adopt a different approach. They establish an abstraction at the application layer, encapsulating your application and its dependencies, including the operating system, application code, runtime, system tools, and system libraries. These containers operate on the shared operating system kernel of the host machine, allowing one or more processes to run within each container. This approach eliminates the need for pre-allocating RAM, as memory is dynamically allocated during container creation, unlike VMs which require preset memory allocation.
Containerization offers superior resource utilization and quicker boot-up times compared to VMs, marking a significant advancement in virtualization technology. Containers provide nearly the same level of isolation as virtual machines but require only a fraction of the computing power.

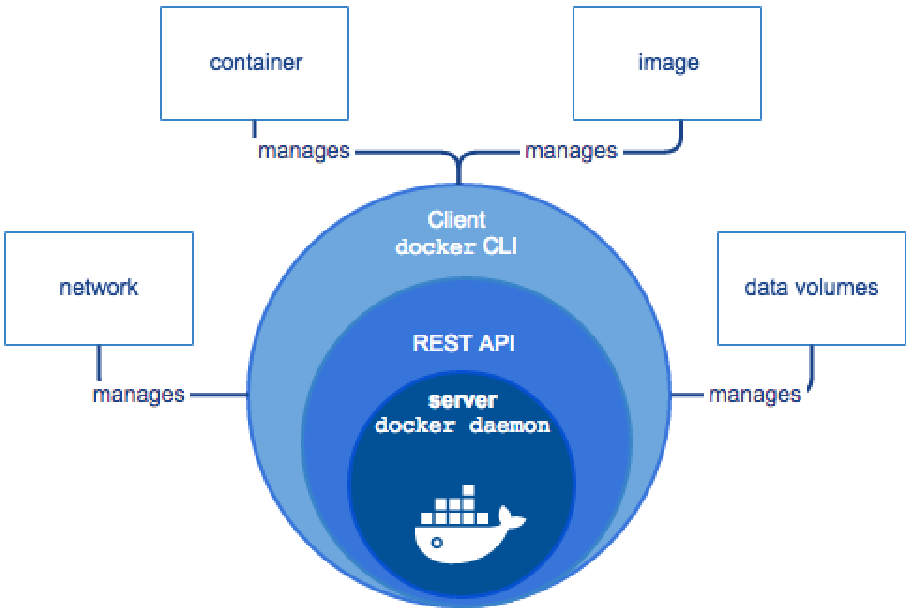


Docker vs VMs	Dockers	Virtual Machines
Boot time	Dockers can boot in seconds	In often takes minutes for VMs to boot
Execution	Makes use of execution engine	Makes use of hypervisor
Memory	More memory efficient as no space needed to virtualize	Less memory efficient as the entire OS needs to be loaded before startimg the service
Isolation	No provision for isolation of systems and hence are more prone to adversities	Efficient isolation mechanism and hence interference possi-bility is less
Ease of deployment	Deploying through dockers is extremely easy as only one image, containerized, can be used across different operating systems	Deploying in virtual machines is a comparatively lengthy pro-cess where separate instance-sare responsible for the execution
Ease of usage	Dockers have comparatively complex usage mechanism which consists of both third party and docker managed tools	The tools associated with a VM are comparatively easier to use and simpler to work with



– **Core Components of Docker**

The Core of the Docker is consists of: **Docker Engine, Docker Images, Docker Containers.**



- **Docker Engine** is one of the core components of Docker. It is responsible for the overall functioning of the Docker platform. It's a client-server based application and consists of 3 main components:

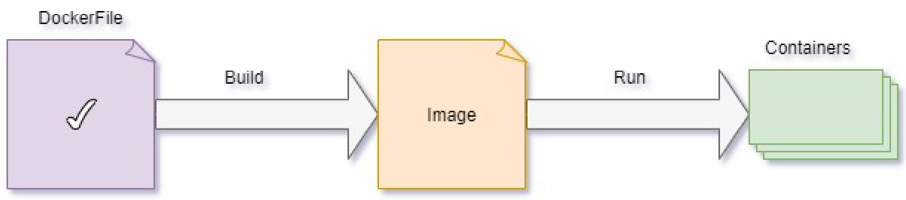
  ∗ **Server (Docker Daemon):** The Server runs a daemon known as dockerd (Docker Daemon), which is nothing but a process. It listens to only Docker API requests and it is responsible for creating and managing Docker Images, Containers, Networks and Volumes on the Docker platform. It also communicates with other daemons to manage Docker services.

  ∗ **REST API:** The REST API defines the methods by which applications can communicate with the Server, directing it to perform their required tasks.

  ∗ **Client:** The Client is nothing but a command line interface, that allows users to interact with Docker using the commands. When we use commands, the client sends these instructions to the Docker daemon (dockerd), which then carries them out. The command used by docker depend on Docker API. In Docker, the client can interact more than one daemon process.

- **Docker Images:** A Docker Image is a template that contains the application, and all the dependencies required to run that application on Docker. This image informs how a container should instantiate, determining which software components will run and how. Images are created using Dockerfiles. Dockerfiles use DSL (Domain Specific Language) and contains instructions for generating a Docker image. Dockerfiles will define the processes to quickly produce an image. While creating your application, you should create a Dockerfile in order, since the Docker daemon runs all of the instructions from top to bottom.

- **Docker Containers:** As stated earlier, a Docker Container is a logical entity. In more precise terms, it is a running instance of the Docker Image.



- **Docker Volumes:** Volumes are the directories or files that exist on the host filesystem and are mounted to the containers for persisting data generated or modified by them. They are stored in the part of the host filesystem managed specifically by Docker and it should not be modified by non-Docker processes. Volumes are the most preferred way to store container data as they provide efficient performance and are isolated from the other functionalities of the Docker host.

- **Docker Network:** A network is a group of two or more devices that can communicate with each other either physically or virtually. The Docker network is a virtual network created by Docker to enable communication between Docker containers. If two containers are running on the same host they can communicate with each other without the need for ports to be exposed to the host machine. You may use Docker to manage your Docker hosts in any platform manner, regardless of whether they run Windows, Linux, or a combination of the two. Docker supports several network types, each serving different use cases:

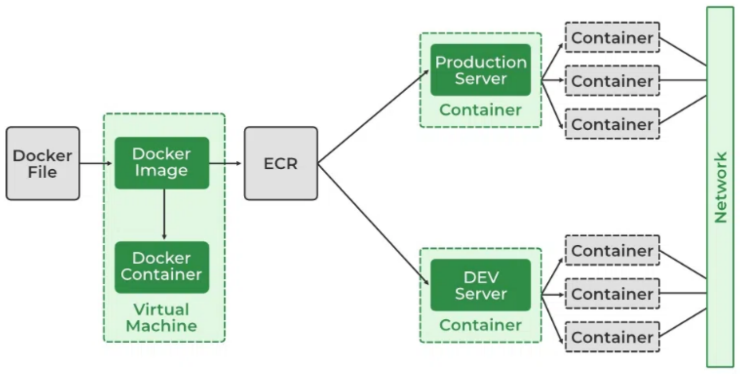
  ∗ **Bridge Network:** The default network mode when you run a container. It creates a private internal network on the host, and containers connected to this network get an internal IP address. Ideal for standalone containers that need to communicate.

  ∗ **Host Network:** Removes network isolation between the container and the Docker host. The container shares the host's networking namespace and is directly accessible on the host's network.

  ∗ **None Network:** Disables all networking for the container. Used when you want complete network isolation.

  ∗ **Overlay Network:** Enables networking among multiple Docker daemons and is ideal for Docker Swarm, supporting multi-host networking. (Docker Swarm is a container orchestration tool)

  ∗ **Custom Networks:** You can create custom networks using Docker's network drivers or third-party plugins to tailor network environments.



– **Docker Registries(Hub)**

A Docker registry is a system for storing and distributing Docker images with specific names. There may be several versions of the same image, each with its own set of tags. A Docker registry is separated into Docker repositories, each of which holds all image modifications. The registry may be used by Docker users to fetch images locally and to push new images to the registry (given adequate access permissions when applicable). The registry is a server-side application that stores and distributes Docker images. It is stateless and extremely scalable.

- **Key Aspects of Docker Registries**

  ∗ **Storage of Docker Images:** Registries store Docker images, which can include pre-configured operating systems, applications, and dependencies.

  ∗ **Image Versioning and Tagging:** They maintain different versions of images using tags, allowing users to track and roll back to specific versions as needed.

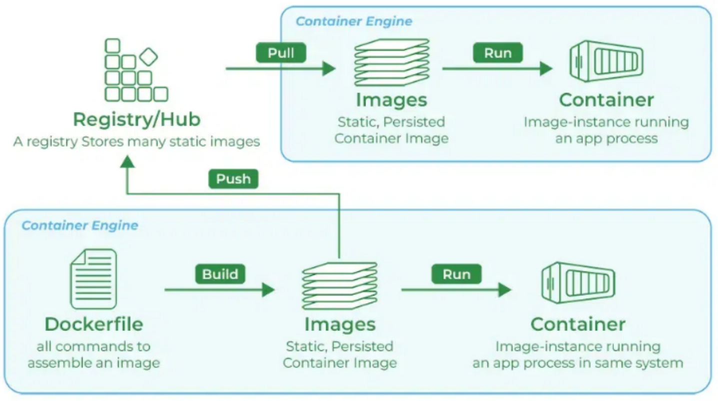
  ∗ **Public and Private Repositories:** Docker registries can host both public repositories, accessible by anyone, and private repositories for sensitive images that should be kept confidential.

- **Types of Docker Registries**

  ∗ **Docker Hub:** The default public registry for Docker images. It hosts a large number of official images from various software vendors and individual developers' images.

  ∗ **Private Registries:** Organizations can set up their private registries to store and manage images internally. This is often used for proprietary software or images containing sensitive information.

  ∗ **Third-Party Registries:** Other cloud providers and third-party services offer Docker registry services, such as AWS Elastic Container Registry (ECR), Google Container Registry (GCR), and Azure Container Registry (ACR).


Container Engine
Registry/Hub
A registry Stores many static images
Pull
Images
Static, Persisted Container Image
Run
Container
Image-instance running an app process
Push
Container Engine
Dockerfile
all commands to assemble an image
Build
Images
Static, Persisted Container Image
Run
Container
Image-instance running an app process in same system

– **Docker Editions and Subscriptions**

Docker offers different editions to cater to a wide range of users, from individual developers to large enterprises. Here's an overview of the available Docker editions:

- **Docker Community Edition (CE):** This is the open-source, free version of Docker, ideal for individual developers, small teams, and anyone looking to experiment with containerization. It offers a robust platform for creating and managing containers and provides access to support from the Docker community, as well as certified containers and plugins available on the Docker Store.

- **Docker Enterprise Edition (EE):** This enterprise-grade solution is tailored for organizations that demand advanced features, security, scalability, and support. It's available in three offerings:

  * **Docker EE Basic:** Includes the Docker Engine for certified infrastructure and support from Docker Inc., along with access to certified containers and plugins.

  * **Docker EE Standard:** Offers advanced image and container management capabilities, LDAP integration, and role-based access control (RBAC).

  * **Docker EE Enterprise:** The highest tier, offering enhanced security features (like vulnerability scanning and image signing), support for Kubernetes, and dedicated enterprise support.

Docker subscriptions offer a tailored range of services and features, catering to diverse needs from individual developers to large-scale enterprises, ensuring that every user can find the right tools for their containerization projects.

- **Docker Personal:** Ideal for open-source communities, individual developers, education, and small businesses. It includes unlimited public repositories, access tokens, collaborators for public repositories, and Docker Scout Free for software supply chain security.

- **Docker Pro:** Geared towards individual developers, offering more control over the development environment. It includes all features of Docker Personal plus unlimited private repositories, 5000 image pulls per day, auto builds with 5 concurrent builds, and 300 vulnerability scans.

- **Docker Team:** Offers collaboration, productivity, and security features for groups of developers. It includes everything in Docker Pro, plus unlimited teams, auto builds with 15 concurrent builds, unlimited vulnerability scanning, and advanced collaboration and management tools.

- **Docker Business:** Designed for enterprises using Docker at scale, offering centralized management and advanced security features. It includes everything in Docker Team, hardened Docker Desktop (enhanced version of Docker Desktop), image and registry access management, and tools for managing multiple organizations and settings.

  Each edition and subscription of Docker is designed to meet specific needs, ensuring a range of options for users of different levels and requirements. From individual development to large-scale enterprise deployments, Docker provides a suite of tools and services to build, ship, and run containerized applications efficiently.

– **System Requirements**

- **Windows**

  ∗ **WSL 2 backend**

    · WSL version 1.1.3.0 or later.

    · Windows 11 64-bit: Home or Pro version 21H2 or higher, or Enterprise or Education version 21H2 or higher.

    · Windows 10 64-bit: We recommend Home or Pro 22H2 (build 19045) or higher, or Enterprise or Education 22H2 (build 19045) or higher. Minimum required is Home or Pro 21H2 (build 19044) or higher, or Enterprise or Education 21H2 (build 19044) or higher.

    · Turn on the WSL 2 feature on Windows. For detailed instructions, refer to the Microsoft documentation.

    - The following hardware prerequisites are required to successfully run WSL 2 on Windows 10 or Windows 11: **1)** 64-bit processor with Second Level Address Translation (SLAT - Second Level Address Translation). **2)** 4GB system RAM. **3)** Enable hardware virtualization in BIOS. For more information, see Virtualization.

  - **Hyper-V backend and Windows containers**
    - Windows 11 64-bit: Pro version 21H2 or higher, or Enterprise or Education version 21H2 or higher.
    - Windows 10 64-bit: We recommend Home or Pro 22H2 (build 19045) or higher, or Enterprise or Education 22H2 (build 19045) or higher. Minimum required is Home or Pro 21H2 (build 19044) or higher, or Enterprise or Education 21H2 (build 19044) or higher. For Windows 10 and Windows 11 Home, see the system requirements in the WSL 2 backend tab.
    - Turn on Hyper-V and Containers Windows features.
    - The following hardware prerequisites are required to successfully run Client Hyper-V on Windows 10: **1)** 64-bit processor with Second Level Address Translation (SLAT - Second Level Address Translation). **2)** 4GB system RAM. **3)** Turn on BIOS-level hardware virtualization support in the BIOS settings. For more information, see Virtualization.

- **Linux**

  Docker provides .deb and .rpm packages from the following Linux distributions and architectures: Ubuntu, Fedora, Debian.

  - 64-bit kernel and CPU support for virtualization.
  - KVM virtualization support. Follow the KVM virtualization support instructions to check if the KVM kernel modules are enabled and how to provide access to the KVM device.
  - QEMU must be version 5.2 or later. We recommend upgrading to the latest version.
  - systemd init system.
  - Gnome, KDE, or MATE Desktop environment. For many Linux distros, the Gnome environment does not support tray icons. To add support for tray icons, you need to install a Gnome extension. For example, AppIndicator.

- ∗ At least 4 GB of RAM.
- ∗ Enable configuring ID mapping in user namespaces, see File sharing.
- ∗ Recommended: Initialize pass for credentials management.

- **Mac**
  - ∗ **Mac with Intel chip**
    - · A supported version of macOS.
    - · At least 4 GB of RAM.
  - ∗ **Mac with Apple silicon**
    - · A supported version of macOS.
    - · At least 4 GB of RAM.

      Beginning with Docker Desktop 4.3.0, we have removed the hard requirement to install Rosetta 2. There are a few optional command line tools that still require Rosetta 2 when using Darwin/AMD64. See Known issues. However, to get the best experience, we recommend that you install Rosetta 2. To install Rosetta 2 manually from the command line, run the following command:

For more information please refer to the Windows, Linux and Mac system requirements pages.

– **Installation and Instructional resources**

For the installation process on various operating systems, comprehensive guidance was sought from the official Docker documentation for Windows, Linux, and Mac. Instructional resources regarding the utilization of Docker are accessible via the following links: GeeksforGeeks, freeCodeCamp, and Docker Curriculum. Additionaly, we recommend watching a video tutorial provided here.

2 What is Kubernetes

– **Kubernetes Overview**

As the definition says, Kubernetes or k8s is an open-source orchestration and cluster management for container-based applications maintained by the Cloud Native Computing

Foundation. In simple words, Kubernetes helps to manage containerised applications in various types of physical, virtual, and cloud environments. Also, it makes the container deployment so easy using a declarative YAML file. You specify how you want the container to be deployed, and Kubernetes takes care of it by reading the information provided in the YAML. As per the state of Kubernetes report by Splunk 96% of organizations are either using or evaluating Kubernetes and 5.6 million developers are using kubernetes today. Also, there has been a more than 300% increase in container production usage in the past 5 years.

– **Kubernetes Features**

Kubernetes helps you to control the resource allocation and traffic management for cloud applications and microservices. It also helps to simplify various aspects of service-oriented infrastructures. Kubernetes allows you to assure where and when containerized applications run and helps you to find resources and tools you want to work with. Here are the essential Kubernetes features:

- **Automated Scheduling:** Automatically allocates resources and schedules containers based on their requirements and available resources, optimizing for efficiency.
- **Self-Healing Capabilities:** Automatically replaces or restarts containers that fail, do not respond, or do not meet user-defined health checks.
- **Automated Rollouts & Rollback:** Manages the deployment of new versions of applications and automatically rolls back to a previous stable version in case of failure.
- **Horizontal Scaling & Load Balancing:** Facilitates the scaling of applications up or down based on demand, and distributes network traffic to ensure stability.
- **Offers Environment Consistency for Development, Testing, and Production:** Ensures that applications run consistently across different development, testing, and production environments.
- **Infrastructure is Loosely Coupled:** Each component operates independently, which increases the overall system's resilience and flexibility.
- **Provides a Higher Density of Resource Utilization:** Efficiently maximizes resource use, reducing waste and lowering costs.
- **Offers Enterprise-Ready Features:** Includes features like security, governance, and scalability that are necessary for enterprise-level deployment.
- **Application-Centric Management:** Focuses on managing the application instead of individual machines, making it easier to deploy complex applications.

- **Auto-scalable Infrastructure:** Automatically adjusts the computing resources as needed, based on the workload requirements.

- **You Can Create Predictable Infrastructure:** Allows for the creation of a more stable and predictable operational environment for deploying applications.

– **Kubernetes Basics**

- **Cluster:** It is a collection of hosts(servers) that helps you to aggregate their available resources. That includes ram, CPU, ram, disk, and their devices into a usable pool.

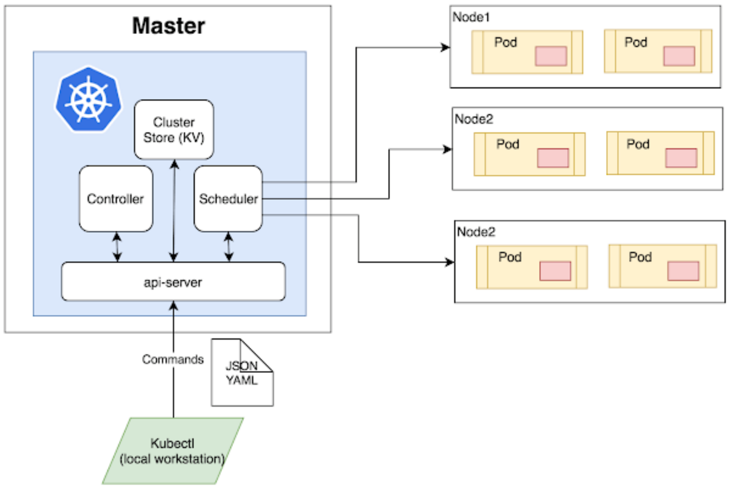
- **Master:** The master is a collection of components which make up the control panel of Kubernetes. These components are used for all cluster decisions. It includes both scheduling and responding to cluster events.

- **Node:** It is a single host which is capable of running on a physical or virtual machine. A node should run both kube-proxy, minikube, and kubelet which are considered as a part of the cluster.

- **Namespace:** It is a logical cluster or environment. It is a widely used method which is used for scoping access or dividing a cluster.

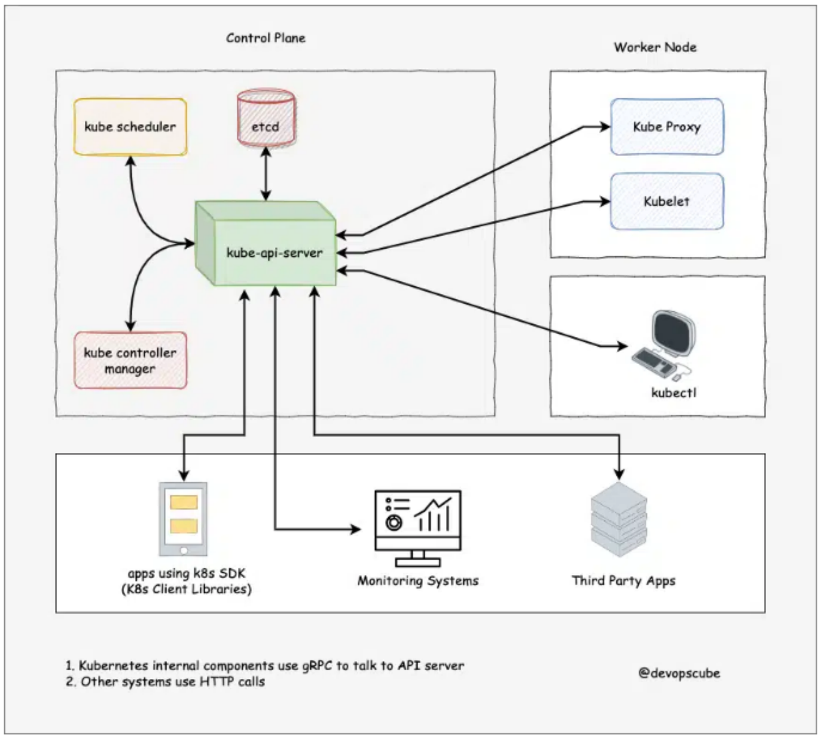
– **Kubernetes Architecture**

- **Control Plane (Master Node):** The master node is the first and most vital component which is responsible for the management of Kubernetes cluster. It is the entry point for all kind of administrative tasks. There might be more than one master node in the cluster to check for fault tolerance. The master node has various components like API Server, Controller Manager, Scheduler, and ETCD. Let see all of them:

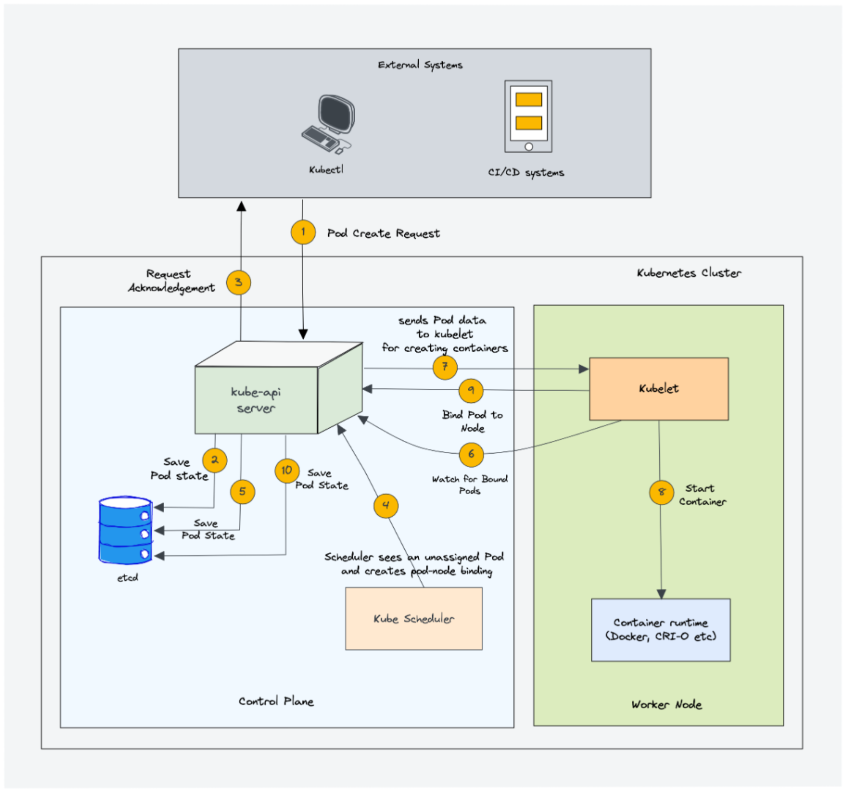


Kubernetes Architecture Diagram

* **API Server:** The API server acts as an entry point for all the REST commands used for controlling the cluster. So when you use kubectl to manage the cluster, at the backend you are actually communicating with the API server through HTTP REST APIs. However, the internal cluster components like the scheduler, controller, etc talk to the API server using gRPC.



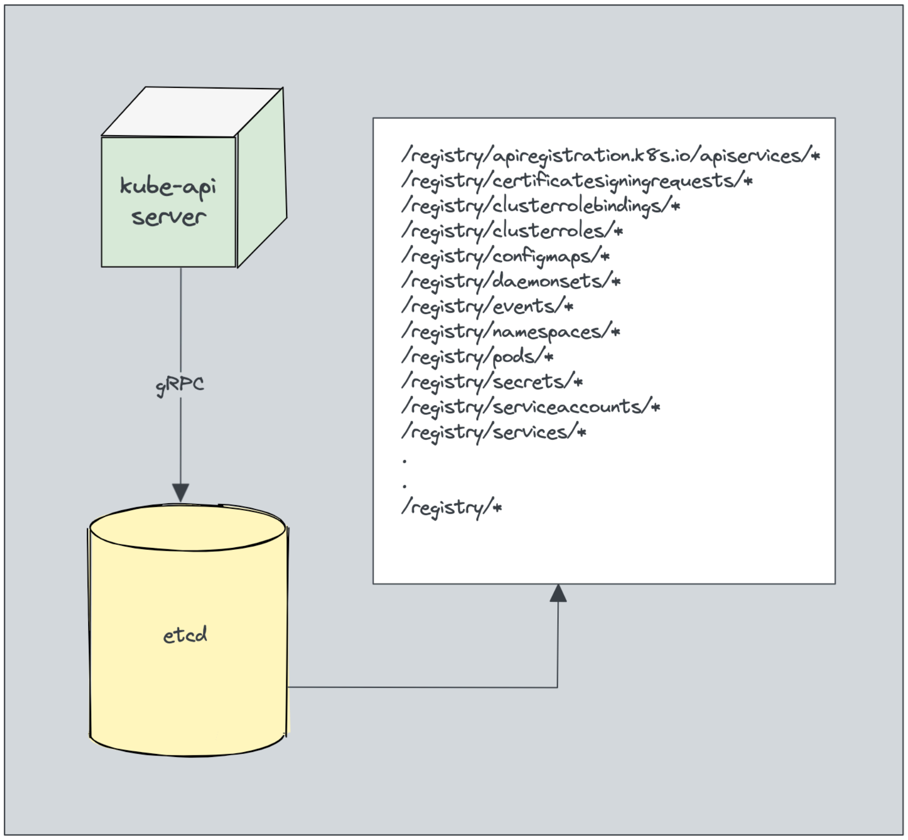
* **Scheduler:** The scheduler schedules the tasks to the slave node. It stores the resource usage information for every slave node. It is responsible for distributing the workload. It also helps you to track how the working load is used on cluster nodes. It helps you to place the workload on resources which are available and accept the workload. Here is how the scheduler works.



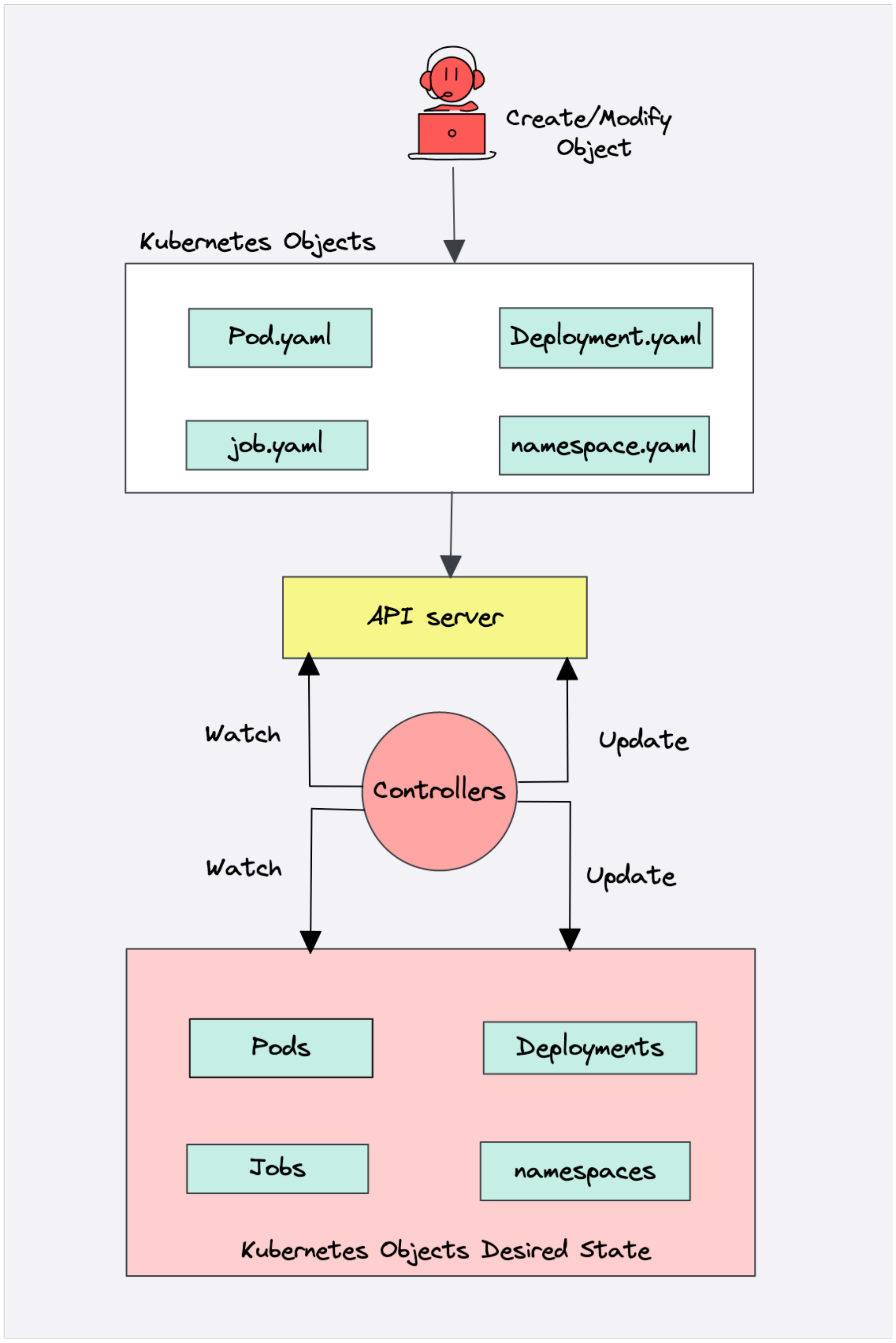
∗ **etcd:** Kubernetes is a distributed system and it needs an efficient distributed database like etcd that supports its distributed nature. It acts as both a backend service discovery and a database. You can call it the brain of the Kubernetes cluster. Etcd is an open-source strongly consistent, distributed key-value store:

- **Strongly consistent:** If an update is made to a node, strong consistency will ensure it gets updated to all the other nodes in the cluster immediately. Also if you look at CAP theorem, achieving 100% availability with strong consistency and & Partition Tolerance is impossible.
- **Distributed:** etcd is designed to run on multiple nodes as a cluster without sacrificing consistency.
- **Key Value Store:** A nonrelational database that stores data as keys and values. It also exposes a key-value API. The datastore is built on top of BboltDB which is a fork of BoltDB.

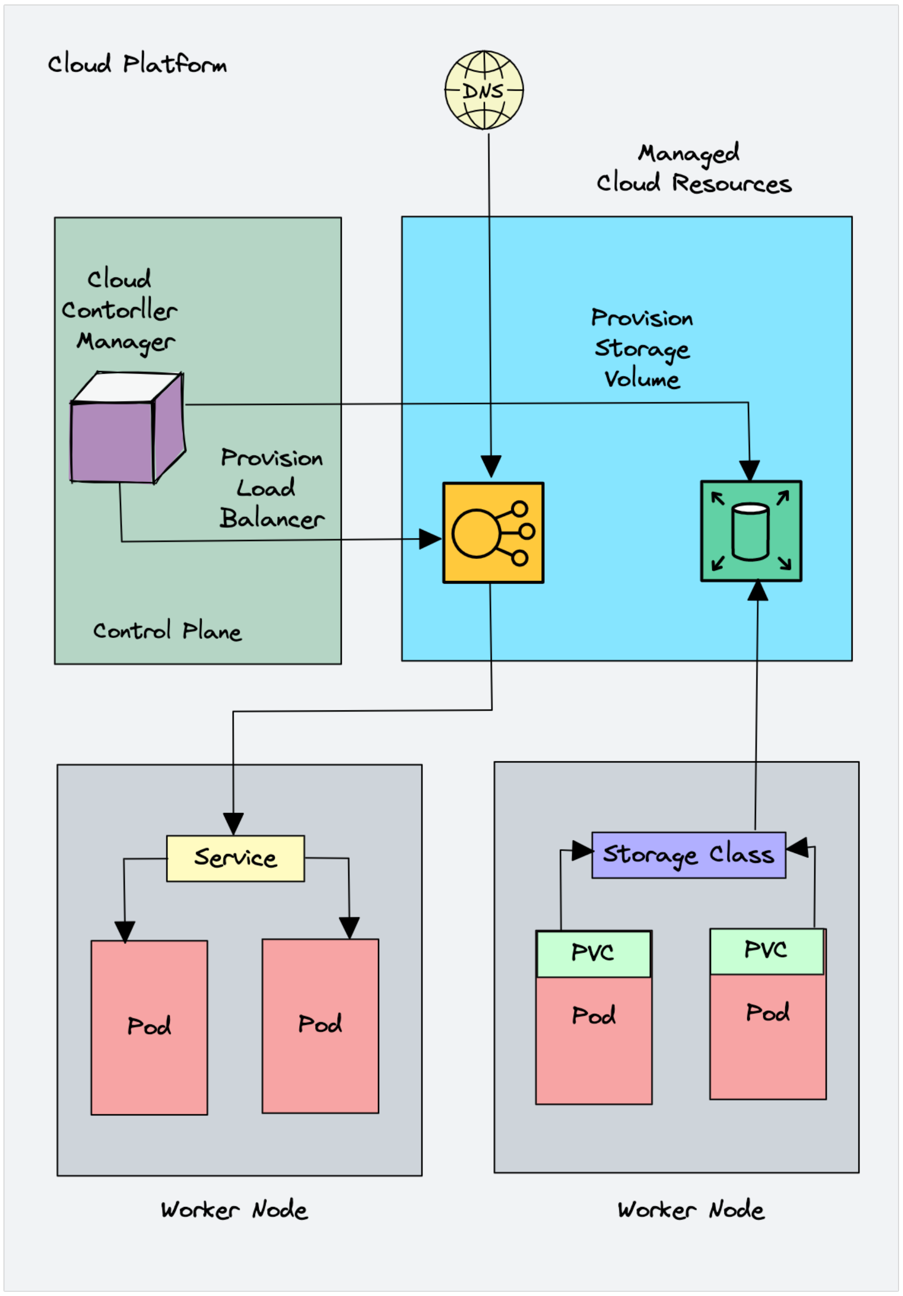
Etcd uses raft consensus algorithm for strong consistency and availability. It works in a leader-member fashion for high availability and to withstand node failures.



∗ **Controller Manager:** is a component that manages all the Kubernetes controllers. Kubernetes resources/objects like pods, namespaces, jobs, replicaset are managed by respective controllers. Also, the scheduler is also a controller managed by the controller manager. (Controllers are programs that run infinite control loops. Meaning it runs continuously and watches the actual and desired state of objects. If there is a difference in the actual and desired state, it ensures that the kubernetes resource/object is in the desired state).

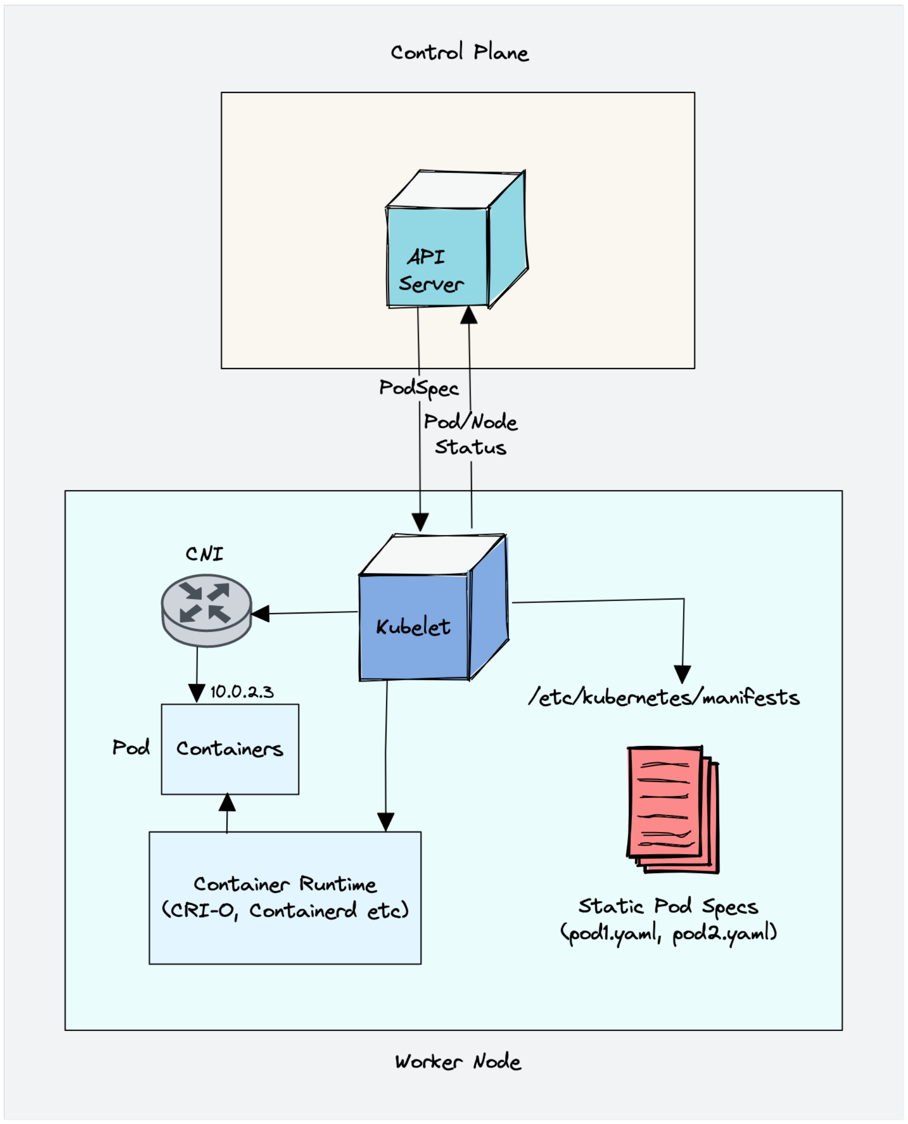

Create/Modify Object
Kubernetes Objects
Pod.yaml
Deployment.yaml
job.yaml
namespace.yaml
API server
Watch
Update
Controllers
Watch
Update
Pods
Deployments
Jobs
namespaces
Kubernetes Objects Desired State

* **Cloud Controller Manager:** When kubernetes is deployed in cloud environments, the cloud controller manager acts as a bridge between Cloud Platform APIs and the Kubernetes cluster. This way the core kubernetes core components can work independently and allow the cloud providers to integrate with kubernetes using plugins. (For example, an interface between kubernetes cluster and AWS cloud API) Cloud controller integration allows Kubernetes cluster to provision cloud resources like instances (for nodes), Load Balancers (for services), and Storage Volumes (for persistent volumes).

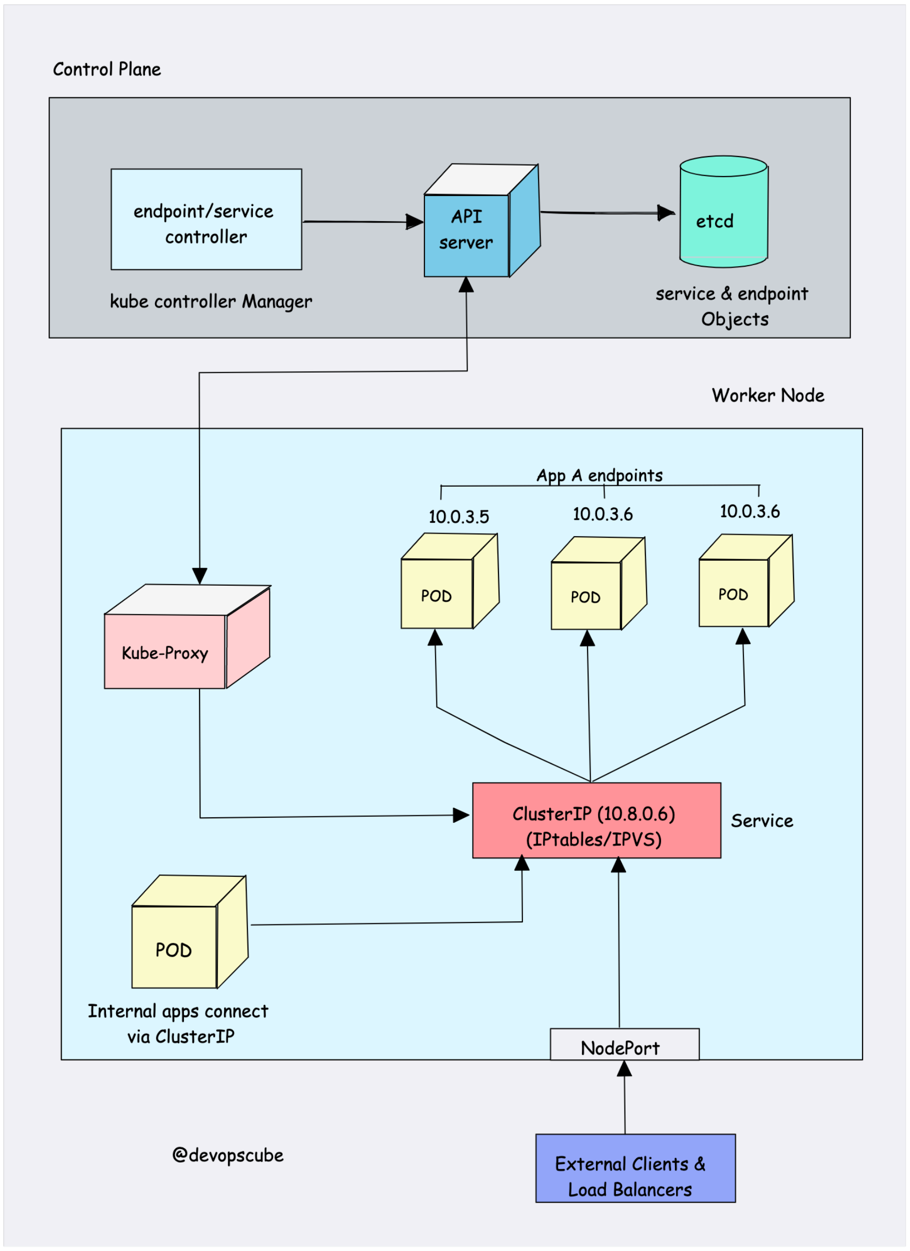

Cloud Platform
DNS
Managed Cloud Resources
Cloud Contorller Manager
Provision Storage Volume
Provision Load Balancer
Control Plane
Service
Pod
Pod
Storage Class
PVC
Pod
PVC
Pod
Worker Node
Worker Node

- **Node (Worker Node):** Nodes are the workhorses of a Kubernetes cluster. They run the applications and workloads. Each node contains:

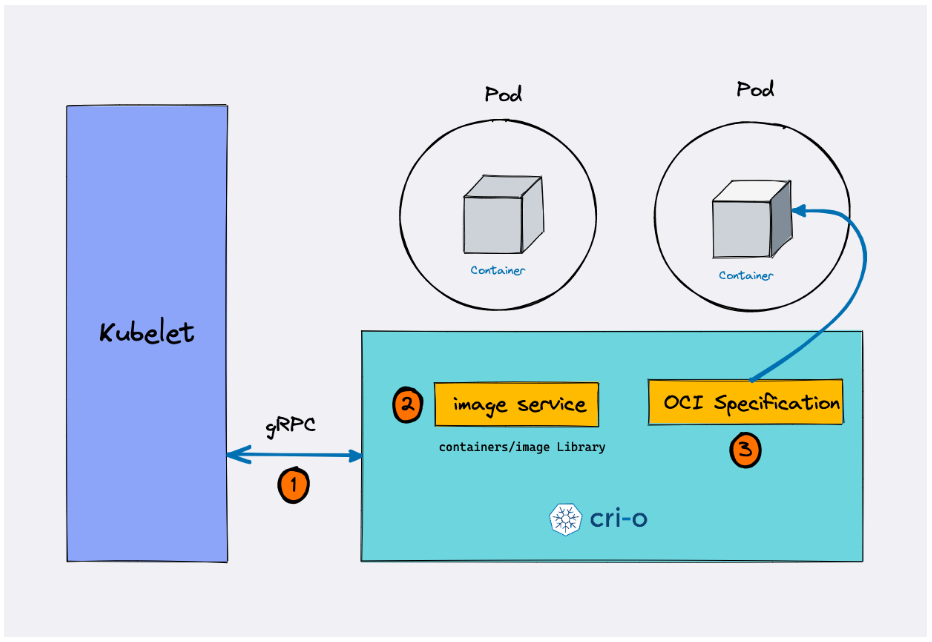
  * **Kubelet:** gets the configuration of a Pod from the API server and ensures that the described containers are up and running.


Control Plane
API Server
PodSpec
Pod/Node Status
CNI
Kubelet
10.0.2.3
Pod
Containers
/etc/kubernetes/manifests
Container Runtime (CRI-O, Containerd etc)
Static Pod Specs (pod1.yaml, pod2.yaml)
Worker Node

* **Kube-Proxy:** A network proxy that runs on each node, maintaining network rules and allowing communication to your Pods from network sessions inside or outside of your cluster.


Control Plane
endpoint/service controller
API server
etcd
kube controller Manager
service & endpoint Objects
Worker Node
App A endpoints
10.0.3.5
10.0.3.6
10.0.3.6
POD
POD
POD
Kube-Proxy
ClusterIP (10.8.0.6) (IPtables/IPVS)
Service
POD
Internal apps connect via ClusterIP
NodePort
@devopscube
External Clients & Load Balancers

∗ **Container Runtime:** The software responsible for running containers (like Docker).



- **Pods:** The smallest deployable units created and managed by Kubernetes. A pod is a group of one or more containers, with shared storage/network resources, and a specification for how to run the containers.

- **Services and Ingress:** Services define a logical set of Pods and a policy by which to access them. This is achieved using IP addresses and DNS names. Ingress, on the other hand, provides external access to the services within a cluster, typically via HTTP.

- **ConfigMaps and Secrets**: Allow you to store configuration data and sensitive information separately from your application code, which is particularly useful in microservices architecture.

- **Volumes:** Provide a way of storing data generated by and used by the Pods, independent of the lifecycle of the Pods.

– **Kubernetes High Availability**
High Availability (HA) in Kubernetes is a crucial feature for ensuring that applications and services running on the cluster are available and accessible at all times, minimizing

downtime and service interruptions. Let’s dive into how Kubernetes achieves high availability:

- **Control Plane Redundancy:**

  * In a high availability setup, the Kubernetes control plane components (API Server, etcd, Controller Manager, Scheduler) are replicated across multiple nodes. This ensures that if one or more nodes fail, the control plane remains operational.

  * The etcd storage, which is a critical component of the control plane, is often run in a clustered mode across multiple nodes to prevent data loss and enhance availability.

- **API Server Load Balancing:**

  * Requests to the Kubernetes API server are load balanced across multiple server instances. This prevents any single point of failure and ensures that the API server is always reachable.

  * In cloud environments, this is often managed by cloud load balancers, while in on-premises environments, it may involve dedicated hardware or software-based load balancing solutions.

- **Node Redundancy:**

  * High availability also extends to the worker nodes. By running multiple nodes, Kubernetes can tolerate the failure of one or more nodes while still keeping the applications running.

  * Kubernetes’ scheduling algorithm takes node availability into account, distributing Pods across different nodes to reduce the impact of a single node failure.

- **Automated Failover:**

  * Kubernetes continuously monitors the health of nodes and pods. If a node or pod fails, Kubernetes automatically reschedules the pods to healthy nodes.

  * Services in Kubernetes use selectors to automatically redirect traffic to available pods, ensuring that application services are continuously accessible even if some pods fail.

- **ReplicaSets and Deployments:**

- ∗ ReplicaSets ensure that a specified number of pod replicas are running at any given time. If a pod fails, the ReplicaSet automatically creates a new one.

- ∗ ReplicaSets ensure that a specified number of pod replicas are running at any given time. If a pod fails, the ReplicaSet automatically creates a new one.

- **StatefulSets:** For stateful applications (like databases), StatefulSets ensure that the state of the application is maintained even when pods are rescheduled. This is critical for applications that require persistent state and stable network identifiers.

- **Persistent Storage:** Kubernetes supports persistent storage, allowing data to be stored independently of pods. This means that even if a pod is rescheduled to a new node, it can still access its data, which is essential for high availability in stateful applications.

- **Cluster Federation:** For environments that require even higher levels of availability, Kubernetes supports cluster federation, allowing multiple Kubernetes clusters to be linked together. This provides fault tolerance across regions or data centers.

– **System Requirements**

- **Windows**

  - ∗ **CPU**: 2-core processor.
  - ∗ **Memory**:4 GB RAM minimum (8 GB recommended).
  - ∗ **Storage**: 20 GB of free space.
  - ∗ **Operating System**: Windows 10 Pro, Enterprise, or Education; Build 15063 or later.
  - ∗ **Container Runtime**: Docker Desktop for Windows (includes Kubernetes support), or Windows Subsystem for Linux 2 (WSL2) for a more Linux-like experience.
  - ∗ **Network**: General broadband internet or LAN connection.
  - ∗ **Virtualization**: Hardware virtualization support with Hyper-V and Containers Windows features enabled.

    ∗ Unique hostname, MAC address, and product_uuid for every node. See here for more details.

- **Linux**
    - ∗ **CPU**: 2-core processor.
    - ∗ **Memory**: 2 GB RAM minimum (4 GB recommended).
    - ∗ **Storage**: 20 GB of free space.
    - ∗ **Operating System**: Most modern distributions like Ubuntu 20.04, CentOS 7, Fedora, Debian, etc.
    - ∗ **Container Runtime**: Docker, containerd, or any runtime compatible with Kubernetes CRI (Container Runtime Interface).
    - ∗ **Network**: General broadband internet or LAN connection.
    - ∗ **Virtualization Support (for Minikube or similar)**: Enabled in BIOS.
    - ∗ Unique hostname, MAC address, and product_uuid for every node. See here for more details.
- **Mac**
    - ∗ **CPU**: Intel or Apple Silicon processor.
    - ∗ **Memory**: 4 GB RAM minimum (8 GB recommended).
    - ∗ **Storage**: 30 GB of free space.
    - ∗ **Operating System**: macOS 10.15 (Catalina) or later.
    - ∗ **Container Runtime**: Docker Desktop for Mac (includes Kubernetes support).
    - ∗ **Network**: General broadband internet or LAN connection.
    - ∗ **Virtualization**: Docker Desktop provides a virtual environment.
    - ∗ Unique hostname, MAC address, and product_uuid for every node. See here for more details.

– **Installation and Instructional resources** We adhered to the official Kubernetes guidelines for the installation process on various operating systems. Specifically, we followed the official instructions for installing on Windows, Linux, and Mac. Additionally, for a comprehensive understanding, we also referred to external tutorials like the Guru99 Kubernetes Tutorial and the DevOpsCube Kubernetes Guide for Beginners. Additionaly, we recommend watching a video tutorial provided here

3 Docker and Kubernetes: better together

To begin with, it's important to understand that Docker and Kubernetes are distinct technologies, each with its own purpose. They are not in competition with each other; rather, they are complementary tools often used in conjunction. The question shouldn't be about choosing one over the other or comparing their capabilities. Instead, it's about recognizing their individual roles within the DevOps landscape and how they can synergistically operate.Docker specializes in encapsulating your application into containers. Its primary role is in the packaging and distribution of your application.Conversely, Kubernetes serves as a container orchestration system. Its primary use is in the deployment and scaling of your application. When Docker and Kubernetes are combined, they form an efficient duo for DevOps processes. As previously mentioned, merely running containers isn't sufficient for production environments; they require effective management. This is where Kubernetes excels, offering features such as auto-scaling, health monitoring, and load balancing, which are vital for efficient container lifecycle management.

4 Conclusion

We have explored the integral roles of Docker and Kubernetes in the realm of DevOps, emphasizing their synergistic relationship. Docker, with its efficient containerization capabilities, offers a robust solution for packaging and distributing applications consistently across various environments. Kubernetes complements Docker's functionalities by providing advanced orchestration capabilities, ensuring scalable and reliable deployment and management of these containerized applications.

The combination of Docker's containerization and Kubernetes' orchestration forms a powerful duo in the DevOps toolkit, addressing key challenges in software development and deployment. This synergy not only enhances resource utilization and operational efficiency but also ensures high availability and scalability of applications in production environments.

Furthermore, We underscore that Docker and Kubernetes are not competing technologies but rather complementary. Docker simplifies the development process and creates a portable environment for applications, while Kubernetes efficiently manages these applications at scale. This partnership is crucial for enterprises looking to leverage the full potential of cloud-native technologies.

Looking ahead, the evolving landscape of containerization and orchestration, spearheaded by Docker and Kubernetes, presents new opportunities and challenges. Future research and development in these areas could focus on enhancing security, streamlining workflow management, and integrating emerging technologies to further augment the DevOps ecosystem.

In conclusion, Docker and Kubernetes together provide a comprehensive, efficient, and scalable solution for modern software development and deployment. Their combined use is not just a trend but a paradigm shift in how we build, deploy, and manage applications in the cloud era.

5 Sources

– Docker Documentation

– Docker Documentation

– Docker Documentation

– Docker Documentation

– Docker Documentation

– Docker Documentation

– Kubernetes Documentation

– Kubernetes Documentation

– Kubernetes Documentation