







Kubernetes

<input checked="" type="checkbox"/> Favorite	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> Archived	<input type="checkbox"/>
<input checked="" type="checkbox"/> Fleeting	<input type="checkbox"/>
↗ Area/Resource	
↗ Project	
▼ Type	
📅 Review Date	
📎 Image	
🔗 URL	
🕒 Created	@February 2, 2023 11:48 AM
🕒 Updated	@February 2, 2023 1:04 PM
🔍 Root Area	
🔍 Project Area	
Σ Updated (short)	02/02/2023
↗ Pulls	
🔍 Resource Pulls	
🔍 Project Archived	
Σ URL Base	
Σ 🔍 Recipe Divider	🥗🥗🥗 RECIPE BOOK PROPERTIES 🥗🥗🥗
☰ 🔍 Recipe Tags	
Σ 📖 Book Divider	📖📖📖 BOOK TRACKER PROPERTIES 📖📖📖
☰ 📖 Author	
📅 📖 Date Started	

  Date Finished	
  Book Status	
  Rating	

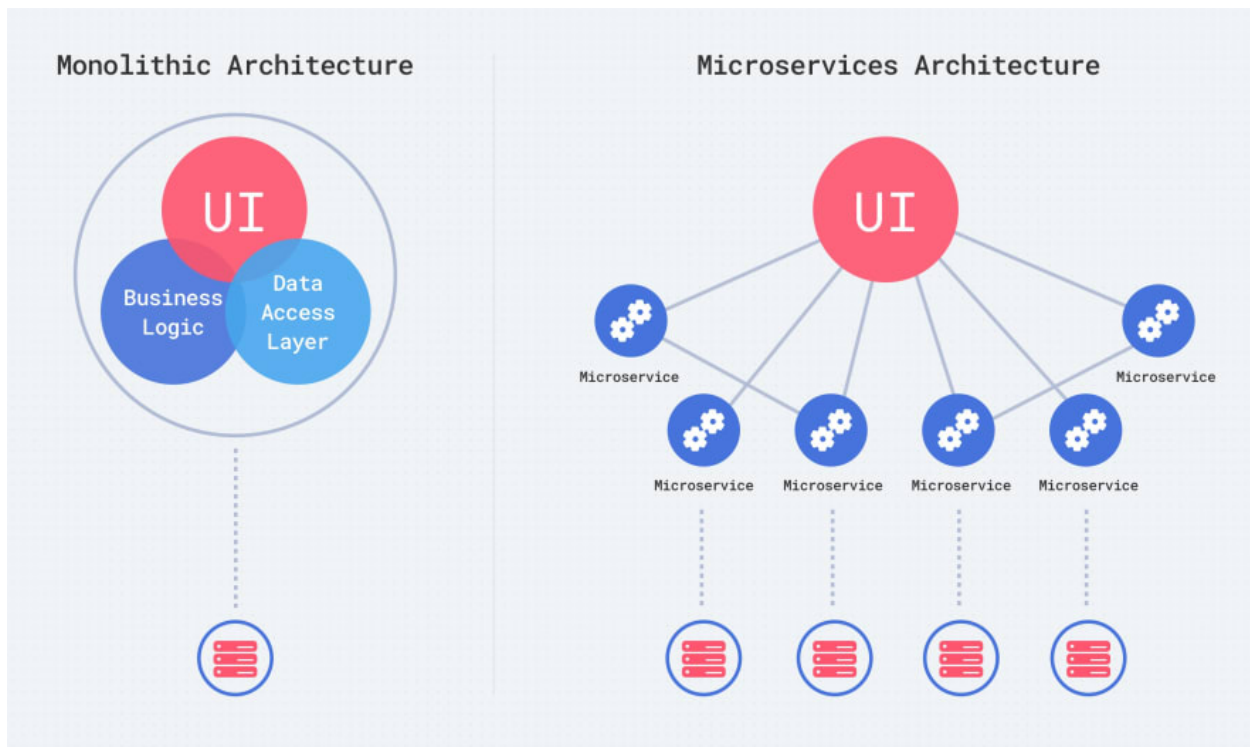


To understand the need for kubernetes we have to understand the rise of microservice architecture in software development.

Monolithic vs Microservices



The shift in software development from monolithic architecture to microservices can be explained by the rising need for software companies to keep up with agile development practices. Quicker deployment times, and delivering value more efficiently and flexibly meant that monolithic app bases were no longer capable of keeping up with the changing trends. The shift reflects the change in how apps are designed, developed, and deployed.



Monolithic vs Microservices architecture

Monolithic architecture



A monolithic architecture is a traditional software design which was popular in the early days of software development where apps were simpler and required less computing power. The functionality of the app including the UI, business logic and data access layers would all be built tightly together. However, this meant that as application complexity grew, each new feature or scaling up meant the entire app would have to be tested and redeployed.

Microservices architecture

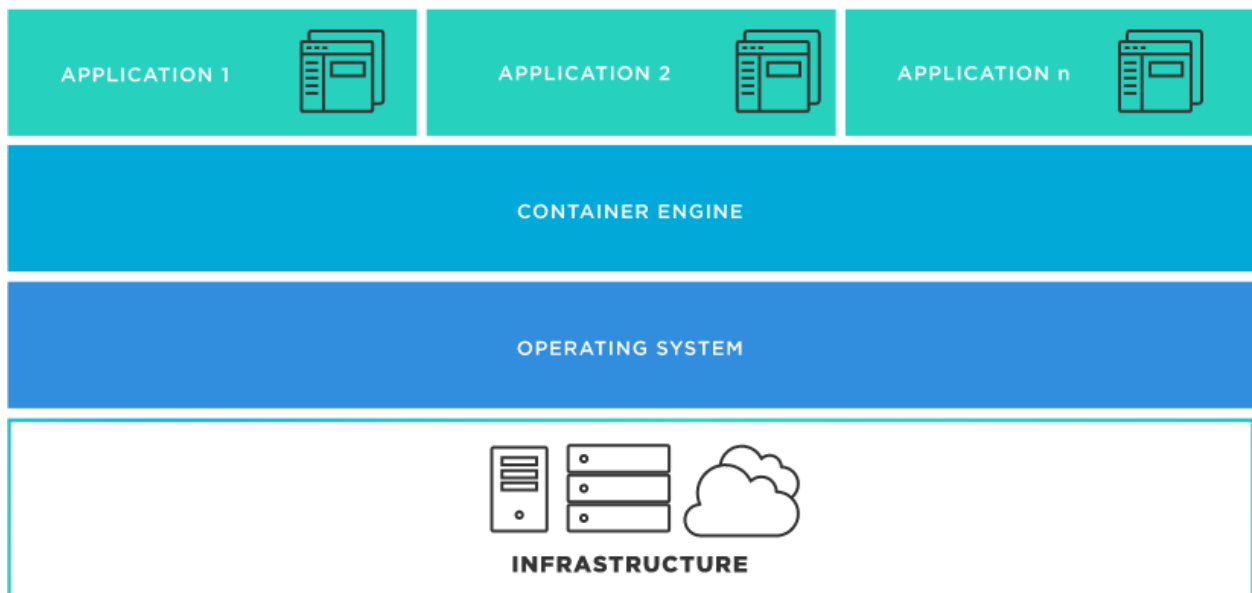


Microservices architecture addressed these limitations by breaking down large app architecture into smaller components which each handled its own business logic. Each of these services are independent and loosely coupled, allowing for better reliability (one error would not take down the whole system), maintainability (easier to test and deploy) and scalability (each function can be independently scaled).

Virtualization and containerization



Containerization is the practice in software development of packaging apps into a logical unit where all libraries, dependencies, code logic and configuration files are deployed in a unit called a container. This makes the application portable and easily deployed in different environments.



Virtualization of underlying physical infrastructure using containers.



Containerization is useful when apps may have different or conflicting dependencies, and means that apps can be quickly deployed on different infrastructure. This is based on the concept of virtualization.



Virtualization is the process of creating a virtual version of something such as an OS, network system, storage device or server. This allows multiple versions to be hosted on the same infrastructure which shares resources. This leads to large cost savings as the underlying resources of the infrastructure resources are not completely dependent on the software running on top of it and can run several versions on it maximizing its resource usage.



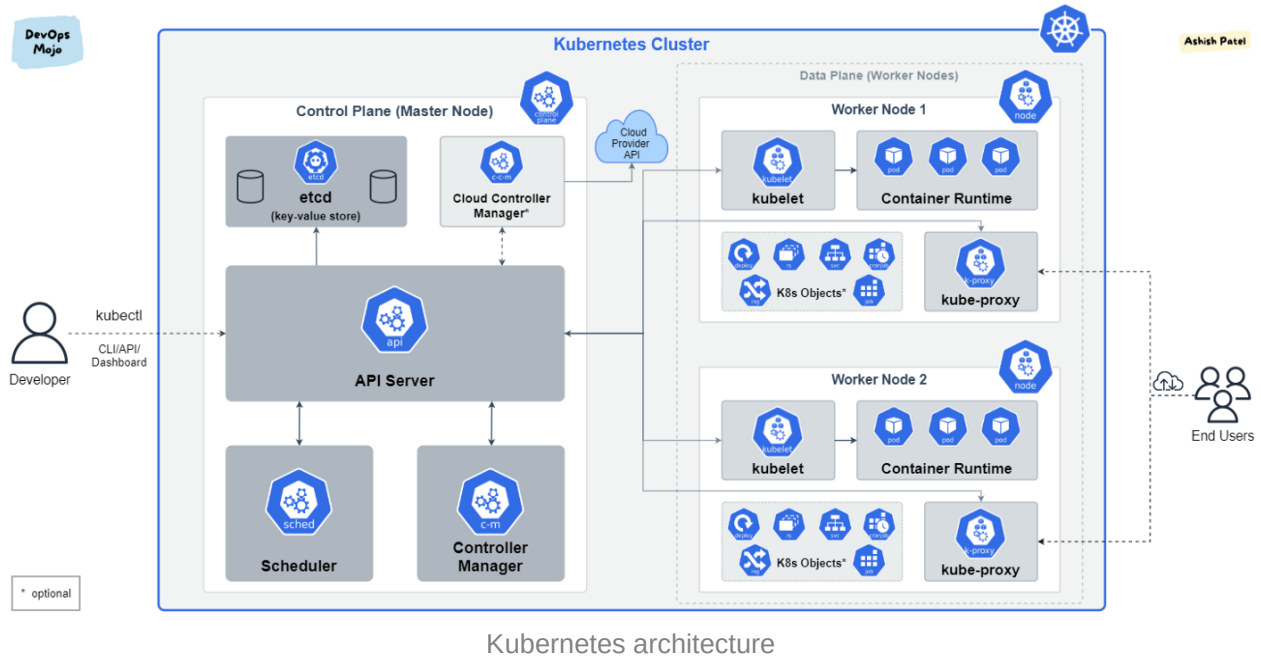
Docker is platform which uses containerization technology which is a form of virtualization. It enables applications to be deployed as containers. The docker runtime which runs on top of the OS would interact with the hosts kernel, and allocate resources to certain containers running on the host.

Kubernetes



Knowing that the shifting software practices to more agile standards, lead to the rise of microservice architecture, which utilizes docker containers based on the concept of virtualization, where does Kubernetes come in? Kubernetes ensures high availability (reduced downtime), scalability (high performance) and disaster recovery (backup and restore). It does this by orchestrating the containers that when combined together through a central API (Django) can create a full purpose app. The containers can be easily scaled, deployed and managed by Kubernetes which will handle high load and traffic demand by horizontally scaling out containers to ensure the features above.

Architecture



Master node (control plane): The main control plane which manages the state of the cluster and exposes the API and is responsible for orchestration and scheduling

etcd: A key value store that stores the configuration data and state of the cluster

API server: The component that exposes the kubernetes API for communicating with other components

Controller manager: Handles routine tasks such as replicating pods, responding to changes in the state of the cluster and adjusting the state of the cluster

Scheduler: Schedules pods to run on nodes based on resource availability and other constraints.

Worker node: The node that runs the containers and is managed by the master node.

Kubelet: A component that communicates with the master node and ensures that the containers are running as expected.

Container runtime: The component that runs containers, such as Docker or CRI-O.



The state of the cluster refers to the current configuration of the cluster. Such as the number of nodes, amount of resources available. It is updated by the API and stored in etcd. It is important because it enables the various components of the cluster, such as the scheduler and controller manager, to make informed decisions about the placement and management of resources. For example, the scheduler can use the state of the cluster to determine which nodes have sufficient resources to run new pods, while the controller manager can use the state of the cluster to monitor the health of the cluster and make adjustments as necessary.



The virtual network allows the nodes, and master nodes to talk to each other. The virtual network turns all the nodes in the cluster into a single unified machine which has the sum of all the resources based in the nodes.

- Worker nodes would be much bigger as it is handling most of the work, therefore it needs more resources
- Master nodes are much more important but are smaller, and therefore you need a backup of your master nodes.

Kubernetes concepts



A pod is the smallest unit that a k8s user can interact with. Each pod hosts the containers. Usually you would have one pod per application. i.e. a database, a message broker, node js app. Each pod is based within a worker node. As the virtual network allows for communication between nodes and pods, each pod has it's own IP address. Therefore each pod is it's own self containing server with it's own IP address. We do not configure or create containers within a cluster. We only work with the abstraction layer over the container which is the pod.

- Pods are ephemeral components, which means they can die and be replaced.

- Each time a pod dies and is replaced it gets a new IP address
 - This is why we can update the state of these pods i.e. the code in the containers



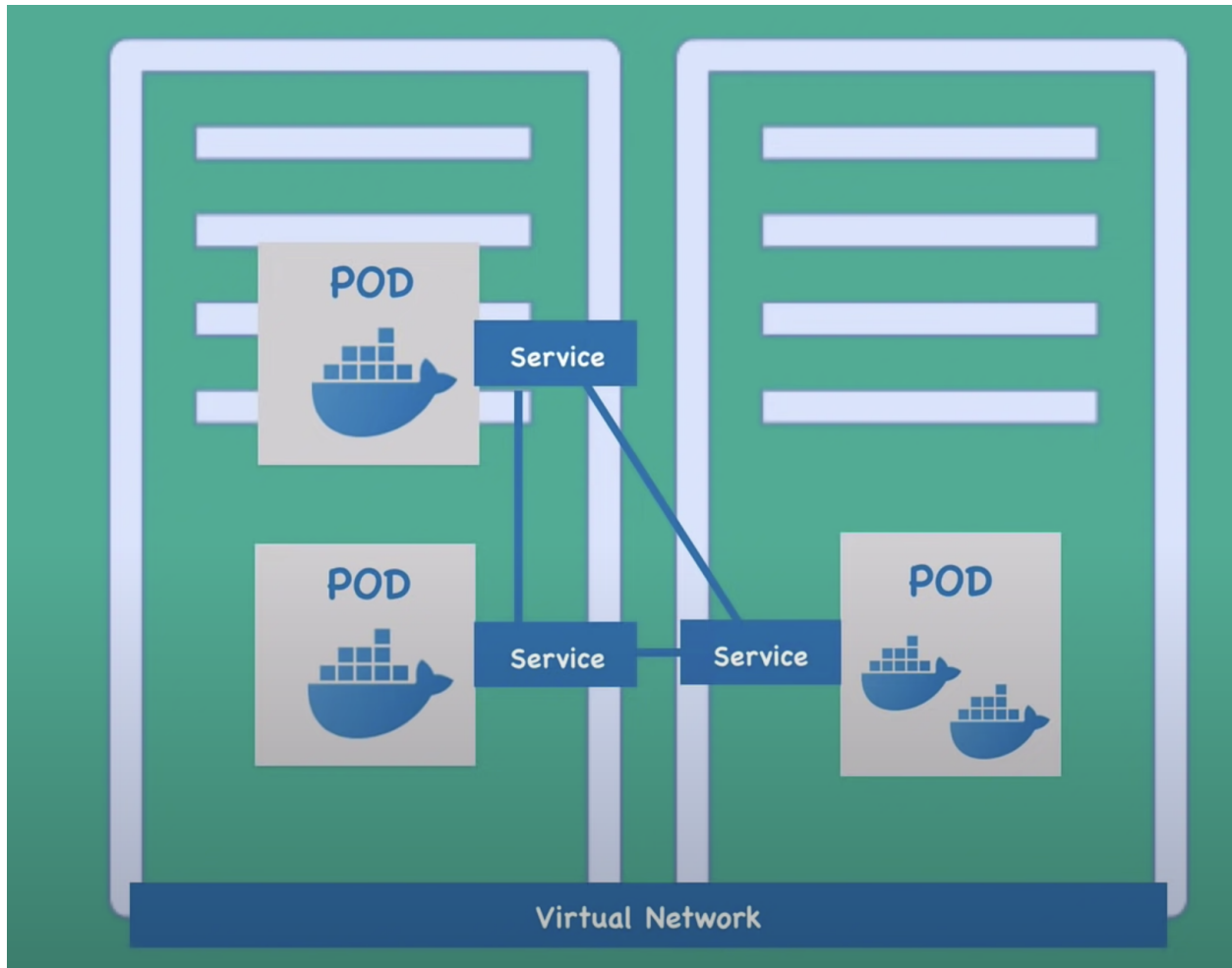
Ephemeral components are meant to be short lived and transitory.



Service: A component of k8s which is an alternative to the IP address. Each service sits in front of a pod, and talks to each other. The lifecycle of a service and a pod are not tied to each other.

Two features of a service:

1. Permanent IP address
2. Load balancer



Kubernetes configuration



Kubernetes client which could be a UI, API or CLI talk to the API server in k8s and send configuration requests to the server. They have to be either in YAML format or JSON format. The configuration requests are declarative form. (Desired outcomes to be met).

- ☐ Look into YAML format
- ☐ Look into ports