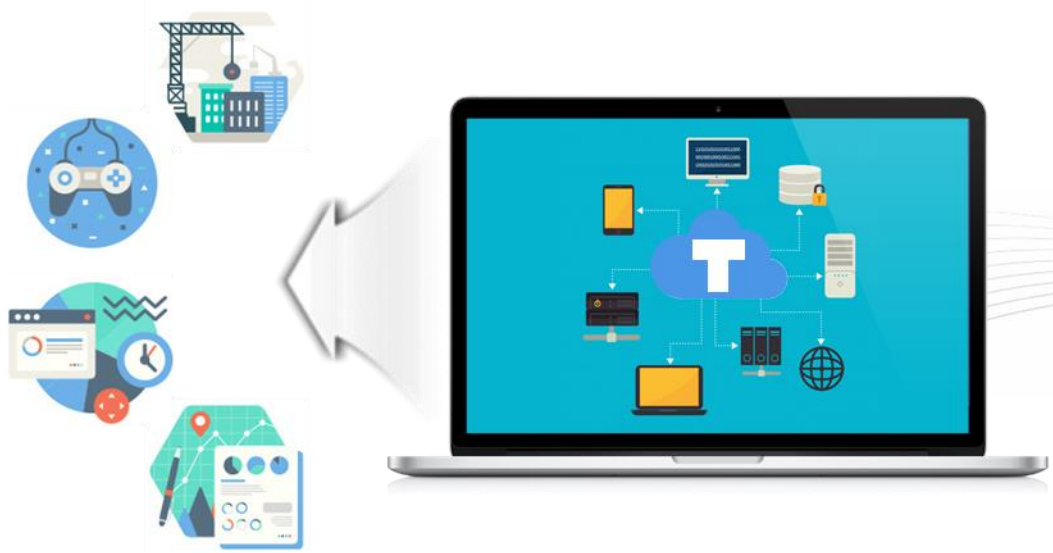


(이론) GCP-7차수

Ⅰ Kubernetes

Ⅱ GKE(Google Kubernetes Engine)

Ⅲ CI/CD

Kubernetes 소개

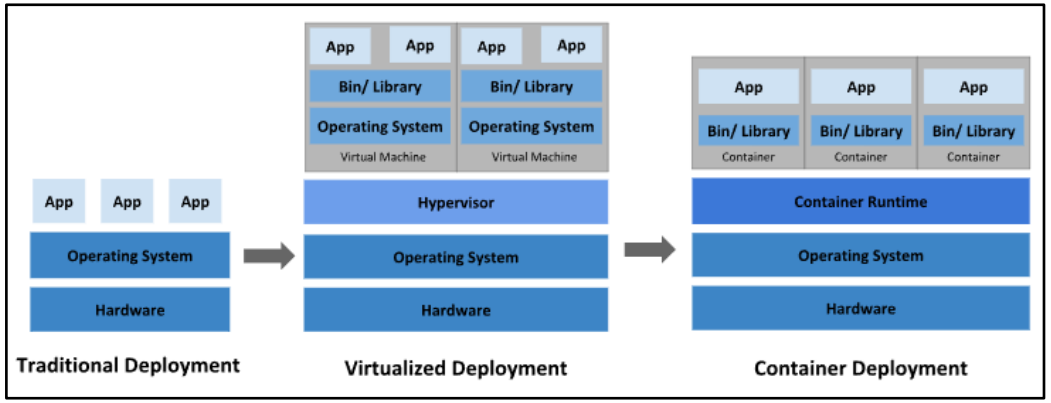
- kubernetes란 명칭은 키잡이(helmsman)나 파일럿을 뜻하는 그리스어에서 유래함
- 컨테이너를 도입하는 조직이 점점 많아짐에 따라 컨테이너 중심 관리 소프트웨어인 Kubernetes는 사실상 컨테이너화된 애플리케이션을 배포하고 운영하기 위한 표준이 됨
- Kubernetes는 원래 Google에서 개발되어 2014년에 오픈소스로 출시됨
- Kubernetes는 15년 동안 축적된 Google의 컨테이너화된 워크로드 실행 경험과 오픈소스 커뮤니티의 소중한 기여를 기반으로 성장함
- Google의 내부 클러스터 관리 시스템인 Borg에서 영감을 얻어 시작된 Kubernetes를 사용하면 애플리케이션의 배포 및 관리와 관련된 모든 작업이 한층 간편해짐
- 자동화된 컨테이너 조정 기능을 제공하는 Kubernetes는 신뢰성을 개선하고 일상적인 작업에 소요되는 시간과 리소스를 절감함
- URL : https://kubernetes.io/

[Kubernetes의 정의]

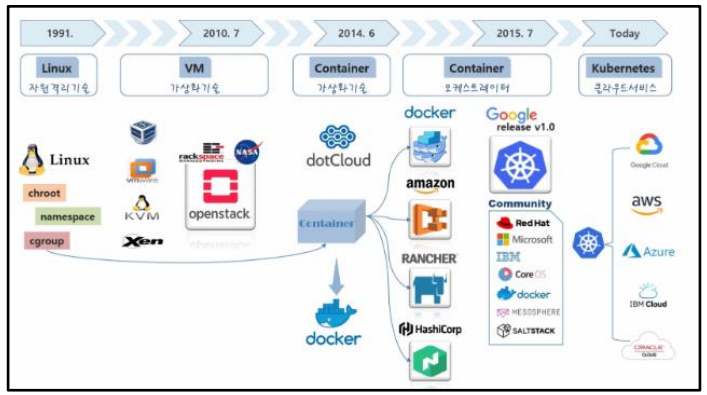
- Kubernetes('K'와 's' 사이의 문자 수를 나타내는 8을 사용하여 K8s로 줄여 쓰기도 함)는 컨테이너화된 애플리케이션을 어디서나 배포, 확장, 관리할 수 있는 오픈소스 시스템

Kkubernetes 역사

- 전통적인 배포 시대:
  - 초기 조직은 애플리케이션을 물리 서버에서 실행함
  - 한 물리 서버에서 여러 애플리케이션의 리소스 한계를 정의할 방법이 없었기에, 리소스 할당의 문제가 발생함
- 가상화된 배포 시대:
  - 그 해결책으로 가상화가 도입되었으며, 이는 단일 물리 서버의 CPU에서 여러 가상 시스템 (VM)을 실행할 수 있게 함
  - 각 VM은 가상화된 하드웨어 상에서 자체 운영체제를 포함한 모든 구성 요소를 실행하는 하나의 완전한 머신임
- 컨테이너 개발 시대:
  - 컨테이너는 VM과 유사하지만 격리 속성을 완화하여 애플리케이션 간에 운영체제(OS)를 공유함
  - VM과 마찬가지로 컨테이너에는 자체 파일 시스템, CPU 점유율, 메모리, 프로세스 공간 등이 있음
  - 기본 인프라와의 종속성을 끊었기 때문에, 클라우드나 OS 배포본에 모두 이식할 수 있음



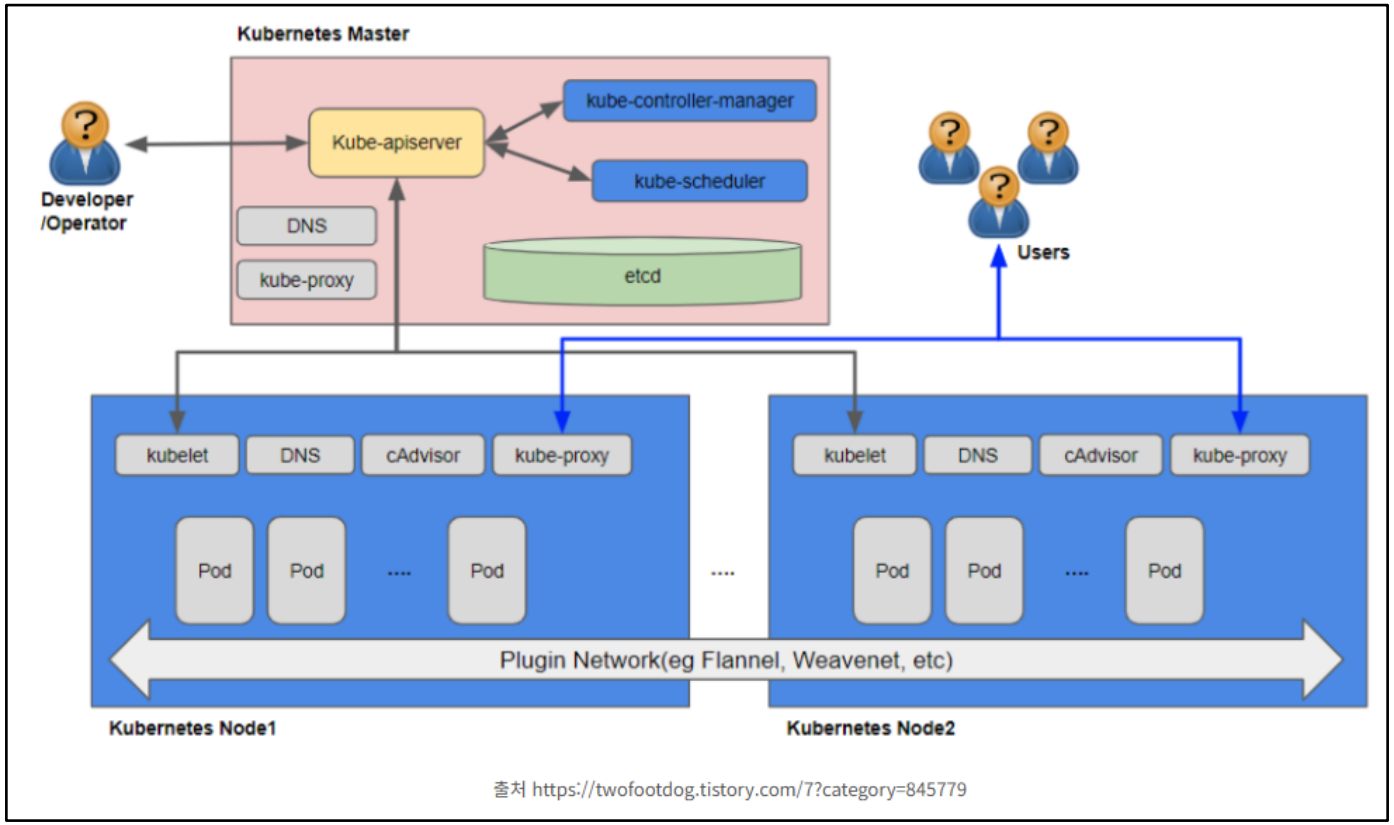
[그림 1] Deployment 환경 변화



[그림 2] Deployment 변화 역사

Kubernetes 아키텍쳐

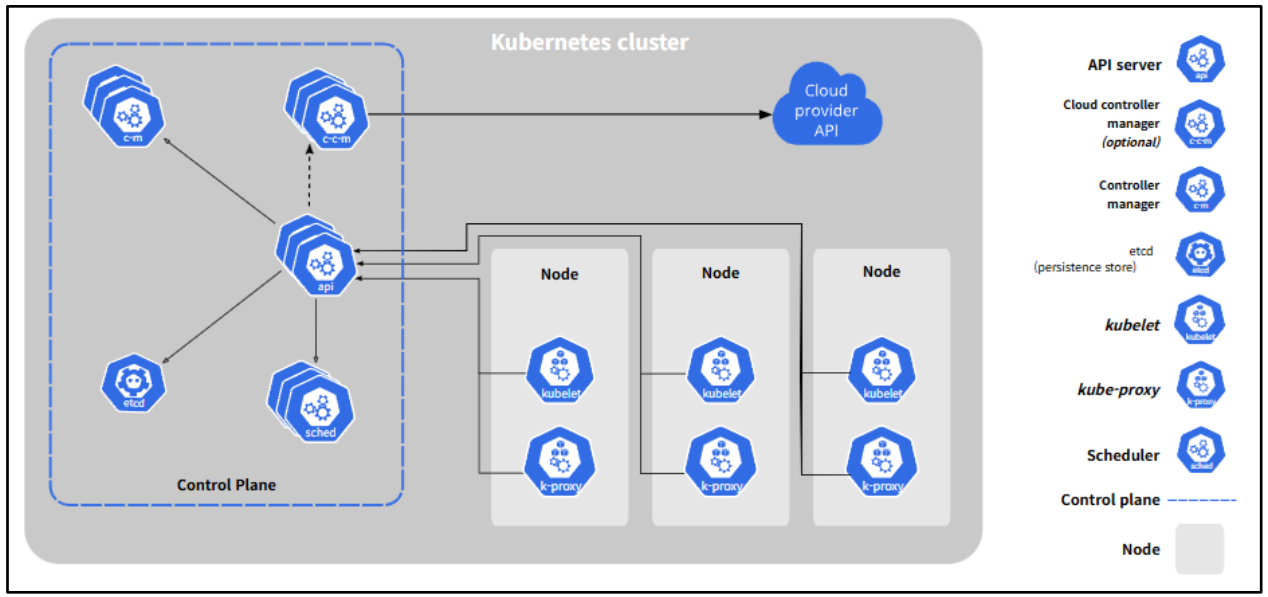
- 하나의 마스터노드와(master-node) 여러 대의 워커노드로(worker-node) 하나의 클러스터를 이루고 있는 구조
- 프러덕션(prd) 환경에서는 마스터 노드가 죽으면 클러스터를 관리할 수 없기 때문에 보통 2대 이상으로 구성함



[그림 3] Kubernetes 아키텍처

Kubernetes 컴포넌트

- 쿠버네티스 클러스터는 컴퓨터 집합인 **노드 컴포넌트**와 **컨트롤 플레인 컴포넌트**로 구성됨
- 쿠버네티스 클러스터는 컨테이너화된 애플리케이션을 실행하는 노드라고 하는 워커 머신의 집합이며, 모든 클러스터는 최소 한 개의 워커 노드를 가짐
- **워커 노드는 애플리케이션의 구성요소인 파드를 호스트함**
- **컨트롤 플레인은 워커 노드와 클러스터 내 파드를 관리함**
- 프로덕션 환경에서는 일반적으로 컨트롤 플레인이 여러 컴퓨터에 걸쳐 실행되고, 클러스터는 일반적으로 여러 노드를 실행하므로 내 결함성과 고가용성이 제공됨



[그림 4] Kubernetes 컴포넌트

컨트롤 플레인 컴포넌트(마스터 노드)

- 클러스터에 관한 전반적인 결정(예를 들어, 스케줄링)을 수행하고 클러스터 이벤트(예를 들어, 디플로이먼트의 replicas 필드에 대한 요구 조건이 충족되지 않을 경우 새로운 파드를 구동시키는 것)를 감지하고 반응함
- 클러스터 내 어떠한 머신에서든지 동작할 수 있음
- 그러나 간결성을 위하여, 구성 스크립트는 보통 동일 머신 상에 모든 컨트롤 플레인 컴포넌트를 구동시키고, 사용자 컨테이너는 해당 머신 상에 동작시키지 않음

구성요소	설명
kube-apiserver	• API 서버는 쿠버네티스 API를 노출하는 쿠버네티스 컨트롤 플레인 컴포넌트 • API 서버는 쿠버네티스 컨트롤 플레인의 프론트 엔드 • 수평으로 확장되도록 디자인되있어 더 많은 인스턴스를 배포해서 확장할 수 있음 • 여러 kube-apiserver 인스턴스를 실행하고, 인스턴스간의 트래픽을 균형있게 조절할 수 있음
etcd	• 모든 클러스터 데이터를 담는 쿠버네티스 뒷단의 저장소로 사용되는 일관성·고가용성 키-값 저장소 • 쿠버네티스 클러스터에서 etcd를 뒷단의 저장소로 사용한다면, 이 데이터를 백업하는 계획은 필수임
kube-scheduler	• 노드가 배정되지 않은 새로 생성된 파드 를 감지하고, 실행할 노드를 선택함 • 스케줄링 결정을 위해서 고려되는 요소는 리소스에 대한 개별 및 총체적 요구 사항, 하드웨어/소프트웨어/정책적 제약, 어피니티(affinity) 및 안티-어피니티(anti-affinity) 명세, 데이터 지역성, 워크로드-간 간섭, 데드라인을 포함함

[표 1] Kubernetes – 컨트롤 플레인 컴포넌트

구성요소	설명
kube-controller-manager	• 논리적으로, 각 컨트롤러는 분리된 프로세스이지만, 복잡성을 낮추기 위해 모두 단일 바이너리로 컴파일되고 단일 프로세스 내에서 실행됨 - 노드 컨트롤러: 노드가 다운되었을 때 통지와 대응에 관한 책임을 가짐 - 잡 컨트롤러: 일회성 작업을 나타내는 잡 오브젝트를 감시한 다음, 해당 작업을 완료할 때까지 동작하는 파드를 생성함 - 엔드포인트슬라이스 컨트롤러: (서비스와 파드 사이의 연결고리를 제공하기 위해) 엔드포인트슬라이스(EndpointSlice) 오브젝트를 채움 - 서비스어카운트 컨트롤러: 새로운 네임스페이스에 대한 기본 서비스어카운트(ServiceAccount)를 생성함
cloud-controller-manager	• 클라우드별 컨트롤 로직을 포함하는 쿠버네티스 컨트롤 플레인 컴포넌트 • 클라우드 컨트롤러 매니저를 통해 클러스터를 클라우드 공급자의 API에 연결하고, 해당 클라우드 플랫폼과 상호 작용하는 컴포넌트와 클러스터와만 상호 작용하는 컴포넌트를 구분할 수 있게 해 줌 • cloud-controller-manager는 클라우드 제공자 전용 컨트롤러만 실행함 • kube-controller-manager와 마찬가지로 cloud-controller-manager는 논리적으로 독립적인 여러 컨트롤 루프를 단일 프로세스로 실행하는 단일 바이너리로 결합함 • 수평으로 확장(두 개 이상의 복제 실행) 해서 성능을 향상시키거나 장애를 견딜 수 있음 • 아래 컨트롤러들은 클라우드 제공 사업자의 의존성을 가짐 - 노드 컨트롤러: 노드가 응답을 멈춘 후 클라우드 상에서 삭제되었는지 판별하기 위해 클라우드 제공 사업자에게 확인함 - 라우트 컨트롤러: 기본 클라우드 인프라에 경로를 구성 - 서비스 컨트롤러: 클라우드 제공 사업자 로드밸런서를 생성, 업데이트 그리고 삭제함

[표 2] Kubernetes – 컨트롤 플레인 컴포넌트

노드 컴포넌트(워커노드)

- 동작 중인 파드를 유지시키고 쿠버네티스 런타임 환경을 제공하며, 모든 노드 상에서 동작함

구성요소	설명
kubelet	• 클러스터의 각 노드에서 실행되는 에이전트 • Kubelet은 파드에서 컨테이너가 확실하게 동작하도록 관리함 • Kubelet은 다양한 메커니즘을 통해 제공된 파드 스펙(PodSpec)의 집합을 받아서 컨테이너가 해당 파드 스펙에 따라 건강하게 동작하는 것을 확실히 함 • Kubelet은 쿠버네티스를 통해 생성되지 않는 컨테이너는 관리하지 않음
kube-proxy	• kube-proxy는 클러스터의 각 노드에서 실행되는 네트워크 프록시로, 쿠버네티스의 서비스 개념의 구현부 • kube-proxy는 노드의 네트워크 규칙을 유지 관리하며, 네트워크 규칙이 내부 네트워크 세션이나 클러스터 바깥에서 파드로 네트워크 통신을 할 수 있도록 함 • kube-proxy는 운영 체제에 가용한 패킷 필터링 계층이 있는 경우, 이를 사용함 • 그렇지 않으면, kube-proxy는 트래픽 자체를 포워드(forward)함
컨테이너 런타임	• 컨테이너 런타임은 컨테이너 실행을 담당하는 소프트웨어 • 쿠버네티스는 containerd, CRI-O와 같은 컨테이너 런타임 및 모든 Kubernetes CRI (컨테이너 런타임 인터페이스) 구현체를 지원함

[표 3] Kubernetes – 노드 컴포넌트

애드온

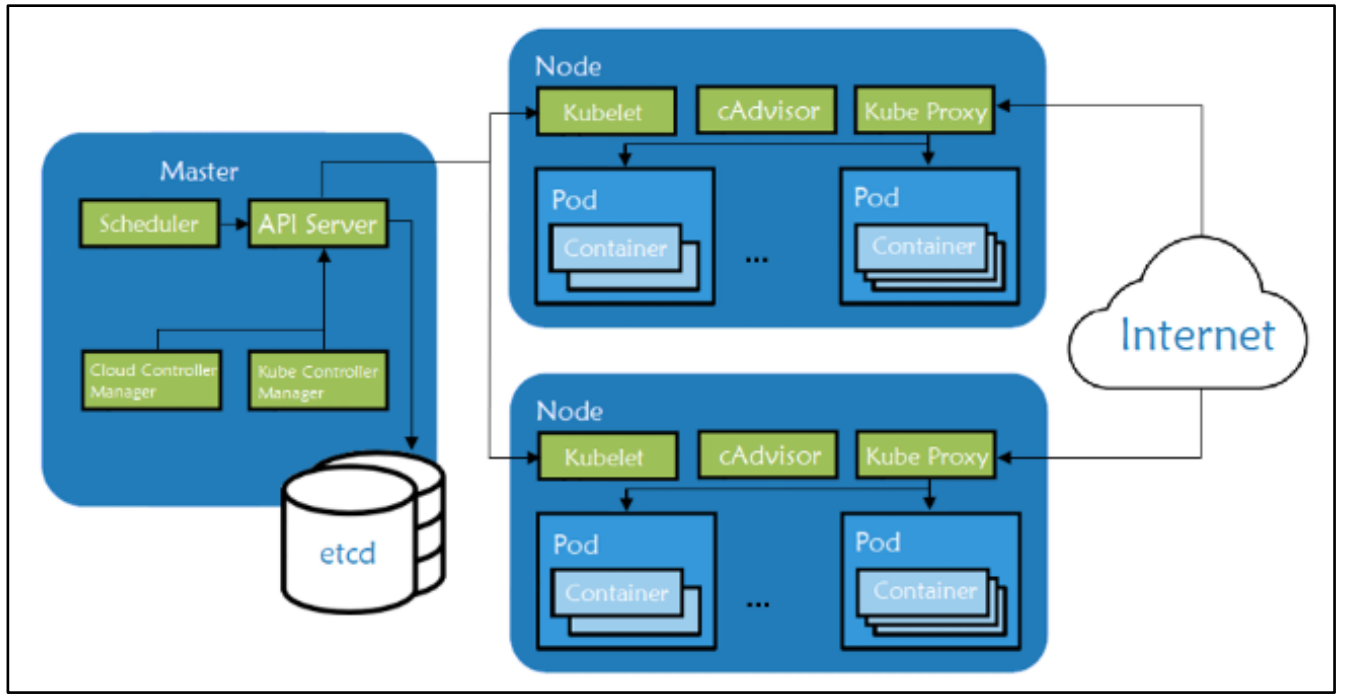
- 쿠버네티스 리소스(데몬셋, 디플로이먼트 등)를 이용하여 클러스터 기능을 구현함
- 이들은 클러스터 단위의 기능을 제공하기 때문에 애드온에 대한 네임스페이스 리소스는 kube-system 네임스페이스에 속함

구성요소	설명
DNS	• 여타 애드온들이 절대적으로 요구되지 않지만, 많은 예시에서 필요로 하기 때문에 모든 쿠버네티스 클러스터는 클러스터 DNS를 갖추어야만 함 • 클러스터 DNS는 구성환경 내 다른 DNS 서버와 더불어, 쿠버네티스 서비스를 위해 DNS 레코드를 제공해주는 DNS 서버임 • 쿠버네티스에 의해 구동되는 컨테이너는 DNS 검색에서 이 DNS 서버를 자동으로 포함
웹 UI (대시보드)	• 대시보드는 쿠버네티스 클러스터를 위한 범용의 웹 기반 UI • 사용자가 클러스터 자체뿐만 아니라, 클러스터에서 동작하는 애플리케이션에 대한 관리와 문제 해결에 도움
컨테이너 리소스 모니터링	• 컨테이너 리소스 모니터링은 중앙 데이터베이스 내의 컨테이너들에 대한 포괄적인 시계열 매트릭스를 기록하고 그 데이터를 열람하기 위한 UI를 제공함
클러스터-레벨 로깅	• 클러스터-레벨 로깅 메커니즘은 검색/열람 인터페이스와 함께 중앙 로그 저장소에 컨테이너 로그를 저장함

[표 4] Kubernetes – 애드온

Pods

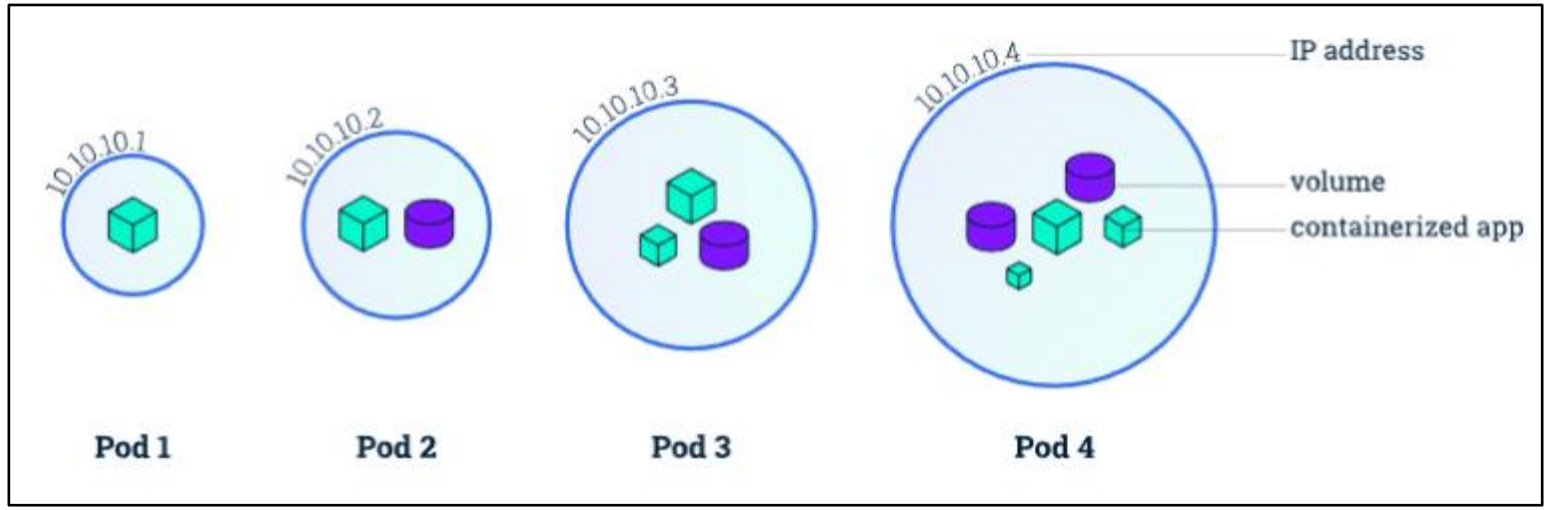
- Pod(파드, 포드, 팟)는 쿠버네티스가 생성하고 관리하는 가장 작은 컴퓨팅 단위
- 파드는 한 개 이상의 리눅스 컨테이너로 구성되며, 애플리케이션(의 인스턴스)이 실행되는 논리적 호스트(컴퓨터)
- 파드는 물리 컴퓨터인 쿠버네티스 워커노드에 배치되어 실행되는데, 쿠버네티스의 목적은 파드들을 안정적이고 효율적으로 클러스터 내에서 실행하는 것임



[그림 5] Kubernetes – Pods

Pod와 Container

- 파드는 한 개 이상의 리눅스 컨테이너로 구성
- 애플리케이션 종류에 따라서 한 애플리케이션이 여러 개의 컨테이너로 구성되게 개발하는 경우도 많음
- 이 컨테이너들을 하나의 파드에 묶어서 쿠버네티스에 실행을 명령할 수도 있고, 각각 다른 파드로 만들어 여러 개의 파드들을 쿠버네티스에 실행시켜달라고 명령할 수도 있음
- 두 가지 방법 모두 사용되고 있는 개발 방식으로 각기 장단점이 존재하기에 자신의 애플리케이션에 어울리는 방식으로 개발해야 함



[그림 6] Pod와 Container의 관계

Pod 내 Container

- 파드 내 컨테이너는 크게 3가지, 네트워크 네임스페이스, IPC 네임스페이스, 스토리지를 공유
- 파드에는 기본적으로 하나의 private IP 주소가 부여됨
- 파드 내 컨테이너들은 컨테이너가 속한 파드의 IP 주소를 이용해 접근할 수 있으며, 파드 내에 여러 컨테이너가 동시에 존재할 경우 이들 간의 구별은 포트(port)를 통해서 이루어짐
- IPC 네임스페이스를 공유하기에 같은 파드 내 컨테이너들끼리는 IPC 통신(shared memory, etc.) 할 수도 있음
- 또한 파드 내 컨테이너들에게만 공유되는 공유 스토리지를 지정할 수 있음

Pod : Container = 1 : 1

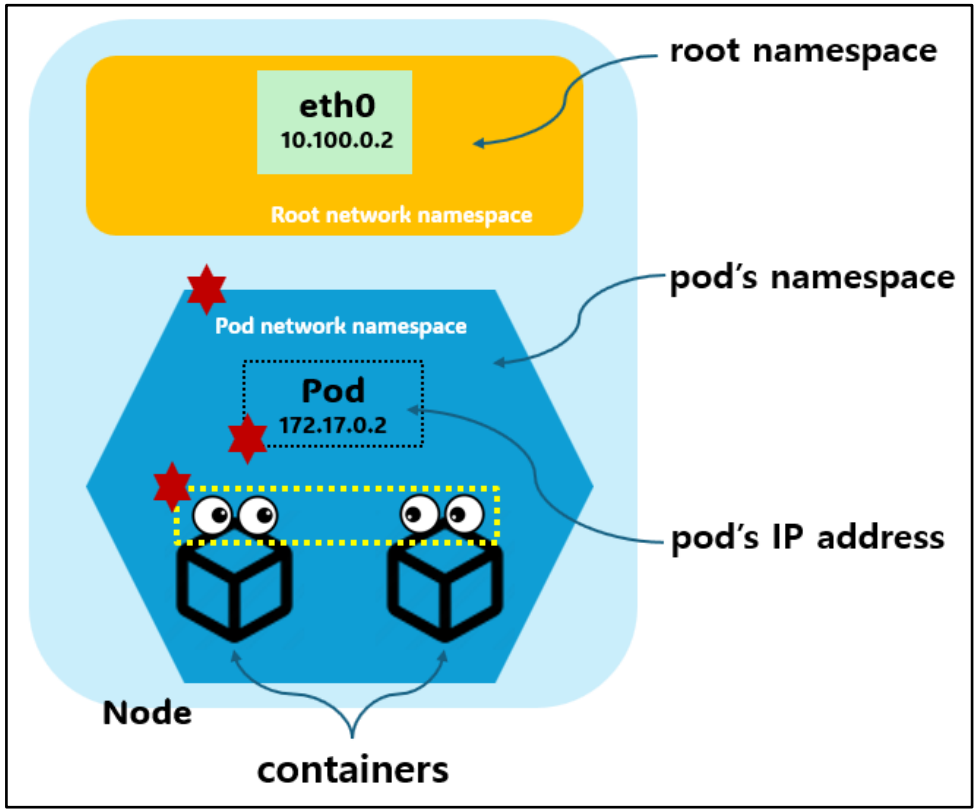
- 일반적으로 애플리케이션의 개별 기능은 각각 파드로 따로 구현하는 것을 추천
- 기능에 버그가 발생하더라도 다른 파드에 영향 없이 그 파드만 제거, 수정할 수 있게 됨
- 각 파드(기능)들 간의 통신은 파드의 private IP 주소를 통해서 이루어진다(REST API 등). (외부 사용자가 파드에 접근하기 위해서는 Service라는 쿠버네티스 리소스를 통해 접근)

Pod : Container = 1 : N

- 파드 내 여러 컨테이너가 함께 포함되는 경우는 한 기능을 여러 컨테이너로 구현하거나, 메인 기능 이외에 부가적인 기능(debug를 위한 log 기능 등)을 같이 사용하는 경우
- 파드 내 여러 컨테이너들을 동시에 사용할 경우 컨테이너는 위의 자원들을 공유하여 사용할 수 있기에 더욱 편리한 개발이 가능
- 다만 파드 하나에 에러가 발생하면 그 파드 내 컨테이너들 모두 재시작되어야 한다거나, 컨테이너 이미지 수정이 필요한 경우에 파드 전체를 종료시키고 새로 만들어야 하기에 개발 과정이 복잡해짐

Pod 배포 시 네트워크 구성

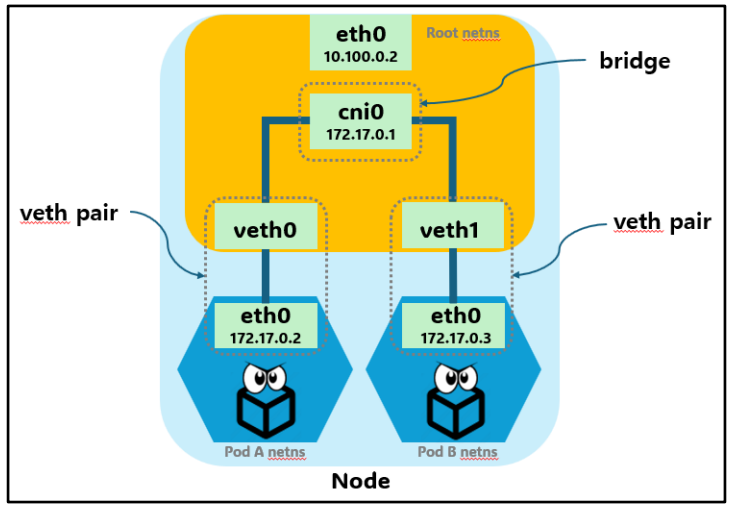
- Pod는 자체 network namespace를 얻음
- IP 주소가 할당됨
- Pod의 모든 컨테이너는 동일한 network namespace를 공유하며 로컬호스트에서 서로를 볼 수 있음



[그림 7] Pod 배포 및 네트워크 구성

Pod 가 다른 Pod에 도달하려면 ?

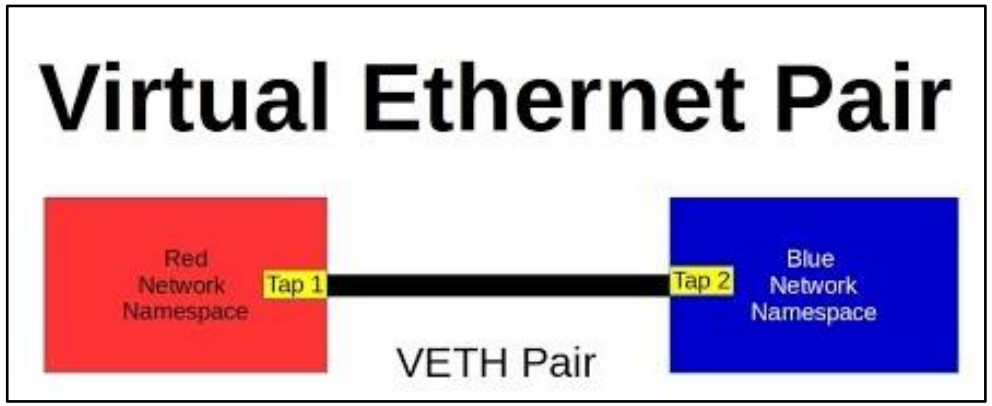
- Pod 가 다른 Pod에 도달하려면 먼저 Node 의 root namespace 에 대한 액세스 권한이 있어야 함
- 이는 두 namespace인 Pod 와 Root를 연결하는 virtual eth pair 을 사용하여 수행됨
- Bridge는 트래픽이 virtual pairs 사이를 이동하고 공통 루트 root namespace를 통과할 수 있게 해줌



[그림 8] Pod 간 통신 구조

"Virtual Ethernet Pair" (VETH, 가상 이더넷 쌍)

- 가상 이더넷 쌍은 컴퓨터 시스템 내에서 두 개의 가상 네트워크 인터페이스를 연결하는 가상의 네트워크 케이블과 같은 역할을 함
- 마치 물리적인 이더넷 케이블이 두 장치를 연결하듯이, 가상 환경에서 두 네트워크 인터페이스를 서로 연결



[그림 9] Virtual Ethernet Pair 구성

Ethernet Bridge

- 여러 네트워크 인터페이스(예: 컨테이너, 가상 머신 등)를 하나의 네트워크로 묶어 주고, 이들이 서로 통신할 수 있게 해줌
- Kubernetes는 CNI(Container Network Interface) 플러그인을 사용해 Pod 간의 네트워킹을 관리하며, 이 CNI 플러그인이 Ethernet Bridge를 사용해 Pod 간의 트래픽을 관리함
- 작동 방식

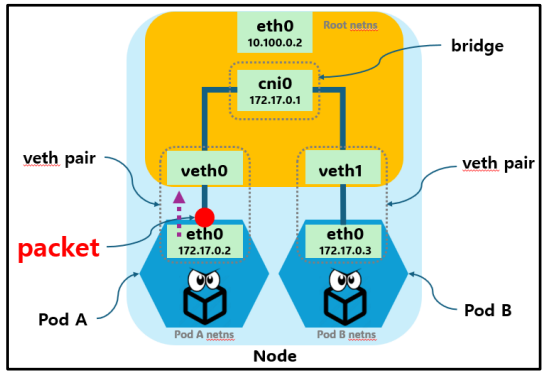
  - Bridge 생성: 호스트 시스템에서 가상 Ethernet Bridge를 생성하면, 이 Bridge는 여러 네트워크 인터페이스를 연결하는 가상 네트워크 스위치처럼 동작함

  - Interface 연결: 각 Pod가 생성되면, Pod의 가상 네트워크 인터페이스(예: veth 인터페이스)가 이 Bridge에 연결됨. 이렇게 연결된 인터페이스는 모두 같은 네트워크에 속하게 되며, 서로 통신할 수 있음

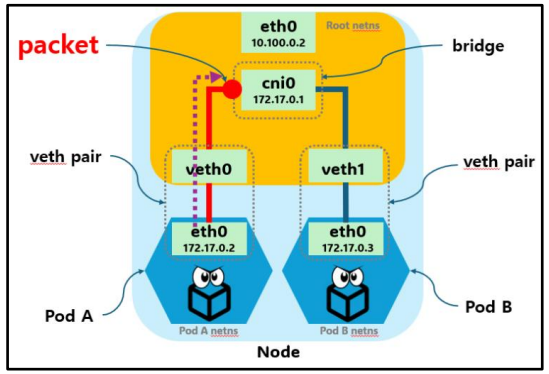
CNI0 인터페이스

- Kubernetes 클러스터에서 Pod 간의 네트워킹을 관리하는 가상 네트워크 브리지(bridge) 인터페이스
- 파드 네트워크 인터페이스와 호스트 네트워크를 연결해 주며, 파드 간 통신 및 외부 네트워크 접근을 가능함
- Pod 네트워크 연결: cni0 인터페이스는 각 노드(node)에서 실행되는 여러 Pod의 네트워크 인터페이스를 연결합니다. 이렇게 연결된 네트워크를 통해, 같은 노드 내의 Pod들은 서로 통신할 수 있음
- 브리지 역할: cni0는 가상 네트워크 브리지로서, 각 Pod의 네트워크 인터페이스(veth pair의 한쪽 끝)를 하나의 네트워크로 묶어주는 역할을 합니다. 이 브리지를 통해, 한 Pod에서 발생한 네트워크 트래픽이 다른 Pod로 전달될 수 있음
- Cni0 브리지는 Kubernetes의 네트워크 플러그인(CNI)을 통해 자동으로 관리되므로, 복잡한 네트워크 설정 없이도 Pod 간의 네트워킹을 쉽게 구성함

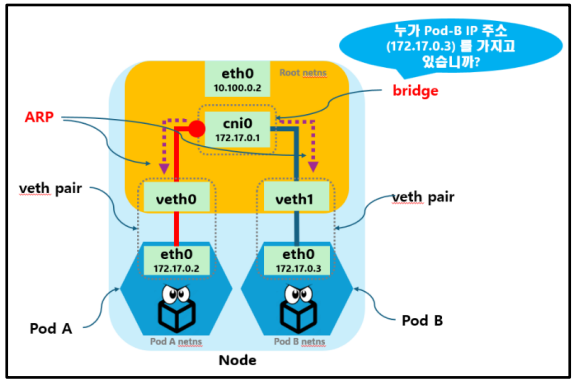
Pod-A 가 Pod-B 에게 메시지를 보내기



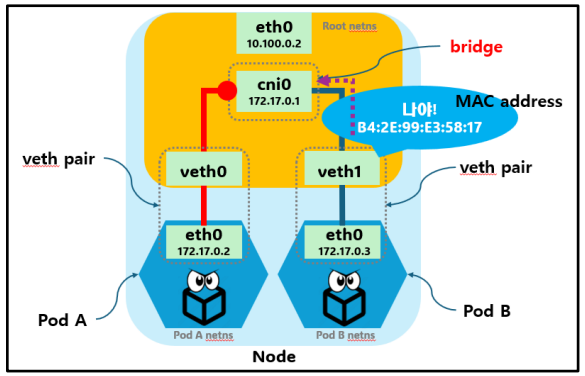
- Pod-B 은 Pod-A 의 network namespace 의 컨테이너가 아니므로 Pod-A 는 기본 인터페이스인 eth0 으로 패킷을 전달
- 이 인터페이스는 veth pair 에 연결되고 패킷은 root network namespace 로 전달



- root network namespace 로 전달 된 패킷은 virtual switch 역할을 하는 ethernet bridge 로 전달

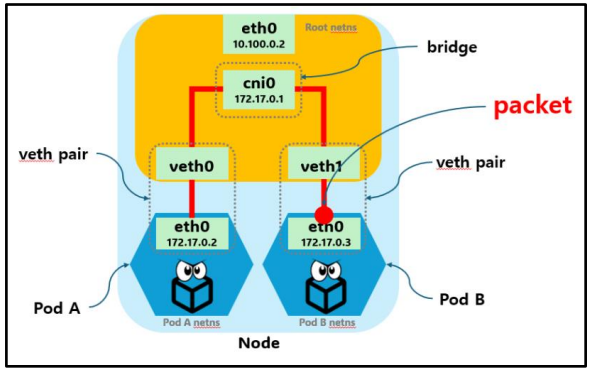


- virtual switch 역할을 하는 ethernet bridge 는 destination pod IP (Pod-B) 를 해당 MAC address 로 확인
- frame 이 bridge 에 도달하면 연결된 모든 장치에 ARP broadcast 가 전송



- bridge ARP cache (lookup table)에 저장된 Pod-B 를 연결하는 인터페이스의 MAC address 가 포함된 응답이 수신

- IP 및 MAC 주소 매핑이 저장되면 bridge 는 테이블에서 조회하여 packet 을 올바른 endpoint 로 전달



- packet 은 root namespace 의 Pod-B veth 에 도달하고, Pod-B namespace 내부의 eth0 인터페이스에 도달

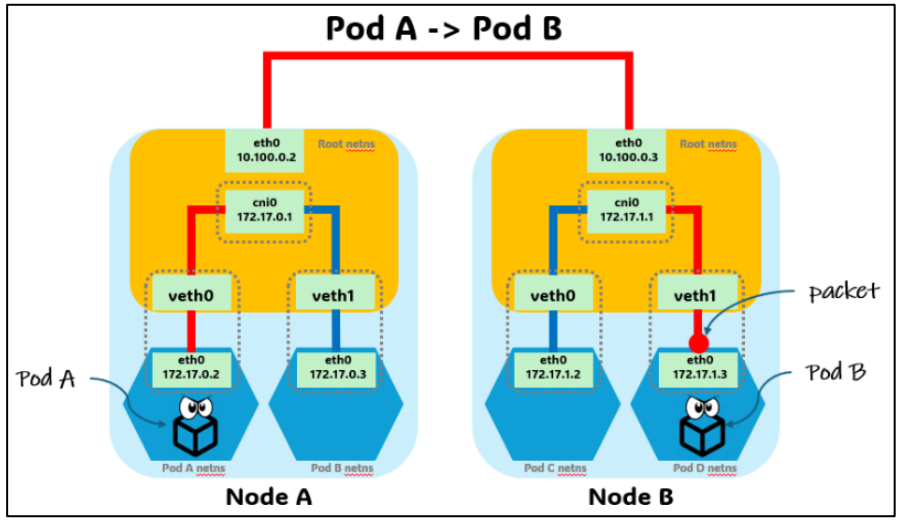
[그림 10] Pod-A 가 Pod-B 에게 메시지를 보내는 절차

다른 노드 간에 통신

- 패킷이 목적지에 도달하기 위해 노드 네트워크를 통과해야 하므로 파드가 다른 노드 간에 통신하려면 추가 홉(hop)이 필요
- 기본 "plain " 네트워킹 버전 또는 Flannel, Calico 등의 네트워크 플러그인 등 활용 가능함

Plain 네트워킹(기본)

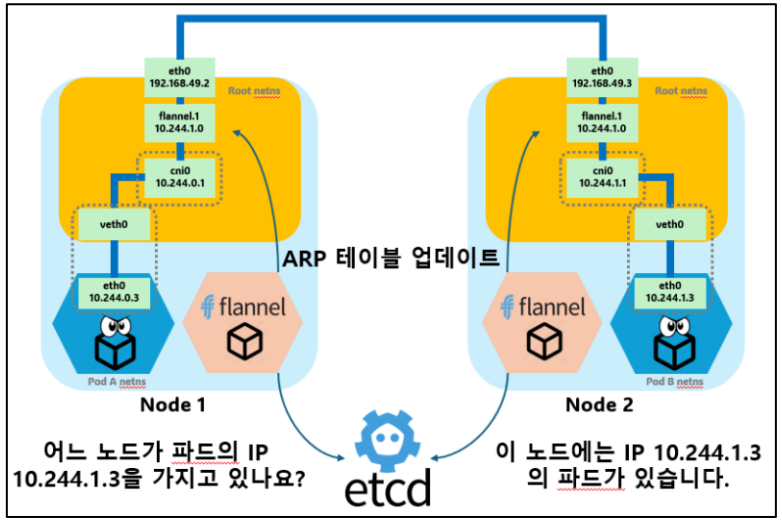
- 각 노드에 할당된 IP 주소를 사용하여 파드 간 통신을 가능하게 하는 간단한 네트워크 모델



[그림 11] Plain 네트워킹 구조

Flannel 네트워킹

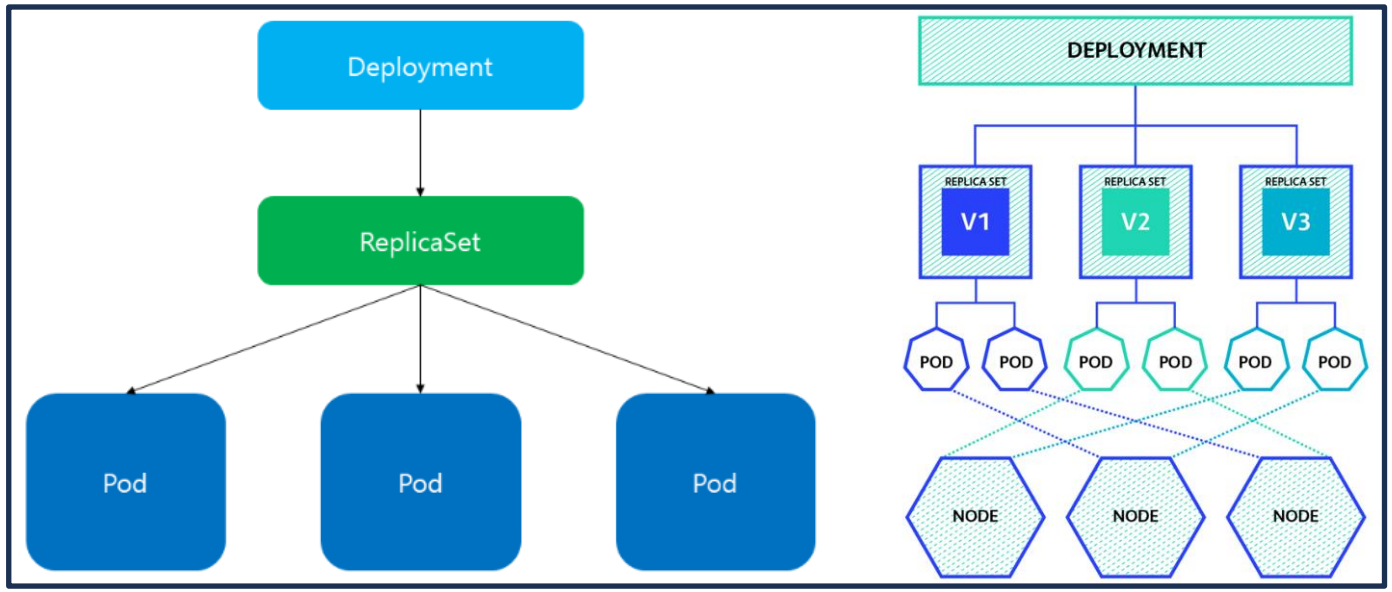
- Flannel은 Kubernetes에서 사용되는 오버레이 네트워크 플러그인
- Flannel은 각 노드에 가상 네트워크 인터페이스를 생성하고, 이 인터페이스를 통해 노드 간에 트래픽을 주고받음
- 각 노드에서 Flannel 데몬은 분산 데이터베이스의 IP 주소 할당을 동기화
- 다른 인스턴스는 이 데이터베이스를 쿼리하여 해당 패킷을 전송할 위치를 결정



[그림 12] Fannel 네트워킹 구조

Deployment

- ReplicaSet의 상위 개념으로, Pod와 ReplicaSet에 대한 배포를 관리
- 운영 중에 애플리케이션의 새 버전을 배포해야 하거나 부하가 증가하면서 Pod를 추가하는 등 여러 가지 동작을 Deployment로 관리
- 배포에 대한 이력을 관리하는데 만약 배포한 새 버전의 문제가 생긴 경우 Deployment를 통해 쉽게 이전 버전으로 롤백 할 수 있음
- 쿠버네티스로 서비스를 운영하는 상황이라면 ReplicaSet 만으로 운영하기보다는 대부분 Deployment 단위로 Pod와 ReplicaSet을 관리하여 운영함
- Deployment = ReplicaSet + Pod + history



[그림 13] Deployment 구조

Deployment-생성

- .spec.replicas
  - 생성할 Pod의 수를 지정
- .spec.selector
  - 식별할 Pod를 지정
  - .spec.selector.matchLables를 통해 식별할 Pod의 라벨을 지정
- .spec.template
  - Pod의 구성을 정의

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: demo-container
          image: nginx:latest
```

[그림 14] deployment.yaml

Deployment-생성

- YAML 파일을 구성한 후 다음 명령어로 Deployment 생성

```
$ kubectl apply -f deployment.yaml
deployment.apps/demo-deployment created
```

[그림 15] command : kubectl apply –f deployment.yaml

- 다음 명령어로 생성된 Deployment를 확인

```
$ kubectl get deploy
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
demo-deployment   2/2     2            2           4m5s
```

[그림 16] command : kubectl get deploy

- Deployment를 생성하게 되면 Deployment의 이름을 포함하고있는 ReplicaSet이 생성

```
$ kubectl get replicaset
NAME                         DESIRED   CURRENT   READY   AGE
demo-deployment-5dd594bdcb   2         2         2       10s
```

[그림 17] command : kubectl get replicaset

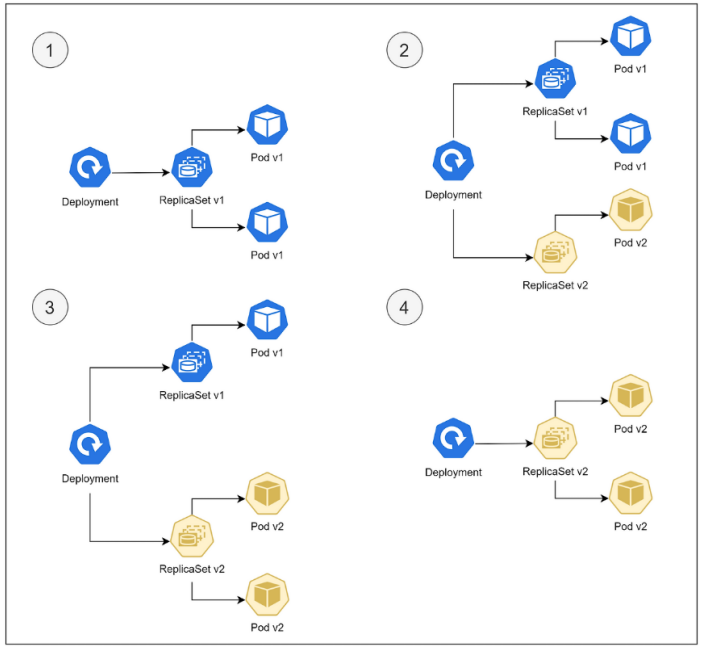
- Deployment에 의해 생성된 Pod는 ReplicaSet의 이름을 가지고 있는 것 확인

```
$ kubectl get pod
NAME                               READY   STATUS    RESTARTS   AGE
demo-deployment-5dd594bdcb-7h8tg   1/1     Running   0          31s
demo-deployment-5dd594bdcb-rjrx2   1/1     Running   0          31s
```

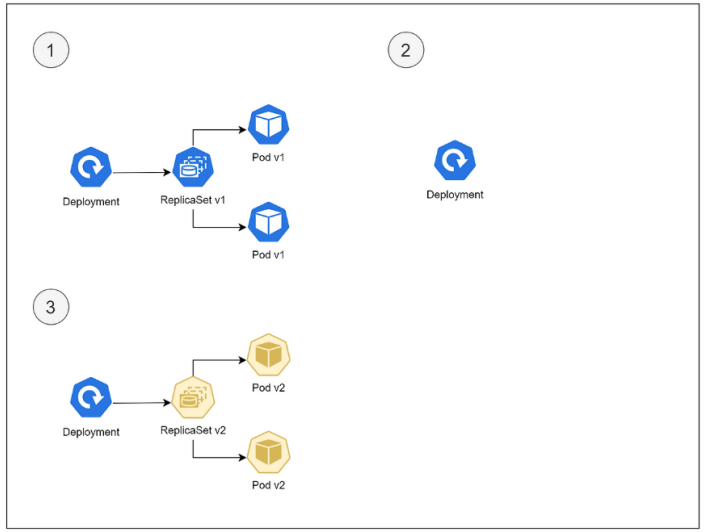
[그림 18] command : kubectl get pod

Deployment-업데이트

- Deployment의 업데이트 전략에는 크게 두 가지
  - RollingUpdate
  - Recreate



[그림 19] RollingUpdate



[그림 20] Recreate

Deployment-RollingUpdate

- RollingUpdate 배포 전략은 무중단 배포 전략으로 기존 버전의 Pod를 하나씩 삭제하고 새 버전의 Pod를 생성하면서 순차적으로 교체하는 방법
- Pod가 순차적으로 교체되기 때문에 다운타임이 발생하지 않는다는 장점이 있지만, 이전 버전의 Pod와 새 버전의 Pod가 공존하는 시간이 발생

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-deployment
spec:
  replicas: 10
  selector:
    matchLabels:
      app: nginx
  strategy:
# .spec.strategy.type을 RollingUpdate로 지정 생략시 기본값으로 RollingUpdate로 지정됨
    type: RollingUpdate rollingUpdate:
      maxSurge: 3
      maxUnavailable: 2
# Pod의 Status가 Ready가 될때까지의 최소대기시간, Pod의 변화를 관찰하기위해 지정해줌
  minReadySeconds: 10
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: demo-container
        image: nginx:latest
```

[그림 21] RollingUpdate 구성 yaml

Rolling Update의 동작 과정

- v2 버전의 ReplicaSet을 생성한다. 이때 replicas는 (maxSurge + maxUnavailable)이 되고 해당 수치만큼 Pod를 생성
- v1 버전의 ReplicaSet의 replicas가 (replicas - maxUnavailable)로 변경되고, 해당 수치만큼 Pod를 제거
- 배포가 진행되는 동안 v1 버전과 v2 버전에 트래픽이 분산
- v2 버전의 replicas를 1 증가시키고, v1 버전의 replicas를 1 감소 (v2 버전의 replicas가 템플릿에 정의된 replicas와 일치할 때까지 반복)
- v1 버전의 replicas를 0으로 변경한 후 남은 Pod를 삭제

Deployment-Recreate

- Recreate 배포 전략은 가장 단순한 배포 전략으로, 기존 버전의 Pod를 모두 삭제한 후 새 버전의 Pod를 생성하는 방법
- Recreate 배포 전략은 단순하지만 위 그림처럼 새 버전의 ReplicaSet이 생성되기 전까지 다운타임이 발생할 수 있음

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-deployment
spec:
  replicas: 10
  selector:
    matchLabels:
      app: nginx
  strategy:
    type: Recreate # .spec.strategy.type을 Recreate로 지정
  minReadySeconds: 10
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: demo-container
          image: nginx:latest
```

[그림 22] Recreate 구성 yaml

Recreate의 동작 과정

- v1 버전의 ReplicaSet의 replicas를 0으로 변경
- v1 버전의 Pod가 제거
- v2 버전의 ReplicaSet이 생성
- v2 버전의 Pod가 생성

Services 란?

- Service는 클러스터 외부로부터 요청을 받을 수 있게 IP를 노출하는 역할을 하는 리소스
- 그리고 Deployment로 Pod를 수평확장하는 상황에, 트래픽을 적절히 분산시키기는 역할을 할 수 있는 리소스
- 근본적으로 Pod는 클러스터 내부에서만 접근이 가능하며, 클러스터 외부로 노출되어 있지 않음
- 따라서 클러스터 외부에서 해당 Pod에 직접 요청할 수가 없음
- Pod를 외부로 노출시키려면 포트포워딩을 이용하거나 쿠버네티스 Service 오브젝트를 이용해야 함
- 다만 포트포워딩은 테스팅 목적이 강하므로 Service 리소스를 활용할 줄 알아야 한함

Services 기능

- 여러 Pod에 대해 클러스터 내에서 사용 가능한 고유 도메인을 부여함 (DNS 시스템)
  - 클러스터 내부 etcd를 통해서 도메인 관리를 수행
- 여러 Pod에 대한 요청을 분산하는 로드밸런서 기능을 수행
  - L4 즉, IP/Port 계층 기반으로 로드 밸런서 기능을 수행하게 됨
- 일반적으로 실무에서는 ClusterIP 타입의 Service와 함께 Ingress 리소스를 사용하여 외부 트래픽을 처리함
  - Ingress을 이용하면 L7 즉, 애플리케이션 계층에서 로드 밸런서 기능을 수행할 수 있음
- 참고로 Pod의 IP는 항상 변할 수 있음
  - Pod는 언제든지 생성되거나 삭제될 수 있는 비영구적인 리소스
  - 따라서 가변적인 IP가 아닌 다른 방식으로 Pod에 접근할 수 있어야 하는데, Service가 그 역할을 함

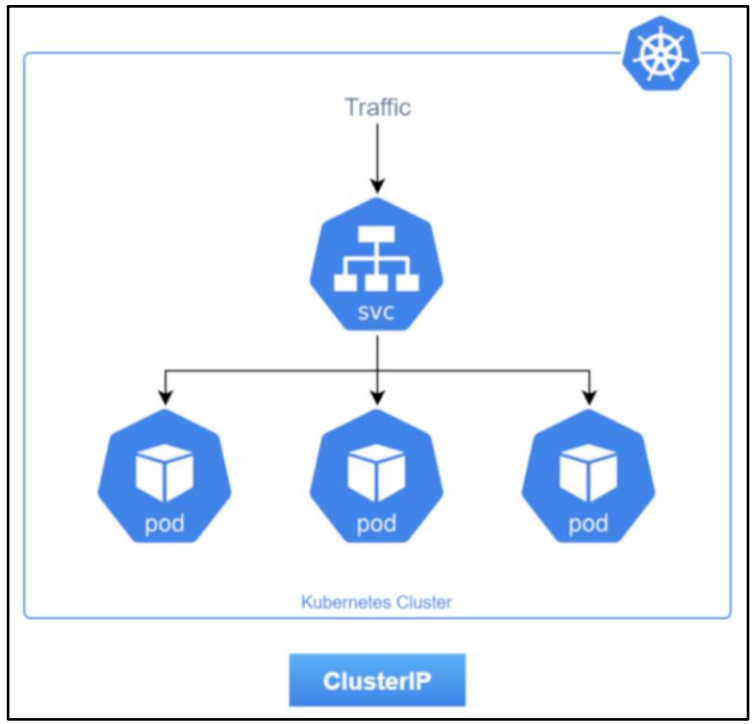
Service 원리

- Pod의 경우에는 오토스케일링과 같은 동적으로 생성/삭제 및 재시작 되면서 그 IP가 바뀌기 때문에, Service에서 Pod의 목록을 필터 할 때 IP 주소를 이용하는 것은 어렵움
  - 그래서 사용하는 것이 label과 label selector라는 개념
- spec.metadata 부분에 Pod 생성 시 사용할 label을 정의
- spec.selector.machLabels 부분에는 Service 생성 시 어떤 Pod들을 Service로 묶을 것인지 label Selector를 정의함
- Service를 생성하면, label Selector에서 특정 label을 가진 Pod들만 탐지함
  - Service는 이렇게 필터 된 Pod의 IP들을 엔드포인트로 묶어 관리하게 됨
  - 그래서 하나의 Service를 통해 여러 Pod에 로드밸런싱이 이루어질 수 있음
- 그리고 Service를 생성하면 Service 이름으로 DNS가 생성되는데, 해당 DNS 이름으로 트래픽이 들어오면 여러 Pod의 엔드포인트로 로드밸런싱 됨

Service Type

타입	설명
ClusterIP	• 가장 기본이 되는 Service 타입이며, 클러스터 내부 통신만 가능하고 외부 트래픽은 받을 수 없음 • 클러스터 내부에서 Service에 요청을 보낼 때, Service가 관리하는 Pod들에게 로드밸런싱 하는 역할을 함
NodePort	• 클러스터 내부 및 외부 통신이 가능한 Service 타입 • NodePort는 외부 트래픽을 전달을 받을 수 있고, NodePort는 ClusetIP를 wrapping 하는 방식이기 때문에 종장의 흐름은 결국 ClusetIP 비슷한 방식으로 이루어짐 • NodePort는 이름 그대로 노드의 포트를 사용함(port : 30000-32767) • 클러스터를 구성하는 각각의 Node에 동일한 포트를 열게 되는데, 이렇게 열린 포트를 통해서 Node마다 외부 트래픽을 받아서 -> ClusetIP로 모인 후 다시 로드를 분산시키는 방식
LoadBalancer	• LoadBalancer는 기본적으로 외부에 존재하며, 보통 클라우드 프로바이더와 함께 사용되어 외부 트래픽을 받는 역할을 받음 • 받은 트래픽을 각각의 Service로 전달해서 L4 분배가 일어나게 됨 • 역시 마찬가지로 흐름은 처음엔 LoadBalancer를 통하고, 이후엔 NodePort를 거쳐 ClusterIP로 이어지기 때문에 해당 기능을 모두 사용할 수 있음 • 클라우드 프로바이더를 사용하는 경우 클라우드 로드밸런서를 사용하여 외부로 노출시킴
ExternalName	위 3가지와 전혀 다른 Service 타입이라 할 수 있다. 다른 타입이 트래픽을 받기 위한 용도였다면 ExternalName 타입은 외부로 나가는 트래픽을 변환하기 위한 용도 ExternalName을 통해 a.b.com이라는 도메인 트래픽을 클러스터 내부에서는 a.b로 호출을 할 수 있게 해줌 (즉, 도메인 이름을 변환하여 연결해 주는 역할)

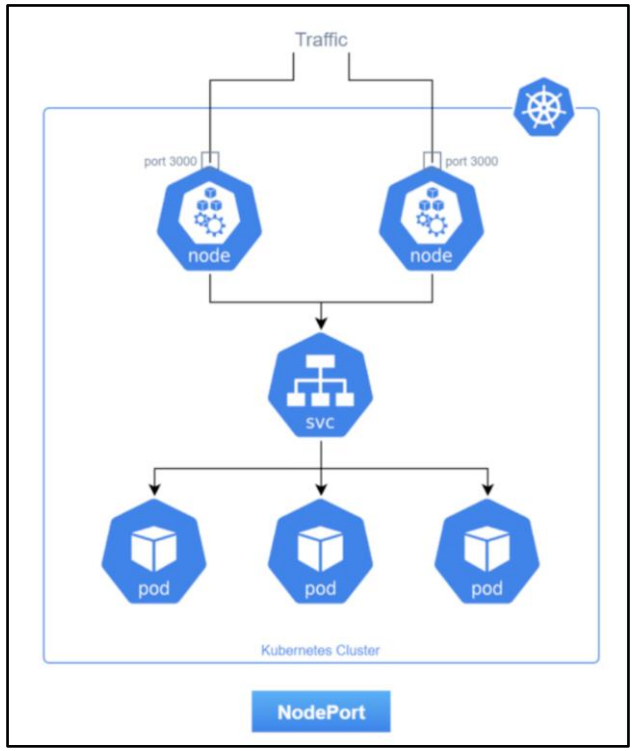
[표 5] Service Type



[그림 23] Cluster 타입 구조

Service – Cluster Type

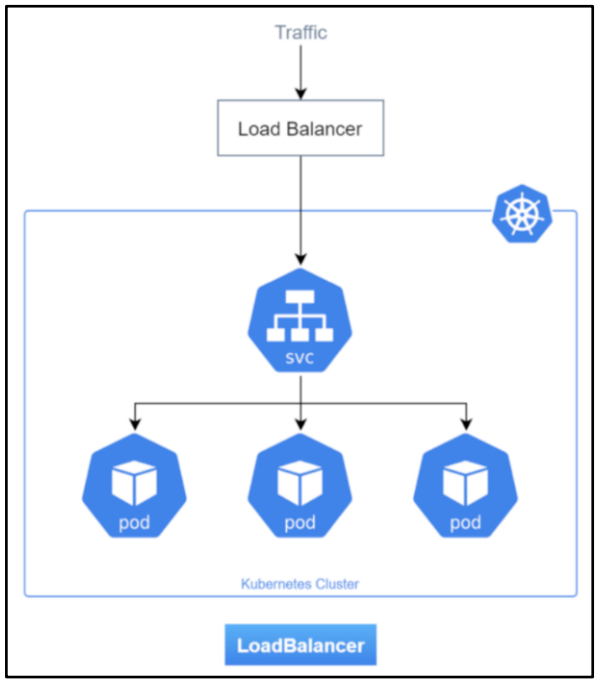
- Service API 리소스의 가장 기본적인 타입
- ClusterIP 타입의 Service는 쿠버네티스 클러스터 **내부 통신 목적으로만 사용 가능함**
- Pod에 부여되는 Pod IP는 내부 전용이라 외부에서 사용을 할 수 없음
- 반면 Service 오브젝트는 ClusterIP라는 가상 IP를 가짐
- ClusterIP가 가지게 되는 IP의 대역은 Cluster IP CIDR(Classless Inter-Domain Routing)이라고 부름
- Service는 Deployment와 같은 다른 리소스와 마찬가지로, Label Selector를 통해서 Service와 연결할 Pod 리스트를 관리함
- ClusterIP로 들어오는 요청에 대하여서는 **Pod에 L4 계층의 로드밸런싱**이 이루어짐
- ClusterIP 뿐만 아니라 이와 연결되는 내부 DNS을 통해 Service 이름을 부여하게 되는데, 이렇게 부여된 Service 이름을 통해서도 통신이 가능함
- 이러한 통신을 서비스 디스커버리라 하며, 배포된 애플리케이션이 어디에 있는지를 찾아내는 장치임



[그림 24] NodePort 타입 구조

Service – NodePort Type

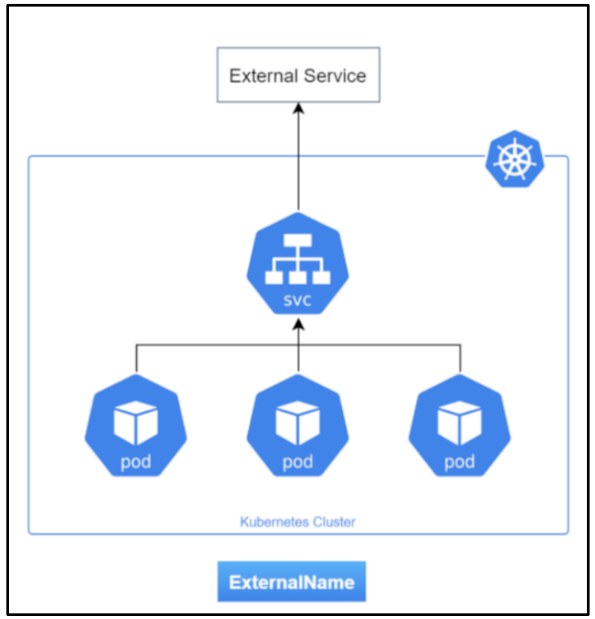
- NodePort는 Service를 외부에 노출할 수 있도록 하는 타입
- NodePort는 ClusterIP 타입 서비스를 한 번 더 감싸서 만들어진 것이므로, NodePort를 사용하더라도 ClusterIP를 동일하게 사용 가능함
- NodePort Service는 모든 쿠버네티스 노드의 동일 포트를 개방함
- 실제로 해당 포트로 트래픽이 들어오면, 결국 ClusterIP로 전달되어 Pod로 로드밸런싱이 이루어짐
- 즉, ClusterIP에 Service 포트와 Pod IP가 사용할 공통의 타깃 포트를 지정하는 설정은 그대로 있고, 거기에 모든 Node IP에 공통의 NodePort를 지정하여 오픈하는 설정이 추가됨



[그림 25] LoadBlancer 타입 구조

Service – LoadBalancer Type

- LoadBalancer 타입의 Service 리소스는 클라우드 프로바이더에서 제공하는 로드밸런서와 연동되는 기능
- 클라우드 프로바이더에서 제공하는 로드밸런서를 동적으로 생성하는 방식
- 아쉬운 건 AWS, GCP 등과 같은 클라우드 환경이 아니라, minikube 같은 개발 환경의 클러스터에서는 로드밸런서 기능 이용이 불가능함
- 하지만 MetalLB 같은 기술 등을 사용하면 로컬 환경이나 온프레미스 환경에서도 LoadBalancer 타입 사용이 가능함
- LoadBalancer 타입 Service는 NodePort 타입 Service를 한 번 더 감싸서 만들어짐
  - 따라서 LoadBalancer Service도 ClusterIP 사용이 가능함
  - LoadBalancer Service를 통해 만들어진 로드밸런서는 NodePort를 타겟 그룹으로 생성함.
  - NodePort로 들어온 요청은 실제로 ClusterIP로 전달되어 Pod로 포워딩됨
  - NodePort랑 비교하면, 처음 트래픽 진입 시점에 과정 하나가 더 생김
- NodePort의 단점은 노드가 사라졌을 때 자동으로 다른 노드를 통해 접근이 불가능함
  - 예를 들어, 3개의 노드가 있다면 3개 중에 아무 노드로 접근해도 NodePort로 연결할 수 있지만 어떤 노드가 살아 있는지는 알 수가 없음
- 자동으로 살아 있는 노드에 접근하기 위해 모든 노드를 바라보는 Load Balancer가 필요함



[그림 26] ExternalName 타입 구조

Service – ExternalName Type

- ExternalName Service는 외부로 요청을 전달하는 목적의 서비스를 제공
  (기존의 다른 3가지 타입과 다른 성격을 가지고 있다.)
- 보통은 클러스터 외부에 존재하는 레거시 시스템을 쿠버네티스로 마이그레이션 해야 하는 상황에 활용
- 다만 실질적으로는 앞서 다룬 3가지 서비스 타입과 비교해 많이 사용되지는 않음
- 서비스가 파드를 가리키는 것이 아닌 외부 도메인을 가리키도록 구성이 가능함
- DNS의 CNAME 레코드와 동일한 역할을 수행하는 데 많이 쓰이며, 이는 도메인의 또 다른 alias 도메인을 만들어서 리다이렉트 시켜주는 기능을 뜻하는 것임

GKE(Google Kubernetes Engine)

- GKE는 Kubernetes 오픈소스 컨테이너 조정 플랫폼을 Google 관리형으로 구현한 환경
- Kubernetes는 Google이 사내 클러스터 관리 시스템인 Borg에서 대규모로 프로덕션 워크로드를 운영해온 수년간의 경험을 바탕으로 개발됨
- GKE를 사용하면 Google 인프라를 사용하여 자체 컨테이너화된 애플리케이션을 대규모로 배포하고 운영할 수 있음

GKE 장점	
플랫폼 관리	• 기본 제공되는 강화 및 자동으로 적용되는 권장사항 구성이 포함된 GKE Autopilot 모드의 완전 관리형 노드 • 보안, 안정성, 규정 준수를 개선하기 위한 출시 채널이 포함된 관리형 업그레이드 환경 • 비즈니스 요구사항 및 아키텍처 제약조건에 맞게 업그레이드 유형 및 범위를 구성할 수 있는 유연한 유지보수 기간 및 제외 • GKE Standard 모드에서 가용성을 최적화하고 중단을 관리할 수 있는 유연한 노드 업그레이드 전략 • Autopilot 모드 또는 Standard 모드의 노드 자동 프로비저닝을 사용하는 클러스터의 포드 수를 기준으로 한 노드의 자동 확장 • 노드 자동 복구로 노드 상태 및 가용성을 유지 관리 • 기본 제공 로깅 및 모니터링 • Google Cloud CI/CD 옵션을 Cloud Build 및 Cloud Deploy와 통합함

[표 6] GKE 장점

GKE 장점	
보안 상황 개선	• 앱용 강화 노드 운영체제: Container-Optimized OS • 보안 조치 기본 제공 • 새 GKE 버전으로 자동 업그레이드 • 보안 상태 대시보드를 사용하는 통합 보안 상황 모니터링 도구 • Google Cloud Observability와Google Cloud 로깅 및 모니터링 통합
비용 최적화	• Autopilot 모드에서는 실행 중인 포드에서 요청하는 컴퓨팅 리소스에 대한 요금만 부과 • GKE Standard 모드에서는 포드 요청에 관계없이 노드의 모든 리소스에 대한 요금이 청구 • Spot 포드에서 일괄 작업과 같은 내결함성 워크로드를 실행하여 비용을 절약 • Google에서 노드와 컨트롤 플레인을 모두 관리하므로 Autopilot 모드의 운영 오버헤드가 최소화
안정성과 가용성	• 99% 월별 업타임 SLO(Service-Level Objective) • Google에서 노드를 관리하므로 Autopilot 클러스터의 포드 수준 SLA 적용 • Autopilot 모드 및 리전 Standard 클러스터의 고가용성 컨트롤 플레인 및 워커 노드 • 지원 중단으로 인한 잠재적 워크로드 중단을 완화하기 위한 사전 예방적 모니터링 및 추천 • 멀티 클러스터 서비스 기능

[표 7] GKE 장점

GKE 작동방식

- GKE 환경은 그룹화되어 클러스터를 형성하는 Compute Engine 가상 머신(VM)인 노드로 구성
- 앱(워크로드라고도 함)을 컨테이너로 패키지화하며, 컨테이너 모음을 노드에 포드로 배포함
- Kubernetes API를 사용하여 관리, 확장, 모니터링을 포함해 워크로드와 상호작용함
- GKE에서 **Google이 컨트롤 플레인과 시스템 구성요소를 관리**함
- GKE를 실행하는 데 권장되는 방법인 **Autopilot 모드에서는 Google이 워커 노드도 관리**함
- Google이 구성요소 버전을 자동으로 업그레이드하여 안정성과 보안을 개선하고, 고가용성을 보장하고, 클러스터의 영구 스토리지에 저장된 데이터의 무결성을 보장함

GKE Autopilot 모드 vs Standard 모드

	Autopilot 모드	Standard 모드
운영책임	• Google이 대부분 관리함 (노드 관리 불필요)	• 사용자가 노드 및 인프라 관리
과금 방식	• Pod 단위 요금 (리소스 요청 기준)	• 노드 단위 요금 (VM 인스턴스 기준)
유연성	• 제한적 (노드 설정 불가, 일부 권한 제한)	• 유연함 (커스텀 머신 타입, OS 선택, 데몬셋 등 가능)
자동 스케일링	• 기본 제공 (노드 관리 없이 Pod 수요에 따라 자동 확장)	• 수동 또는 자동 설정 필요 (Cluster Autoscaler)
워크로드 종류	• 일반 앱, 백엔드 API, 빠른 개발 및 테스트에 적합	• 고성능, 커스텀 네트워크, GPU/TPU 등 특수 요구에 적합
노드 접근/제어	• 불가능 (노드에 SSH 불가, DaemonSet 불가)	• 가능 (SSH, Custom DaemonSet 설정 가능)
출시 시기	• 2021년 출시	• GKE 초기부터 제공

[표 8] GKE Mode

노드 풀(Node Pool)

- 노드 풀은 클러스터 내에서 구성이 모두 동일한 노드 그룹
- 노드 풀은 NodeConfig 사양을 사용함
- 풀의 각 노드에는 Kubernetes 노드 라벨인 cloud.google.com/gke-nodepool이 있으며, 이 라벨의 값은 노드 풀의 이름임
- Standard 모드 클러스터를 만들 때는 지정한 노드 수 및 노드 유형을 사용해서 클러스터의 첫 번째 노드 풀을 만듬
- 추후 다양한 크기와 유형의 노드 풀을 클러스터에 추가할 수 있으며, 특정 노드 풀의 모든 노드는 서로 동일함
- 커스텀 노드 풀은 추가 메모리 또는 추가 로컬 디스크 공간 등 다른 것보다 더 많은 리소스가 필요한 포드를 예약해야 할 경우에 유용함
- 포드 예약 위치를 더 세부적으로 제어해야 할 경우에는 노드 taint를 사용할 수 있음
- 전체 클러스터에 영향을 주지 않고 노드 풀 만들기, 업그레이드, 삭제를 개별적으로 수행할 수 있음
- 노드 풀에서 단일 노드를 구성할 수 없으며, 구성 변경사항은 노드 풀에 있는 모든 노드에 영향을 줌
- 노드를 추가 또는 삭제하여 클러스터에서 노드 풀 크기를 조절할 수 있음
- 기본적으로 모든 새 노드 풀은 컨트롤 플레인과 동일한 버전의 Kubernetes를 실행함
- 기존 노드 풀을 수동 또는 자동으로 업그레이드할 수 있음
- 클러스터의 각 노드 풀에서 여러 Kubernetes 노드 버전을 실행하고, 각 노드 풀을 독립적으로 업데이트하고, 특정 배포에 대해 다른 노드 풀을 대상으로 지정할 수 있음

AutoScaling-Pod

- **Horizontal Pod Autoscaler(HPA)**

  - pod에 부하가 늘어남에 따라 임계치와 비교해서 pod의 개수를 조절
  - kubectl autoscale deployment를 사용하면 적용되고 HPA가 적용되며, --cpu-percent, --min, --max 같은 값을 넘기면 측정되는 메트릭에 따라 오토스케일 됨
  - GKE에서 사용하는 HPA는 Pub/Sub 메시지, Google Monitoring 메트릭, Load balancer 트래픽과 같은 더 다양한 수치를 기준으로 임계점을 설정 가능

- **Vertical Pod Autoscaler(VPA)**

  - 경우에 따라서는 스케일 아웃만으로 부족할 수 있음
  - OutOfMemory로 컨테이너가 죽는다거나 cpu가 피크를 찍는다면 성능을 올려야 함
  - 이 경우 VPA가 스케일 업을 담당함
  - pod의 사용률을 모니터링해서 cpu, memory를 분석하고, 리소스가 부족하다고 판단될 경우 pod 리소스를 늘리고 재배치함
  - 자동으로 pod의 스케일 업도 가능하고, 필요하다면 적절한 수치를 제안하는 recommendation 필드만 참조하는 것도 의미가 있음

- **Multidimentional Pod Autoscaler(MPA)**

  - MPA는 GKE에만 존재하는 특수한 Autoscaling 기능
  - MPA는 수평적으로 확장하는 HPA와 수직적으로 확장하는 VPA의 전략을 혼합한 전략을 사용함
  - CPU를 기반으로는 수평적 확장을, Memory를 기반으로는 수직적 확장을 수행

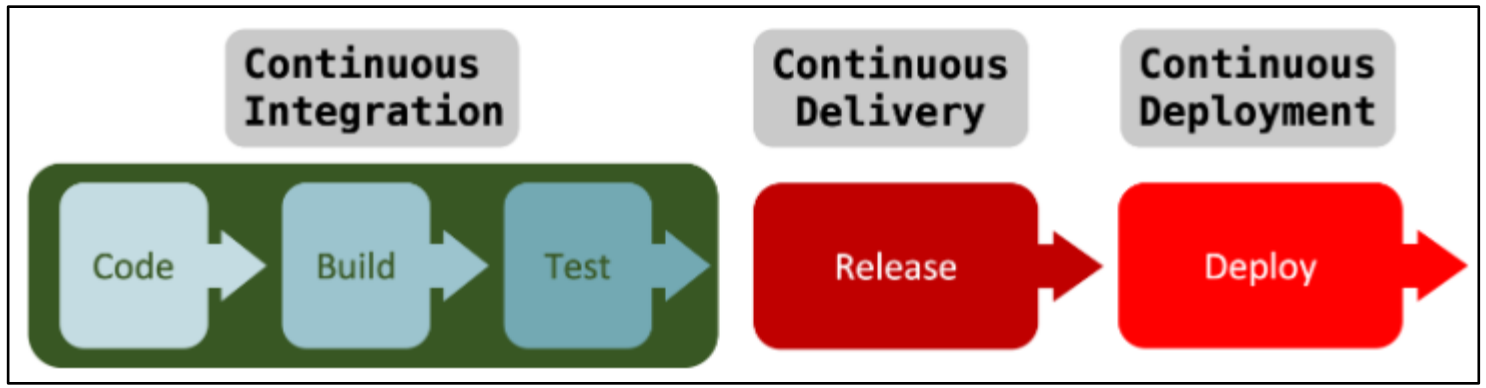
AutoScaling-Node

- **Cluster Autoscaler(CA)**
  - pod이 증가하는데 할당할 node가 부족한 경우 node pool에서 node를 추가함
  - 필요한 pod 수가 줄어들어 node가 비거나, 재배치로 인해 불필요한 노드가 생기면 node를 제거함

- **Node Auto Provisioning(NAP)**
  - 새로운 pod를 프로비저닝 해야 하는데, 수용 가능한 node가 없는 경우 새로운 node pool 추가

CI/CD 란?

- CI/CD는 애플리케이션 개발부터 배포까지 모든 단계를 자동화하여 효율적이고 빠르게 사용자에게 빈번히 배포할 수 있도록 하는 개념
  - CI는 지속적인 통합(Continuous Integration)이라는 의미로 작업한 코드를 주기적으로 빌드 -> 테스트 -> 병합하는 과정
  - CD는 지속적인 제공(Continuous Delivery)과 지속적인 배포(Continuous Deployment)라는 2개의 뜻으로 불림
  - 지속적인 제공은 '수동 배포'
  - 지속적인 배포는 '자동 배포'
  - CI를 거쳐 CD가 진행되기에 CI와 CD를 따로 말하지 않고 CI/CD라고 불림



[그림 27] CI/CD 구조

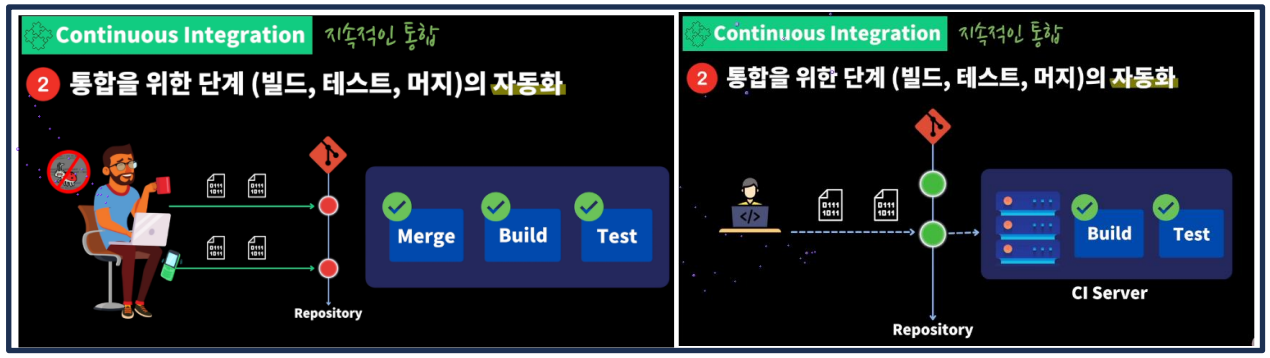
CI(Continuous Integration)

- 지속적인 통합이라는 의미로, 작업한 코드를 주기적으로 빌드 및 테스트하여 레포지토리에 통합(merge)
- 버그 수정, 새로운 기능 개발한 코드가 주기적으로 Main Repository에 병합되는 것을 의미함

[그림 28] CI : 코드 반영

1. 코드 변경사항을 주기적으로 빈번하게 merg

- 코드가 충돌하게 되고 오랫동안 꼬여버린 코드를 병합하는데 시간이 오래 걸림
- 코드 작성 시간보다 코드 충돌 해결 시간에 더 많은 시간을 할애해야 할 수도 있음
- 따라서 작은 단위로 개발하여 '주기적으로 빈번하게 머지 해야 함



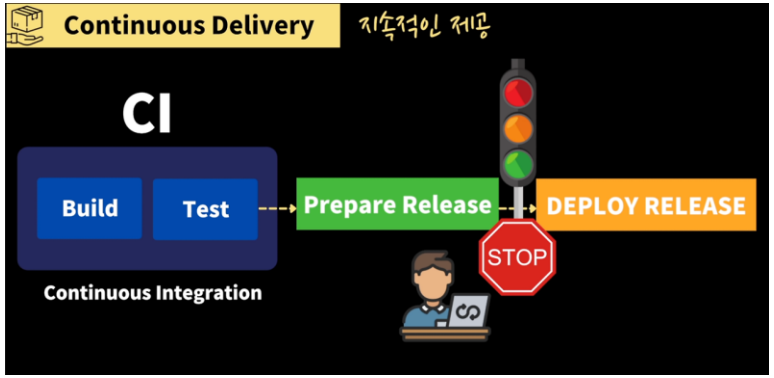
[그림 29] CI : 코드 통합 자동화

2. 통합 단계의 자동화(통합을 위한 단계)

- 개발자가 코드 수정 후 MR 혹은 PR을 요청
- 프로젝트에 작성된 CI Script를 기반으로 변경 및 추가된 코드를 자동으로 Build Test 진행 (Unit Test, Integration Test 등..)
- 이때 모두 성공 시 MR 혹은 PR이 성공, 과정 중 하나라도 실패 시 병합 거절(Close)

CD(Continuous Delivery & Continuous Deployment)

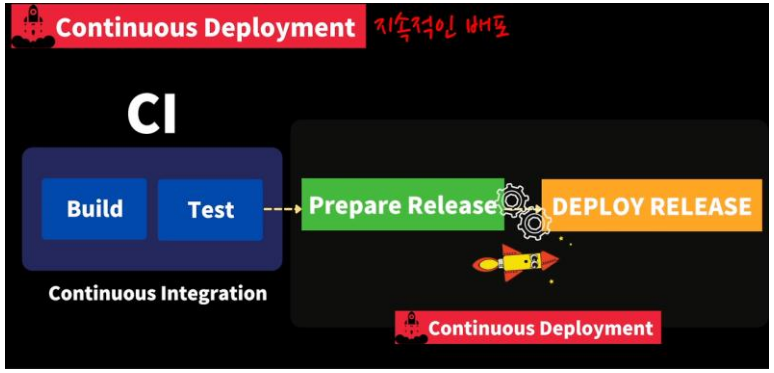
- 두 가지 모두 다 마지막 배포 단계에서 “어떻게 하면 자동화해서 배포를 만들 수 있을까?”를 고민하는 단계



[그림 30] CI : **CD(Continuous Delivery)**

CD(Continuous Delivery) 지속적인 제공

- CI 단계에서 빌드 되고 테스트된 후, 배포 준비 상태가 확인되면 개발자 혹은 검증팀이 수동으로 배포



[그림 31] CI : **CD(Continuous Deployment)**

CD(Continuous Deployment) 지속적인 배포

- CI 단계에서 빌드 되고 테스트된 후, 배포 준비 상태가 확인되면 자동으로 배포까지 진행

CI/CD 파이프라인

- 개발자가 작은 단위로 기능을 분리하여 주기적으로 Main Repository에 Merge
- 자동으로 Build
- 자동으로 Test 진행
- RELEASE 준비 상태
  - 수동 배포
  - 자동 배포

[그림 32] CI/CD 파이프라인

[그림 33] CI/CD 관련 툴

GCP Cloud Build 란?

- Cloud Build는 GCP의 인프라에서 빌드를 실행하는 서비스
- 다양한 저장소 또는 클라우드 스토리지 공간에서 소스 코드를 가져오고 사양에 맞게 빌드를 실행하고 Docker 컨테이너 또는 자바 아카이브와 같은 아티팩트를 생성할 수 있음

Cloud Build의 장점

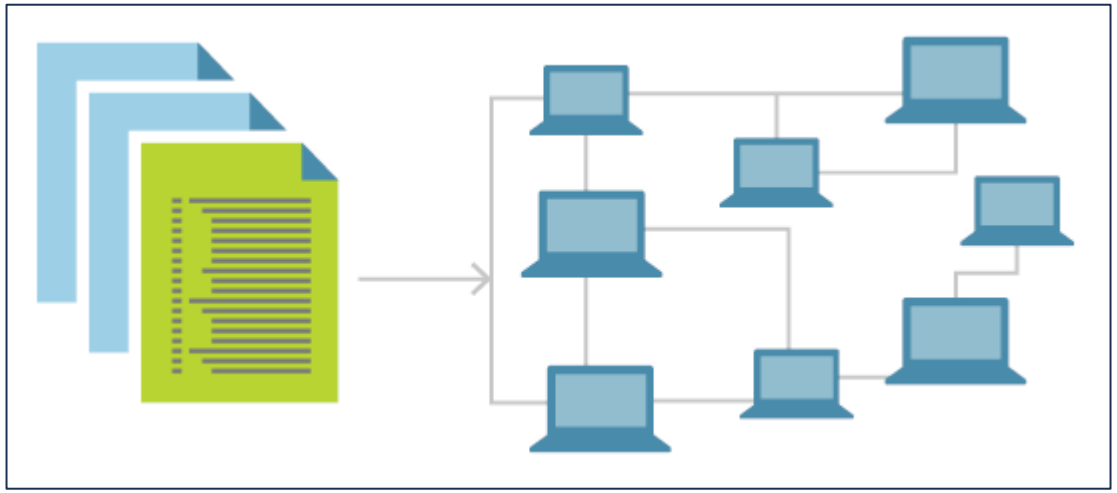
- 자바, Go, Node.js 등의 모든 프로그래밍 언어를 사용하여 신속하게 소프트웨어를 빌드
- 빌드, 테스트, 배포를 위한 커스텀 워크플로를 정의하는 작업을 완벽하게 제어
- VM, 서버리스, Kubernetes 또는 Firebase 등 다양한 환경에서 배포
- CI/CD 파이프라인의 일부로 정밀 보안 스캔을 수행함
- Maven, Gradle, Go, Bazel과 같은 도구를 통해 컨테이너 또는 컨테이너가 아닌 아티팩트로 소스를 패키징함

Cloud Build의 특징

- Docker를 지원
- 빌드 과정을 모니터링할 수 있음
- 컨테이너 이미지의 패키지 취약점을 자동으로 파악함
- 로컬 또는 클라우드에서 빌드가 가능함

Infrastructure as Code(IaC)

- IaC(Infrastructure as Code)는 DevOps 방법론 및 버전 관리와 설명 모델을 사용하여 네트워크, 가상 머신, 부하 분산 장치 및 연결 토폴로지와 같은 인프라를 정의하고 배포함
- 동일한 소스 코드가 항상 동일한 이진 파일을 생성하는 것처럼 IaC 모델은 배포할 때마다 동일한 환경을 생성함
- IaC는 DevOps의 핵심 사례이자 지속적인 업데이트의 구성 요소
- IaC를 통해 DevOps 팀은 통합된 사례 및 도구 집합과 협력하여 애플리케이션과 지원 인프라를 신속하고 안정적으로 대규모로 제공



[그림 34] IaC

GCP Infrastructure Manager

인프라를 코드(IaC)로 관리하는 GCP의 선언형 리소스 관리 도구

- GCP에서 리소스를 코드 기반으로 선언하고 배포하는 도구
- 기존의 Terraform 또는 Deployment Manager (2025년 12월 31일 종료)를 대체하거나 보완
- GCP의 Config Connector를 기반으로 하고 있음
- YAML 또는 Blueprint 형태로 리소스 정의
- Git 기반 인프라 배포 자동화 가능

```yaml
# simple-gcs.yaml
resources:
- name: example-bucket
  type: gcp-types/storage-v1:buckets
  properties:
    name: my-infra-bucket
    location: US
    storageClass: STANDARD
```

[그림 35] 예제 Blueprint

```bash
gcloud beta infra-manager deployments create my-deployment \
  --source=./simple-gcs.yaml \
  --preview
```

[그림 36] 배포 명령