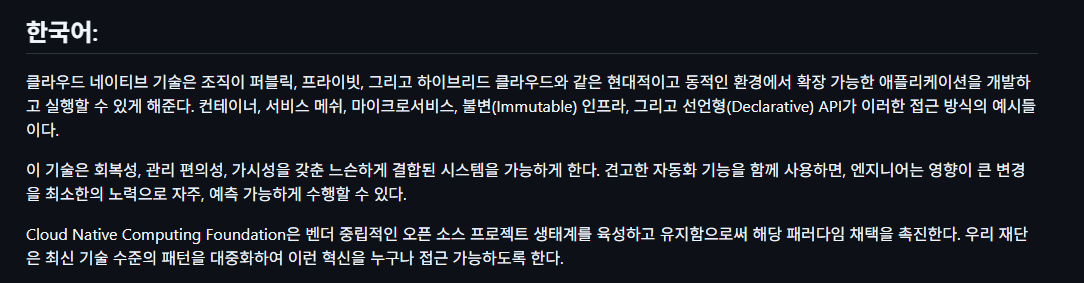
# 클라우드 네이티브란?

* 클라우드의 장점을 최대한 활용하여 정보 시스템을 구축 및 실행하는 환경
* 클라우드 네이티브 기술, 애플리케이션, 아키텍처, 개발방법론, 조직, 프로세스 등 다양한 용어와 결합하여 다양한 의미로 사용
* 쿠버네티스는 클라우드 네이티브 구성요소를 완전히 수행할 수 있는 최고의 플랫폼



출처: [toc/DEFINITION.md at main · cncf/toc · GitHub](https://github.com/cncf/toc/blob/main/DEFINITION.md)

# 기존 애플리케이션과 클라우드 네이티브 애플리케이션의 차이점

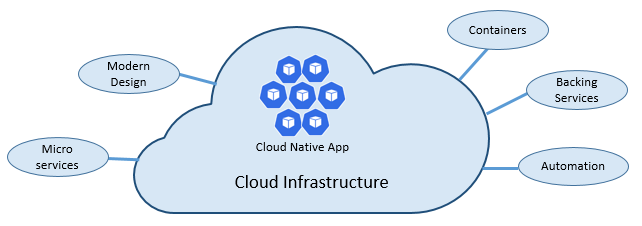
* 기존 애플리케이션은 장기간에 걸쳐 긴말하게 결합된 모놀리식 기반 위에서 동작
* 클라우드 네이티브 애플리케이션은 소규모 서비스 단위의 마이크로서비스로 구성되며 가상 컨테이너 환경에서 동작되도록 설계되고 구현



# 클라우드 네이티브 구성 요소



* 마이크로 서비스: 서비스를 각각 고유한 논리, 상태 및 데이터가 있는 독립적인 서비스로 기능을 분리
* 컨테이너: 코드, 해당 의존성 및 런타임은 컨테이너 이미지라는 이진 파일로 패키지
* 최신 디자인: 클라우드 기반 애플리케이션을 생성하기 위해 널리 허용되는 방법 12단계 애플리케이션(코드 베이스, 의존성, 구성, 지원서비스, 빌드/릴리즈/실행, 프로제스, 동시성, 일회성, Dev/Prod 패리티, 로깅, 관리 프로세스)
* 서비스 지원: 데이터 저장소, 메시지 브로커, 모니터링 및 id 서비스와 같은 다양한 보조 리소스 구성
* 자동화: 스크립트를 활용한 인프라, CD/CD 배포 자동화

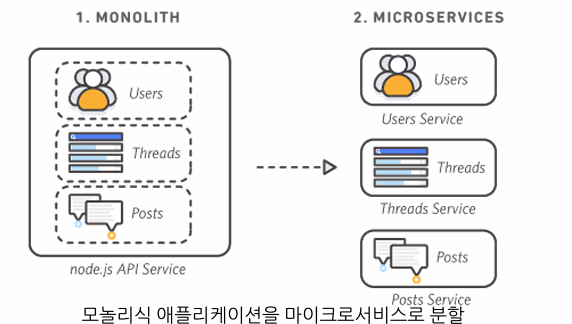


출처: [클라우드 네이티브란? - .NET | Microsoft Learn](https://learn.microsoft.com/ko-kr/dotnet/architecture/cloud-native/definition)

# 모놀리식 아키텍처 & 마이크로서비스 아키텍처

## 모놀리식 아키텍처(Monolithic architecture)

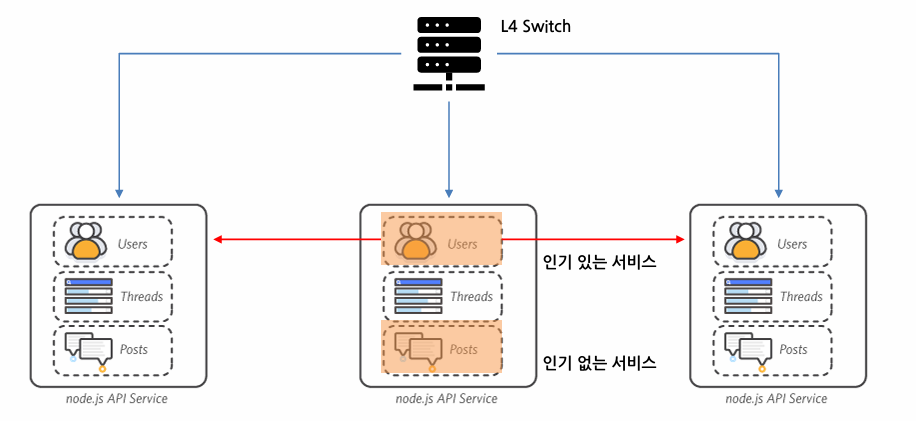
* 전통적인 아키텍처, 기존에 사용하던 서비스 방법
* 서비스가 하나의 애플리케이션으로 돌아가는 구조
* 기존의 개발 방식을 사용해 개발하여 간단히 배포
* 하나의 서비스 또는 어플리케이션이 하나의 거대한 아키텍처
* 다양한 기능을 동작하는 서비스를 서버에서 실행하여 서비스



출처: [마이크로서비스란 무엇입니까? | AWS](https://aws.amazon.com/ko/microservices/)

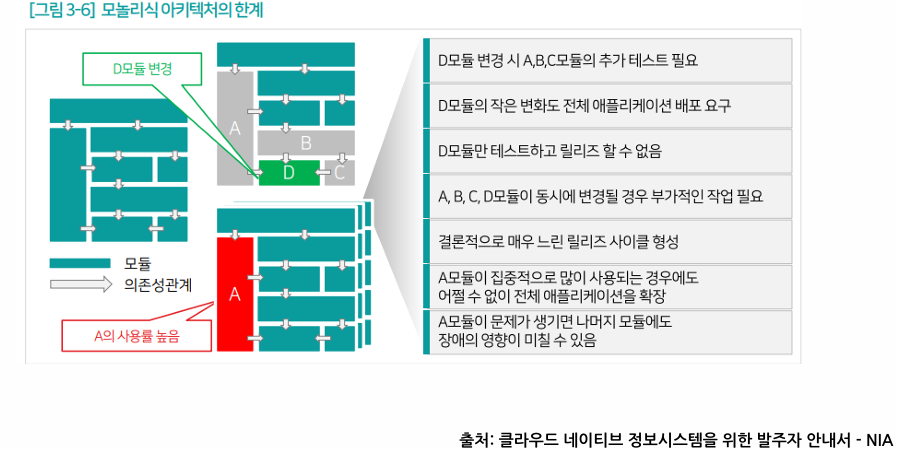
### 단점

* 모놀리식 서비스 아키텍처를 스케일아웃
* 기존의 애플리케이션을 그대로 복제하여 로드밸런싱
* 불필요한 서비스까지 모두 복제



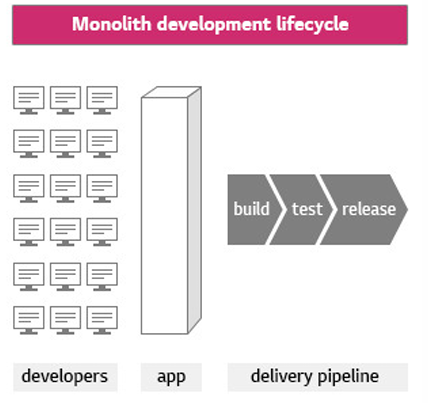
1. 종속적인 라이브러리의 충돌

* 각각의 기능들은 서로 다른 기능을 제공하여 버전의 종속성을 필요한 경우가 존재
* 각 기능의 따른 라이브러리를 매 업데이트마다 관리하기 매우 어려움



1. 조금만 수정해도 전체 빌드 및 배포 필요

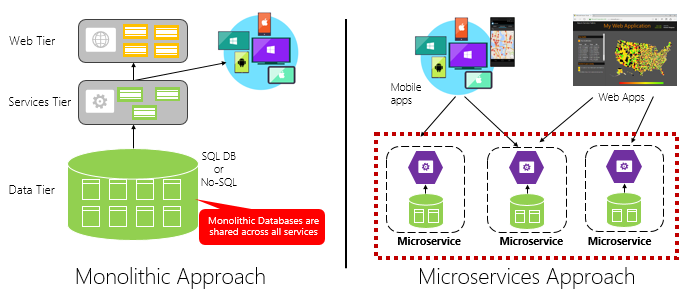
* 소스코드 전체가 하나로써 동작하기 때문에 작은 수정만 있더라도 전체를 빌드하여 다시 배포해야 함
* 프로그램의 크기가 어느정도 커짐면 한 번만 컴파일해서 전체 테스트를 돌려도 30분 내지 수 시간 소모
* 하루에 버그가 여러 개 순차적으로 발견된다면? 막막함



출처: [TISTORY](https://blog.lgcns.com/1278)

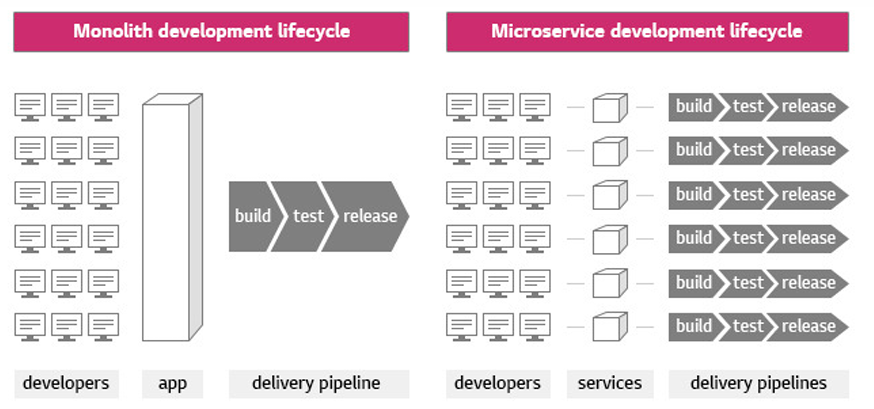
## 마이크로서비스 아키텍처

* 모놀리식 아키텍처의 대안으로 반대되는 개념
* 애플리케이션의 각각의 기능을 분리하여 개발 및 관리
* 마이크로 서비스 장점
  + 개발자가 특정 비즈니스 로직에 대해서만 집중하여 개발 가능해 빠른 개발
  + 개별 서비스 단위로 개발, 패키징, 빌드, 테스트, 배포로 각 서비스마다 유연한 스케줄
  + 서비스 단위로 스케일아웃이 가능하여 불필요한 서비스는 줄이고 더 많은 자원이 필요한 서비스는 확장가능



출처: [클라우드 네이티브란? - .NET | Microsoft Learn](https://learn.microsoft.com/ko-kr/dotnet/architecture/cloud-native/definition)

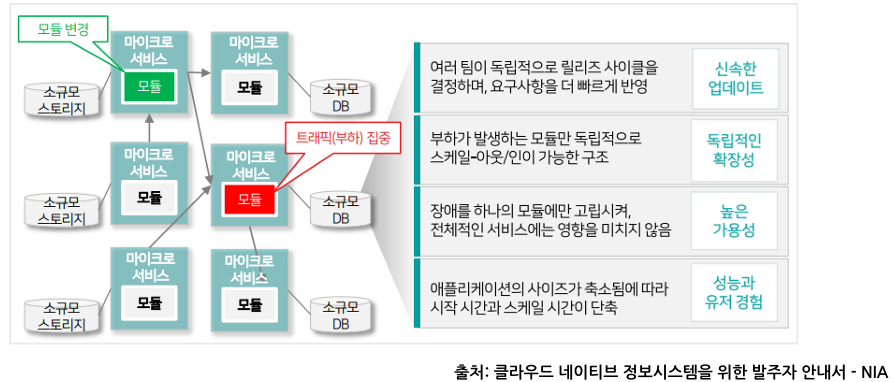
### 모놀리식 라이프 사이클과 마이크로 서비스 라이프 사이클 비교



출처: [TISTORY](https://blog.lgcns.com/1278)

### 장점

* 서비스 단위 고효율 저비용 Scale-Out 구조
  + 서비스 단위로 스케일아웃이 가능하여 불필요한 서비스는 줄이고 더 많은 자원이 필요한 서비스는 확장 가능
* 라이브러리 종속성 고민 X



### 단점

* 분산 시스템 환경에서 Transaction 보장, 테스트, 배포, 관리 복잡

# DevOps 모델 개념

* 데브옵스는 소프트웨어 개발과 IT 운영을 결합한 합성어
* 기존의 분리된 소프트웨어 개발팀과 IT 운영팀의 협업으로 전체 라이프사이클을 함께 관리할 수 있자는 일종의 철학 또는 운동
* 소프트웨어 개발팀과 IT팀이 더 빠르고 안정적으로 소프트웨어를 빌드, 릴리즈할 수 있도록 두 팀 간의 프로세스를 자동화하는 일련의 과정

## 이점

* 속도: 서로의 업무에 대해 더 잘 이해하며 좀더 신속하게 사용자에게 필요한 업데이트를 수행 가능. 개발자는 소비자가 무엇을 원하는지, 운영자는 애플리케이션을 제공하는데 해결할 문제를 인지할 수 있다.
* 신속한 제공: 컨테이너와 마이크로서비스를 사용하면 더 자주 빠르게 릴리즈하는 것이 가능. 더 빠르게 좋은 기능을 제공할 수 있으며 개발자가 운영에 필요한 인프라와 하드웨어에 대해 잘 몰라도 릴리즈가 가능하다.
* 개발과 릴리즈가 편해지므로 안정성이 확보, 협업 강화

## 개발자와 시스템 관리자의 역할 분담

### 개발자의 관점

* 새로운 기능을 만들어 사용자 경험을 개선하는 것을 좋아함
* 기본 운영체제의 보안 패치나 이와 관련된 모든 것이 최선인지 확인하는 것을 시스템 관리자에게 맡기려 함

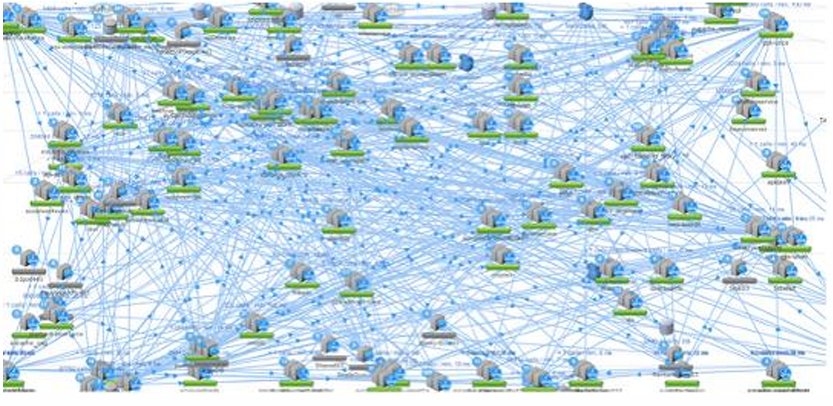
### 시스템 관리자의 관점

* 제품 배포와 운영하는 하드웨어 인프라를 담당하며 시스템보안, 활용, 개발자의 우선순위가 높지 않은 측면에 신경을 씀
* 운영 담당자는 모든 애플리케이션 구성 요소의 암묵적 상호 의존성에 대처하기를 원하지 않음
* 기본 운영체제나 인프라를 변경했을 때 애플리케이션 전체 동작에 어떤 영향을 미칠지는 생각치 못함

# 마이크로서비스 성공 사례

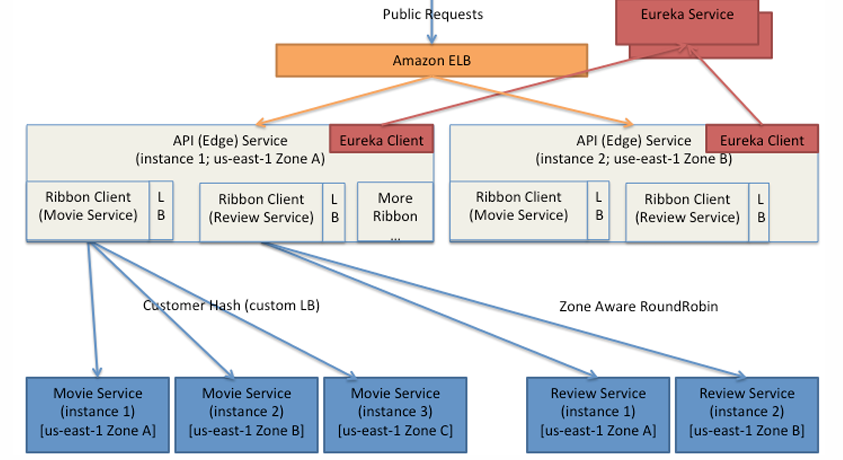
## Neflix 기술 블로그 (2013년)

* 최초의 마이크로서비스 사례
* 수백 개가 넘는 작은 단위의 서비스 운영
* API 기반의 마이크로서비스를 활용한 넷플릭스 아키텍처 의존성 도면  
  (컨테이너가 아닌 VM(AWS EC2)를 이용해 구성)



출처: Neflix 기술 블로그

* Service Oriented Architecture (Netflix의 대표적인 배포 아키텍처)



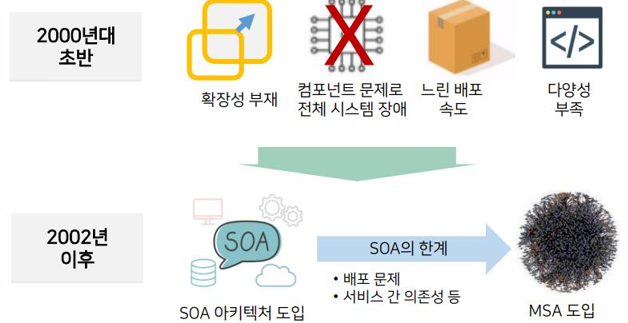
* Netflix 도입 배경과 내용



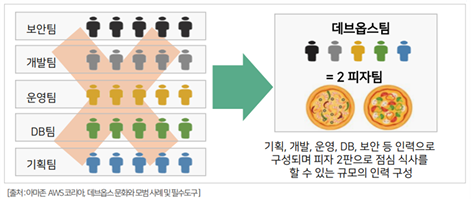
출처: 클라우드 네이티브 정보시스템을 위한 발주자안내서-NIA

## I Love APIs 2015: Microservices at Amazon

* 모놀리식 서비스에서 마이크로 서비스로 변경
* 빌드 타임 감소
* 복잡도 감소
* 수 천개의 DevOps 팀이 자체적으로 배포
* 연간 5천만회(2014년 기준)

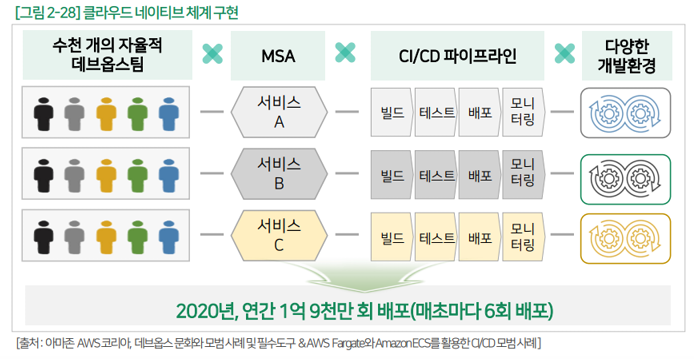


출처: 클라우드 네이티브 정보시스템을 위한 발주자안내서-NIA



### Amazon

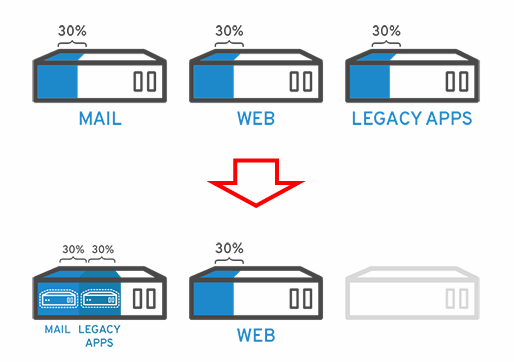
* 수천 개의 자율적 데브옵스팀 구성, MSA 적용, 팀별 CI/CD 파이프라인 자동화를 통한 지속적인 통합/배포
* 다양한 개발환경 구성 등을 통해 신속한 서비스 배포 체계를 구축
* 2020년 기준 연간 1억 9천만 회(매초 6회) 배포



# 하이퍼바이저와 가상 머신의 이해

## 가상화란?

* 가상화는 전통적으로 하드웨어에 종속된 리소스를 사용해 유용한 IT서비스를 만드는 기술
* 가상화를 사용하면 물리적 머신의 기능을 여러 사용자 또는 환경에 배포해 물리적 머신을 최대한 활용



출처: 강의 내 이미지

## 하이퍼바이저

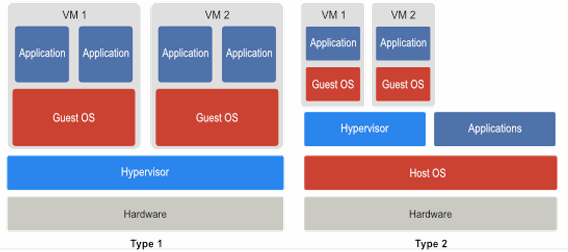
* 소프트웨어가 물기 리소스를 필요로 하는 가상 환경으로부터 물리 리소스를 분리
* 호스트 컴퓨터 대에서 운영체제 다수를 동시에 실행하는 논리적 플랫폼
* 노트북 등의 운영 체제에 배포하거나 서버 등의 하드웨어에 직접 설치
* 하이퍼바이저가 물리 리소스를 분할하여 가상 환경에서 사용



출처: <https://www.redhat.com/ko/topics/virtualization/what-is-virtualization>

### 분류

* Native – type1
  + 해당 하드웨어/베어메탈에 직접 설치
  + 게스트 운영체제는 두 번째 수준으로 실행
  + 제품: Xen, KVM, Xen Server 등
* Hosted – type2
  + 일반 프로그램처럼 호스트 운영체제에서 실행
  + 게스트 운영체제는 세번째 수준으로 실행
  + 제품: Virtualbox, Vmwaare, Parrallesls 등



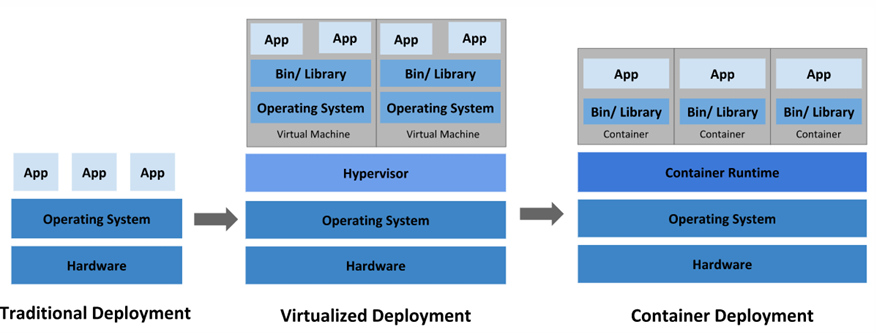
그림출처: [https://www.elktech.org/DevNet/Prese ntations/DEVASC\_Module\_1.pdf](https://www.elktech.org/DevNet/Prese%20ntations/DEVASC_Module_1.pdf)

## 전가상화 방식과 반가상화 방식

* 전가상화(Full Virtualization)
  + 하드웨어를 모두 가상화
  + 게스트 운영체제를 변경하지 않음
  + 물리적인 가상화를 지원하는 CPU 가상화 기술(VT 필요)
  + 네이티브 방식은 이 가상화를 사용
* 반가상화 방식(Para Virtualization)
  + 하드웨어를 완전히 가상화하지 않음
  + 게스트 운영체제 커널 일부 수정이 필요(오픈소스가 아니면 사용 불가)
  + 하이퍼바이저가 모든 제어를 담당하므로 높은 성능을 유지
  + Qemu가 대표적인 도구

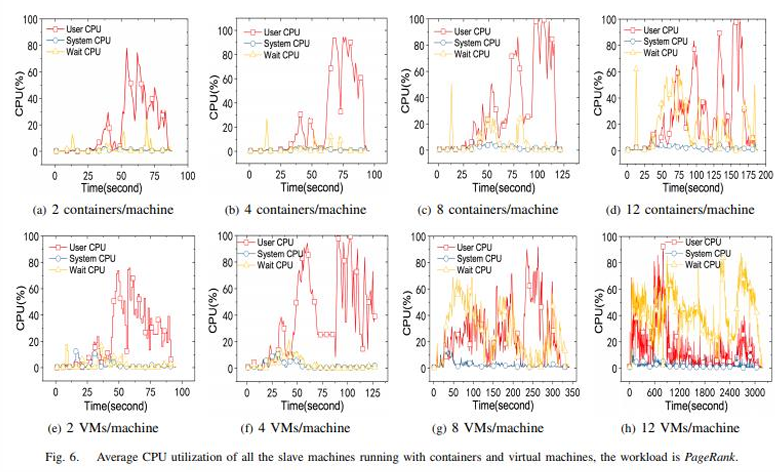
# 컨테이너의 개념과 기술 이해

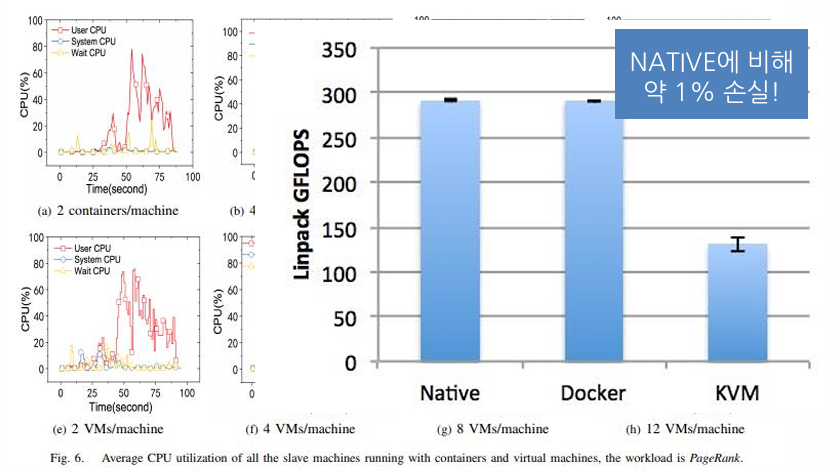
* 컨테이너는 가상환경을 사용해 각 마이크로 서비스를 격리(isolate)하는 기술
* 컨테이너는 가상머신처럼 하드웨어를 전부 구현하지 않기 때문에 매우 빠른 실행 가능



출처: https://Kubernetes.io/docs/concepts/overview/what-is-Kubernetes/

VM과 컨테이너의 실행 시 CPU 사용량 비교

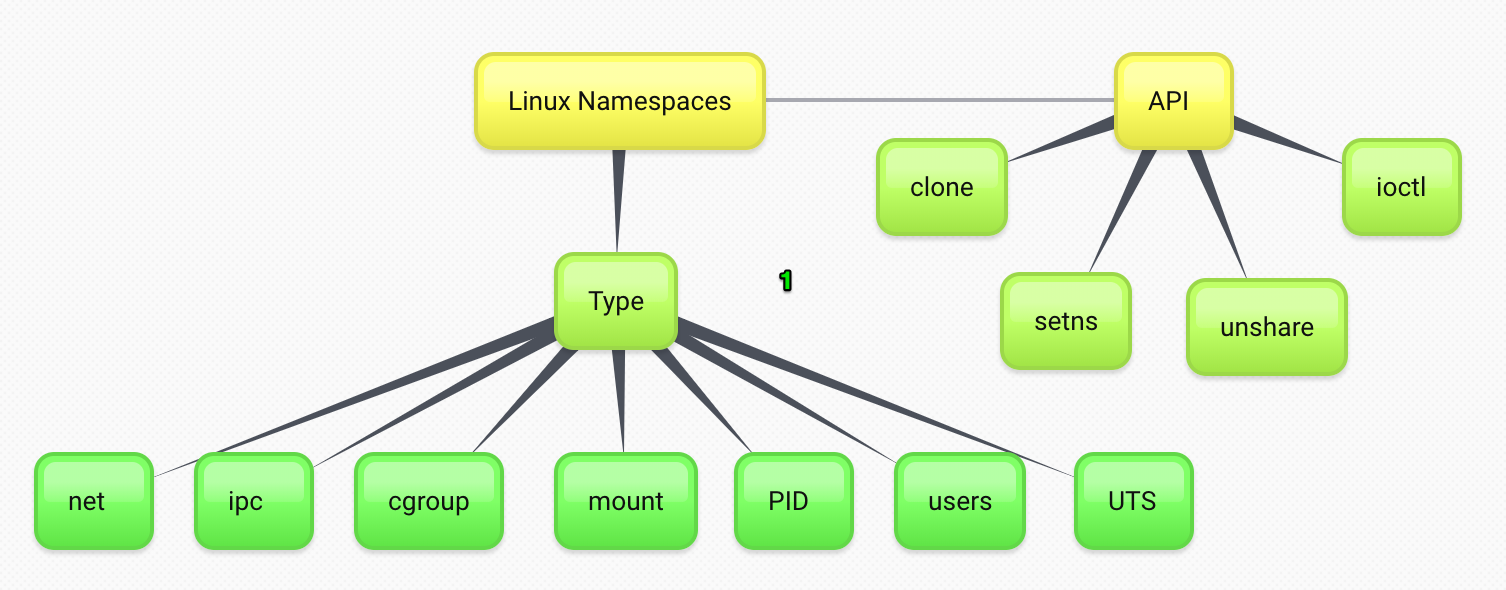




출처: https://arxiv.org/pdf/1807.01842.pdf

## 리눅스 네임 스페이스 – 컨테이너를 격리하는 기술

* 각 프로세스가 파일 시스템 마운트, 네트워크, 유저(uid), 호스트 네임(uts) 등에 대해 시스템에 독립뷰를 제공



출처: [Linux namespaces pid,network,mount,ipc,uts,user,cgroup](https://8gwifi.org/docs/linux-namespace.jsp)

## 리눅스 컨트롤 그룹 – 컨테이너를 격리하는 기술

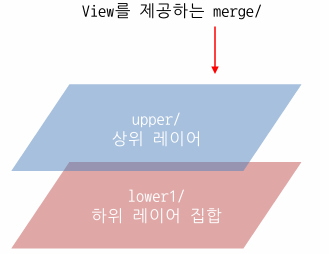
* 프로세스로 소비할 수 있는 리소스 양(CPU, 메모리, I/O, 네트워크 대역대, device 노드 등)을 제한

## 컨테이너를 격리하는 기술 – Union Mount File System

* 동일한 디렉토리에 여러 파일시스템을 마운트하는 기술·기능
* 먼저 마운트된 것을 살려둔 상태로 추가적으로 마운트 하는 것
* 단, 겹치는 것(폴더, 파일)이 있으면 나중 것이 우선
* 리눅스 mount 명령어에 –union 옵션을 붙여 마운트
* 활용사례: CD를 마운트한 후, 같은 곳에 HDD를 유니온 마운트하여, 변경분(증분)이 HDD에 저장되도록 함. 즉, 원본에는 영향을 주지 않고 커스터마이징 부분만 따로 보관할 수 있음



출처: [Storage drivers | Docker Docs](https://docs.docker.com/engine/storage/drivers/)

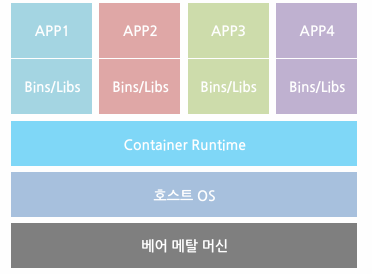


출처: 강의

# 도커와 쿠버네티스 이해

## 도커

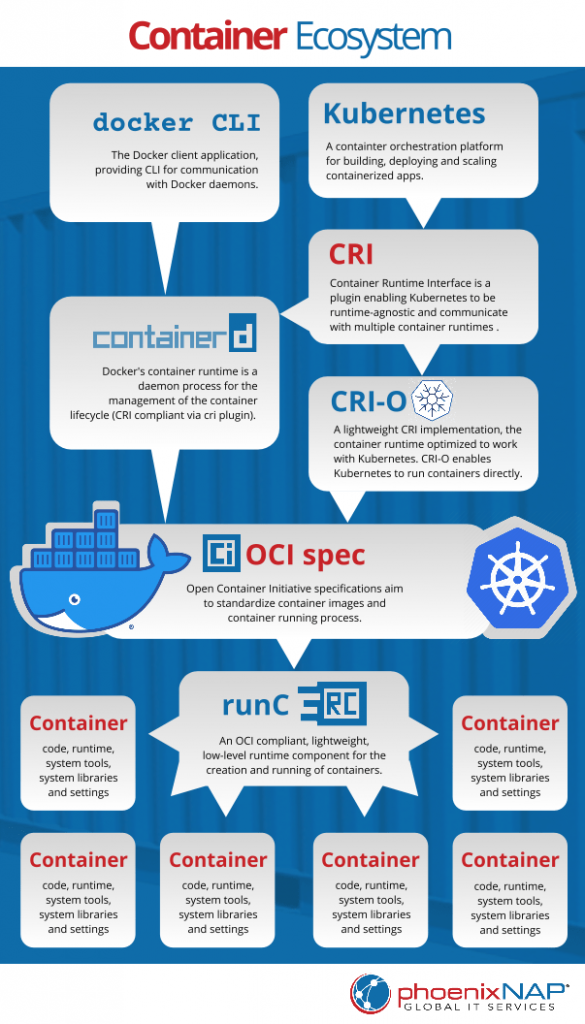
* 컨테이너 기술을 지원하는 다양한 프로젝트 중에 하나
* 컨테이너 기술을 이전에도 있었으나 도커로 인해 알려짐
* 컨테이너 기술의 사실상 표준
* 다양한 운영체제에서 사용 가능(리눅스, 윈도우, MacOS)
* 애플리케이션에 국한되지 않고 의존성 및 파일 시스템까지 패키징하여 빌드, 배포, 실행을 단순화
* 리눅스의 네임 스페이스와 cgroups와 같은 커널 기능을 사용하여 가상화



출처: 강의

도커 아키텍처

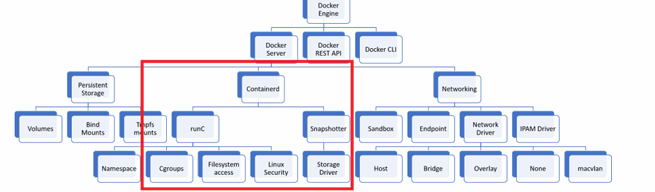
* docker CLI: 컨테이너를 컨트롤하는 프로그램
* CRI: OCI에서 제시하는 컨테이너 런타임 표준
* Containerd, CRI-O: 컨테이너 표준을 만족하는 오픈소스 프로젝트
* runC: 컨테이너 생성 및 실행을 위한 OCI 호환 경량 저수준 런타임



출처: <https://phoenixnap.com/kb/docker-vs-containerd-vs-cri-o->

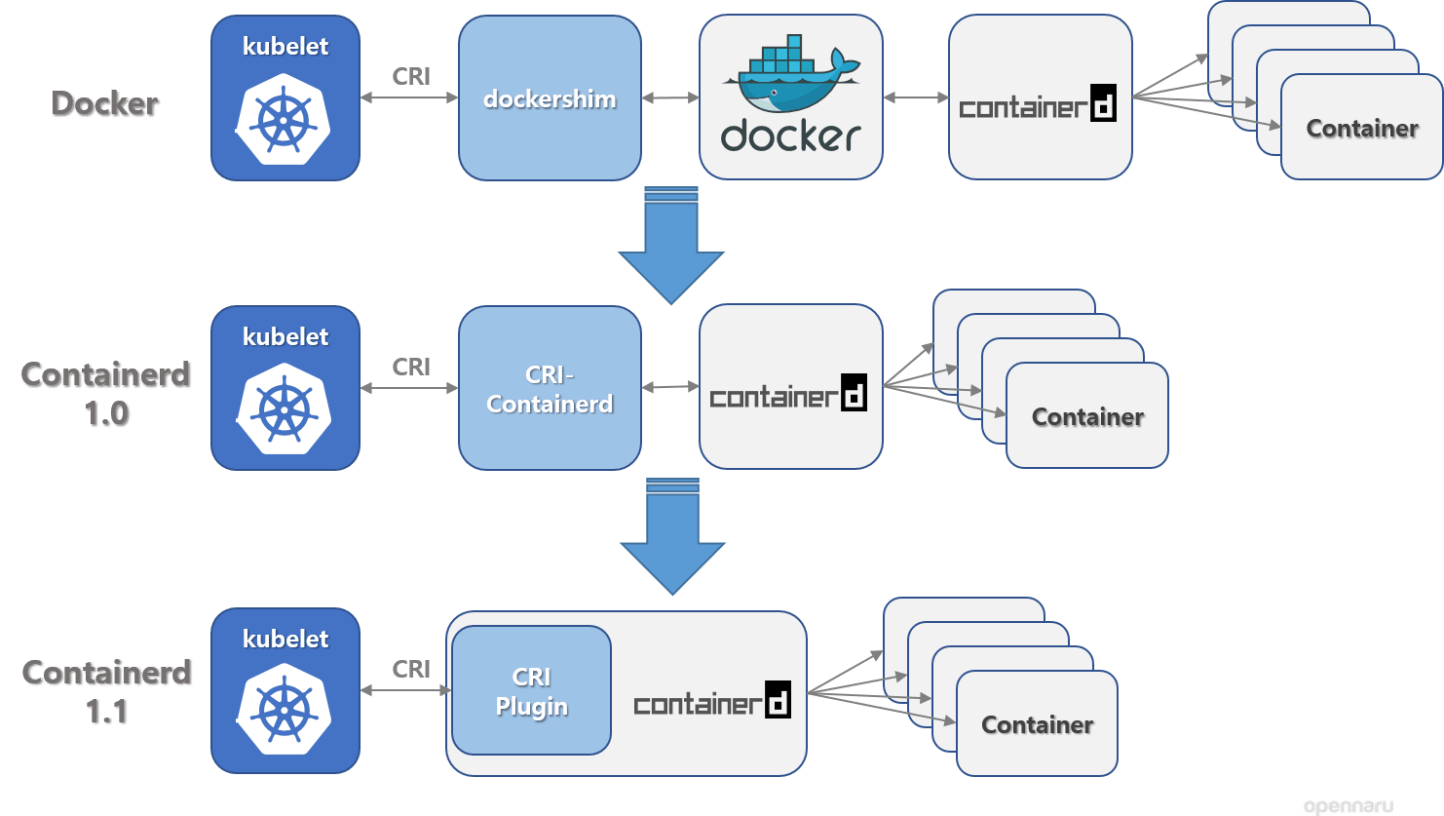
## containerd의 역사

* Containerd는 원래 유명한 컨테이너 플랫폼인 Docker의 구성 요소로 개발
* 레지스트리에서 이미지 가져오기, 컨테이너 시작 및 중지와 같은 컨테이너 관리와 관련된 하위 수준 작업을 처리하기 위해 Docke의 일부로 개발
* 2017년 Docker에서 독립형 프로젝트로 containerd가 추출되어 CNCF(Cloud Native Computing Foundation) 프로젝트가 됨
* 그 이후로 성능, 안정성 및 보안에 중점을 두고 크게 발전하고 개선
* Containerd에는 Docker, Google 및 Red Har과 같은 회사의 개발자를 포함하여 대규모의 활동적인 기여자 커뮤니티가 있음



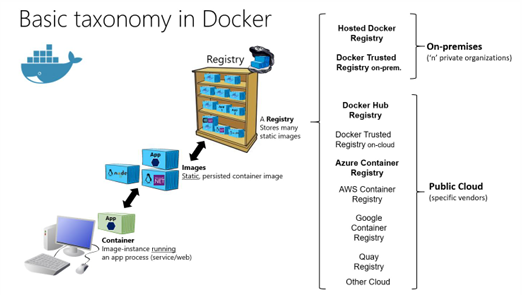
출처: <https://kr.linkedin.com/pulse/topics/home>

## 컨테이너의 진화



출처: <https://yooloo.tistory.com/202>

* 컨테이너: 이미지를 격리하여 독립된 공간에서 실행한 가상 환경
* 이미지: 필요한 프로그램과 라이브러리, 소스를 설치한 뒤 만든 하나의 파일
* 레지스트리: 사용자가 사용할 수 있도록 데이터베이스를 통해 Image를 제공해주고 있음
* 누구나 이미지를 만들어 푸시할 수 있으며 푸시된 이미지는 다른 사람들에게 공유 가능



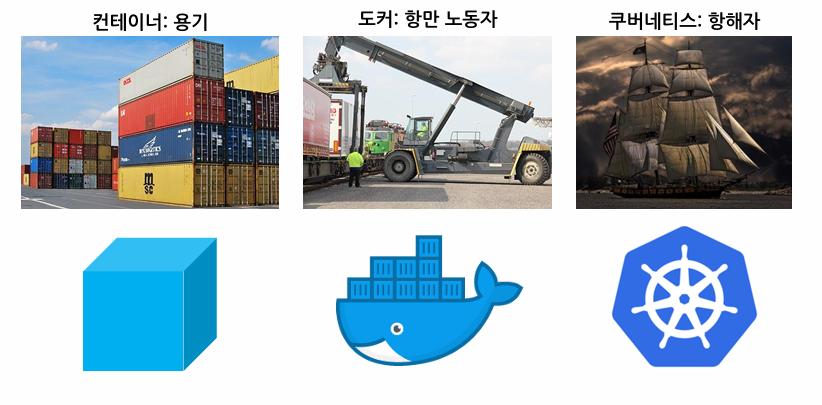
출처: <https://learn.microsoft.com/ko-kr/dotnet/architecture/microservices/container-dockerintroduction/docker-containers-images-registries>

## 도커의 한계

* 서비스가 커지면 커질수록 관리해야 하는 컨테이너의 양이 급격히 증가
* 도커를 사용하여 관리를 한다 하더라도 쉽지 않은 형태

## 쿠버네티스

* 2014년 구글이 오픈 소스를 공개하며 구글의 컨테이너 운영 노하우가 담긴 오픈소스
* 다수의 컨테이너를 자동으로 운영하기 위한 오케스트레이션 도구
* 많은 시스템을 통합, 컨테이너를 다루기 위한 API 제공



출처: 강의