# SDDC 기술

데이터센터의 과거와 현재

# 진행 전…

교육 목표

# 교육 목표

## 부트캠프 개요

이 교육은 기존 레거시 IDC 구조에서 **VIDC/VDC**로 전환 및 AI 기반 SDDC를 다루어 봅니다.

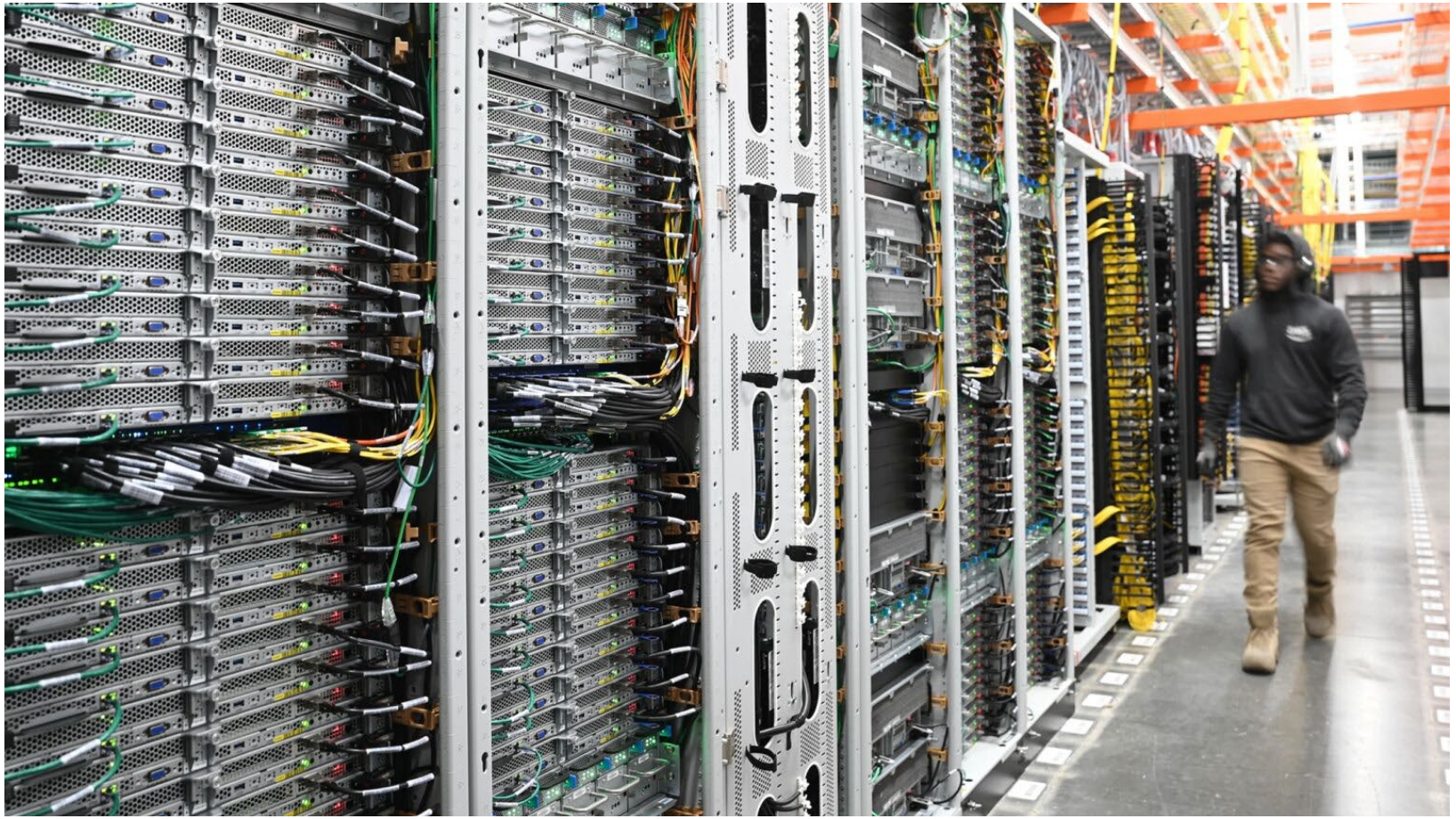
많은 엔지니어는 이미 **VIDC/VDC 개념을 퍼블릭 클라우드**를 통해서 사용자들이 접하고 있습니다.

기존 **온-프레미스 시스템 구조를** 그대로 유지하면서 현대적으로 **VIDC/VDC**로 전환하고, AI를 위한 SDDC를 어떻게 운영 및 관리하는지에 대해서 학습 합니다.

DUSTBOX

# 기존 데이터 센터 정의

## 일반적인 데이터 센터



# 기존 데이터 센터 정의

## 일반적인 데이터 센터

DUSTBOX

# 기존 데이터 센터 정의

## DEFINITION OF DATA CENTER

데이터 센터의 정의는 사전적으로 다음과 같다.

- 애플리케이션 및 서비스를 구축 및 운영하기 위한 IT 인프라가 들어있는 물리적 방/건물/시설
- 서버 운영에 필요한 네트워크 및 서버 리소스의 집합

대다수 데이터 센터는 물리적 장비 및 환경에 매우 민감하며, 대다수 운영 및 설정은 여전히 수동으로 구성 및 운영이 되고 있다.

모든 장비 기능들은 독립적인 하드웨어로 구성이 된 경우가 있으며, 장애를 대비하여 H/A, D/R과 같은 많은 시스템들이 설치 및 구성이 된다.

# 기존 데이터 센터 특징

## DEFINITION OF DATA CENTER

데이터 센터의 특징은 다음과 같다.

1. 물리적 환경에 **스위치/서버/스토리지/라우터 및 냉각설비**가 많이 구성이 된다.
2. 중앙 집중형으로 하나의 건물 혹은 여러 건물에 분산 구성 및 처리가 된다.
3. 여전히 대다수 자원은 수동으로 설치 구성 및 운영이 된다.

# 기존 데이터 센터 장단점

## DEFINITION OF DATA CENTER

기존 데이터 센터의 장점과 단점은 다음과 같다.

1. 계획된 규모 및 크기로 빠르게 데이터센터 구축 및 운영이 가능하다.
2. 통제 및 운영에 유리하고, 낮은 레이턴시 시스템 및 서비스 구성이 가능하다.
3. 유연성 및 확장성이 많이 떨어지며, 신뢰성 및 무결성 부분도 많이 낮다.

# 현재 데이터센터(클라우드)

## NOWADAYS DATA CENTER

현재 새로 짓고 있는 데이터센터 혹은 데이터센터급 규모 시스템들은 **컴퓨트/스토리지/네트워크는** 자원 추상화를 하고 있다.

목적은, 이를 통해서 **빠른 자원 배포 및 구성**, **장애 발생 시 민첩대응**이 가능하다.

이전 데이터센터 클라우드는 특정 회사 제품 및 기술에 종속되어 있었지만, 현재는 대다수 데이터센터는 오픈소스 기반으로 구성된 솔루션으로 구성이 되어 있다.

# VIDC/SDDC

## NOWADAYS DATA CENTER

최근 데이터센터는 이전 데이터센터와 다르게 **단일 유닛 시스템**으로 구성하지 않는다.

다수의 데이터센터 시스템은 **랙 단위(Rack Unit)**로 구성한다. 이러한 구성을 하기 위해서 확장이를 VIDC(VDC) 혹은 SDDC 개념 기반으로 인프라 설계 및 구성한다.

- SDDC: https://www.redhat.com/en/topics/automation/what-is-a-sddc
- VIDC: https://www.hpe.com/emea_africa/en/what-is/data-center-virtualization.html

# VIDC

## VIDC(VDC)는 무엇인가?

VIDC는 Virtual Data Center(**혹은 VDC**)의 약자이다. 말 그대로 이전에 사용하던 하드웨어 구성들을 가상화 기반으로 구현한다.

상대적으로 기존 **Legacy D/C**보다는 비용이 저렴하다. 다만, 가상화 기반으로 네트워크 및 서버 구성이 되다 보니, 데이터센터에서 사용하는 하드웨어는 도입 시, 비용이 높아지는 단점이 있다.

가상화 기반으로, 모든 자원을 논리적 형태로 관리 및 배포하니 **위치/장소/지역 그리고 확장성에 대한 제한이 기존 레거시**보다 매우적다.

# VIDC

## VIDC 구성 시 사용하는 소프트웨어 요소

VIDC 구성 시 다음과 같은 소프트웨어 기반으로 구성한다.

-SDN(Switch, Router)

-NFV(Load Balancer, Firewall)

-Virtualization(Hypervisor)

-Container(Runtime)

# VIDC

## VIDC 구성 시 사용하는 소프트웨어 요소

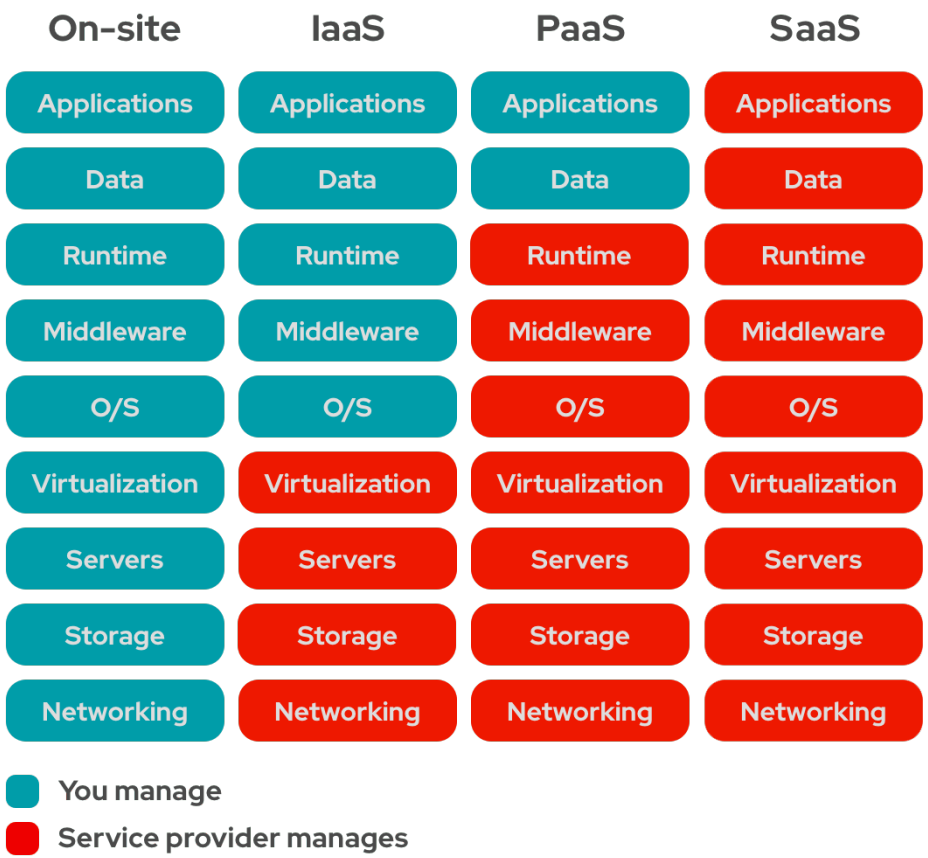
이를 구성 시, 아래와 같은 플랫폼 기반으로 구성한다.

1. PaaS(Platform as a Service)
2. IaaS(Infrastructure as a Service)

대다수 데이터 센터에 구성된 VIDC는 PaaS 혹은 IaaS 기반으로 인프라 시스템을 구축한다. PaaS/IaaS의 차이점은 다음과 같다.

# AS A SERVICE

## 플랫폼 서비스 차이점



# VIDC

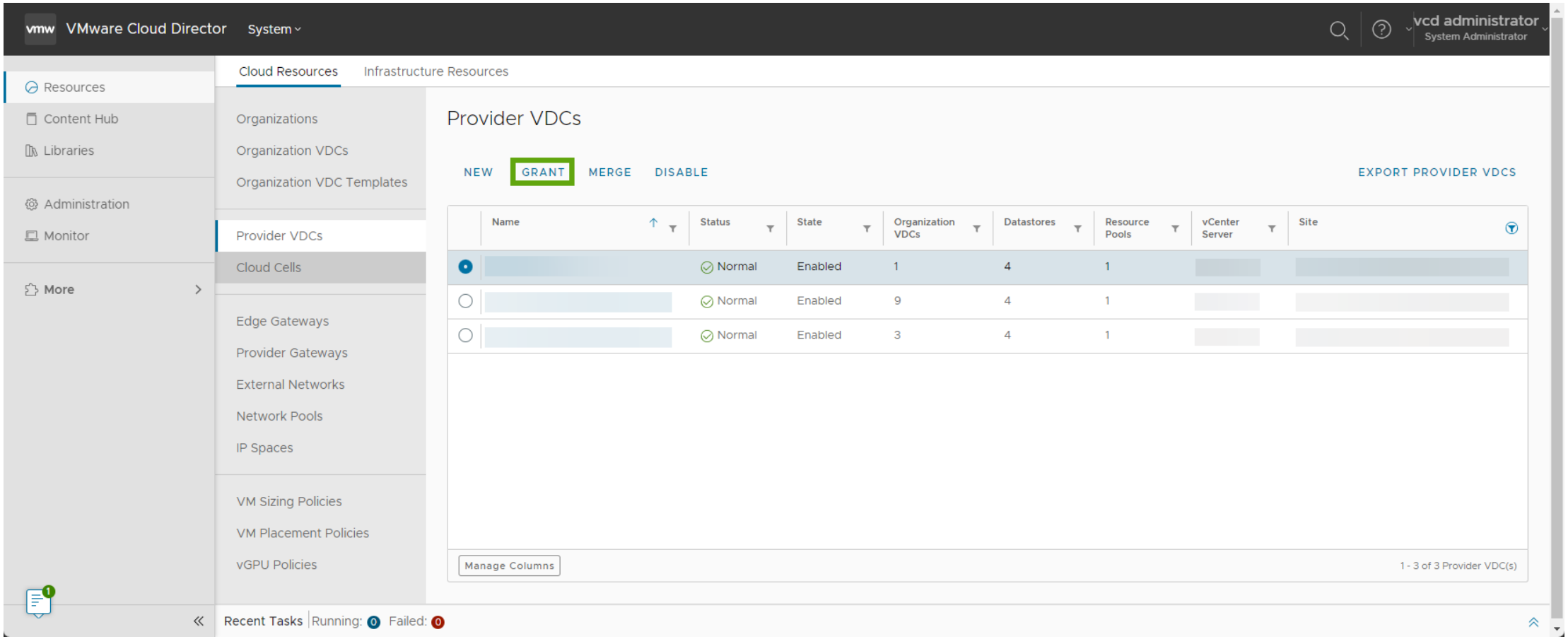
## VIDC(VDC)는 무엇인가?

VIDC의 대표적인 솔루션은 보통 다음과 같다.

1. VMware vShpere(Cloud Director)
2. OpenStack
3. ONOS
4. OpenDayLight
5. Ceph Storage

# VMWARE DEPLOY VDC

## VMWARE VDIC

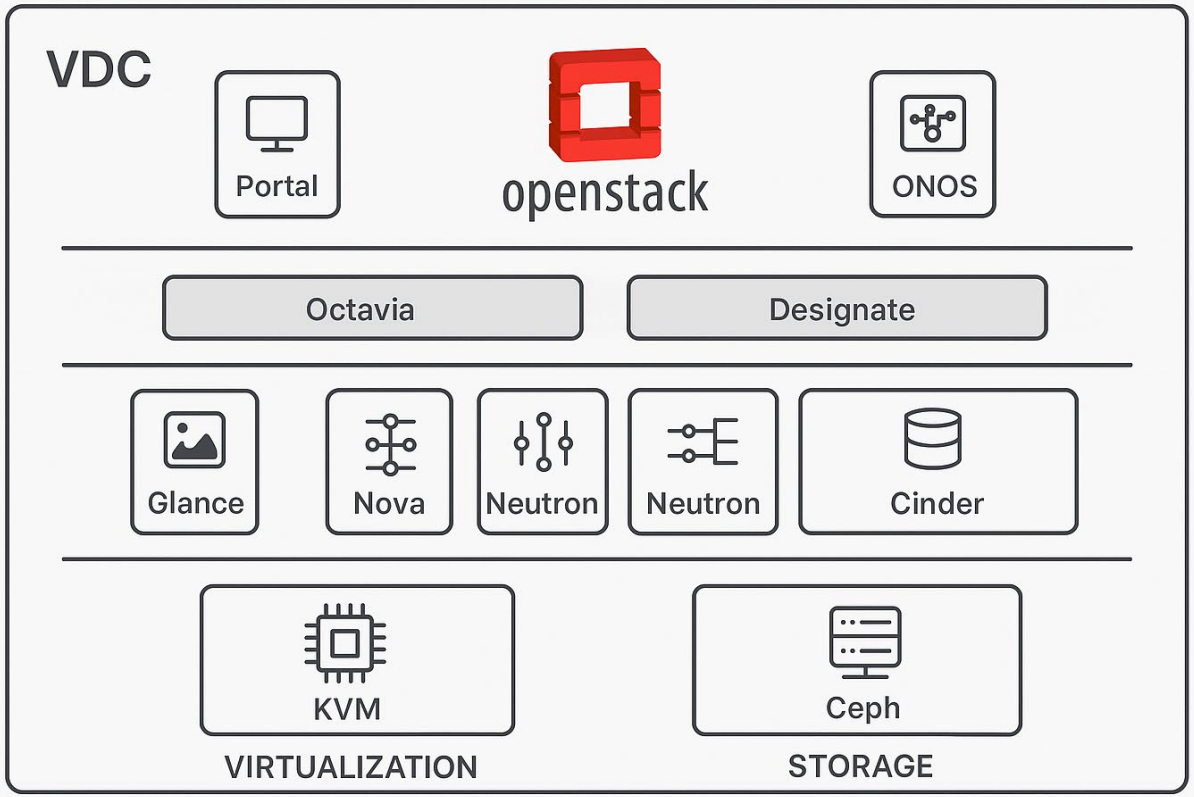





2025-12-26

# OPENSTACK VIDC

## OPENSTACK VIDC



# VIDC에서 사용하는 추상화 계층

## VIDC 계층 분리

| 계층 | 주요 역할 | 대표 기술 / 미들웨어 | 설명 |
|---|---|---|---|
| 가상화 계층 | 물리 자원을 가상화하여 논리적 자원으로 분리 | VMware vSphere / KVM / Hyper-V | VIDC의 기반 가상화 엔진. OpenStack이나 vCloud 등 상위 관리 플랫폼이 이 위에 올라감. |
| 클라우드 관리 계층 | VM, 네트워크, 스토리지, 사용자 정책 등을 통합 관리 | OpenStack / VMware vCloud Director / CloudStack | VIDC의 **"운영 핵심"** 계층으로, 사용자 포털 및 API를 통해 가상 자원 제공. |
| 네트워크 가상화 계층 | 네트워크를 소프트웨어로 제어, 분리 및 자동화 | OpenDaylight / ONOS / VMware NSX / Tungsten Fabric | VIDC 내부 가상 네트워크 구성과 연결 제어를 담당. OpenStack Neutron과 연동됨. |
| 스토리지 가상화 계층 (SDS Layer) | 물리 스토리지를 논리화하여 유연하게 할당 | Ceph / VMware vSAN / OpenEBS / LVM 기반 SDS | VM 또는 서비스 단위로 유연한 스토리지 자원 제공. |
| 오케스트레이션/서비스 관리 계층 | 전체 리소스 배포·자동화 및 정책 기반 관리 | Heat (OpenStack) / Terraform / Ansible / Cloudify | 사용자 요청에 따라 VMs, 네트워크, 보안그룹 등을 자동으로 생성 및 배치. |
| 포털/API/사용자 인터페이스 계층 | 사용자에게 서비스 제공 UI/UX 제공 | Horizon (OpenStack), vCloud Portal, Custom Portal | 사용자가 자원 생성/삭제를 직접 수행하는 인터페이스. |
| 보안 및 인증 계층 | 접근제어, 인증/인가, 정책관리 | Keystone (OpenStack), FreeIPA, LDAP, IAM | 사용자별 권한 및 인증 관리 수행. |

# SDDC는 무엇인가?

## SDDC의 정의

SDDC는 Software Defined Data Center의 약자. SDDC는 기본적으로 VIDC를 기반 기술을 가져가며, 이를 **코드(code)** 기반으로 운영/관리가 가능하도록 한다.

현재 대다수 데이터 센터 및 대규모 서비스는 대다수가 SDDC 기술 및 개념 기반으로 구성 및 운영이 되고 있다. 대표적인 예제는 Netflix, Google, Meta와 같은 업체가 있다.

기존 플랫폼, VMWARE, OPENSTACK, KUBERNETES도 SDDC 개념 구현이 가능하다. 대다수 오픈소스는 YAML, JSON기반으로 SDDC를 반영한다.

# SDDC는 무엇인가?

## SDDC의 정의

SDDC를 구현하기 위해서는 자원을 코드 기반으로 운영이 가능한 미들웨어 혹은 솔루션이 필요하다.

현재 다음과 같은 프로젝트가 오픈소스에서 많이 사용하고 있다.

1. Kubernetes(Container, Virtual Machine)
2. OpenStack(Container, Virtualization)
3. SUSE REK2/Harvester(Container, Virtualization)
4. Redhat OpenShift(Container, Virtualization)

# VIDC/SDDC 기술 비교

## 개념 비교

| 구분 | VIDC (Virtual Internet Data Center) | SDDC (Software Defined Data Center) |
|---|---|---|
| 정의 | 인터넷 기반으로 가상화된 데이터센터 자원을 사용자에게 제공하는 클라우드형 데이터센터 | 데이터센터의 모든 인프라(Compute, Storage, Network, Security)를 소프트웨어로 정의·관리하는 구조 |
| 핵심 개념 | “가상화된 인터넷 자원 제공” (주로 IaaS 형태의 서비스 제공) | “모든 인프라를 코드로 제어” (완전 자동화된 데이터센터 운영) |
| 구성 요소 | 가상 서버, 가상 네트워크, 가상 스토리지 등 인터넷 상에서 제공되는 리소스 | SDN(Software Defined Network), SDS(Software Defined Storage), SDC(Software Defined Compute) |
| 운영 방식 | 주로 서비스 제공자의 인프라 위에서 고객이 가상 자원을 임대·활용 | 기업 내부 또는 클라우드 인프라 전반을 자동화·오케스트레이션하여 자체 운영 |
| 관리 주체 | 클라우드 서비스 제공자 (예: KT, LG U+, Naver Cloud 등) | 기업 내부 IT 조직 혹은 하이퍼컨버지드 인프라 관리자 |

# VIDC/SDDC 기술 비교

## 개념 비교

| 구분 | VIDC (Virtual Internet Data Center) | SDDC (Software Defined Data Center) |
|---|---|---|
| 자동화 수준 | 중간 수준 — 사용자 포털을 통한 리소스 생성·삭제 중심 | 매우 높음 — 인프라 전체를 API/코드 기반으로 제어 (IaC, Orchestration 포함) |
| 목표 | 손쉬운 가상 인프라 제공 및 신속한 서비스 구축 | 인프라 효율 극대화, 정책 기반 자동화, 완전한 데이터센터 추상화 |
| 대표 기술/제품 | OpenStack, VMware vCloud, AWS EC2, Ncloud VIDC | VMware vSphere + NSX + vSAN, OpenStack, Red Hat OSP, Nutanix, Harvester 등 |
| 적용 사례 | 클라우드 호스팅, 공공기관 클라우드 존, 중소기업용 IaaS 서비스 | 대기업 데이터센터, 프라이빗 클라우드, 하이브리드 클라우드, 엣지 DC |
| 장점 | 빠른 구축, 낮은 초기비용, 서비스 확장 용이 | 자동화·정책 기반 운영, 유연한 확장성, 멀티클라우드 통합 가능 |
| 단점 | 커스터마이징 제약, 완전한 제어권 부재 | 초기 구축 복잡, 고비용(기술 인력 필요) |

# 오픈스택 랩

## 오픈스택 학습 내용

오픈스택 대시보드에 접근 후, 간단하게 자원을 확인한다.

1. 프로젝트
2. 스토리지/네트워크/볼륨
3. 네트워크 보안
4. 서비스 설명
5. 자동화

# 자동화

앤서블

# 자동화(IaaC) 개념

## INFRA STRUCTURE AS CODE

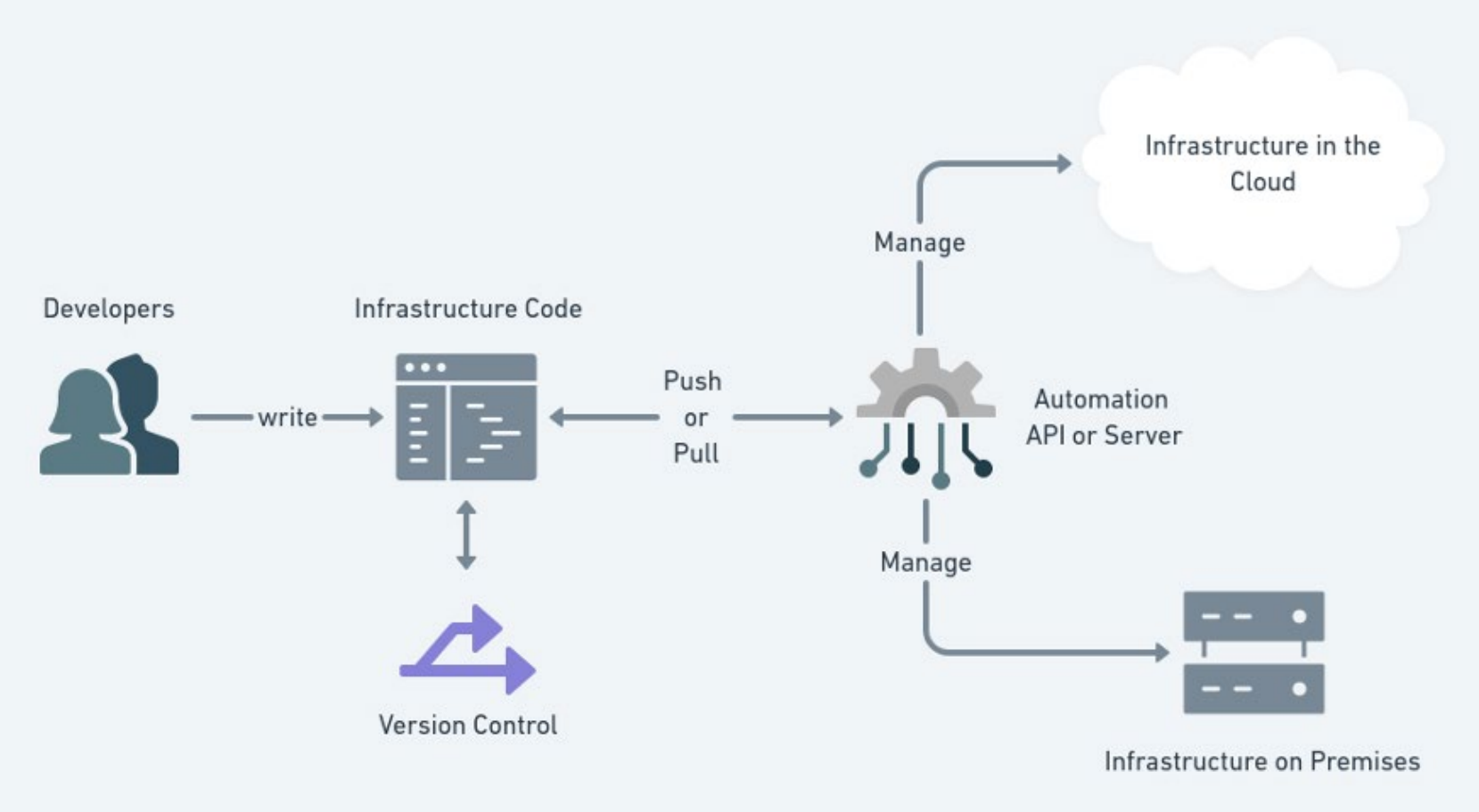
IDC에서 VIDC/VDC로 성공적으로 전환이 되었으면, SDDC구현을 위해서 코드기반 자동화가 필요하다.
이를 위해서 몇 가지 도구가 필요하다.

1. 웹 기반 대시보드
2. API 및 명령어(CLI)기반 접근
3. **코드기반 컴퓨팅 리소스 접근**

3번 조건 구성하기 위해서 어떠한 도구들이 있는지, 어떠한 방법이 있는지 확인한다.

# 자동화(IaaC) 개념

## INFRA STRUCTURE AS CODE



# 자동화(IaaC) 개념

## INFRA STRUCTURE AS CODE


SDDC에서는 작은 작업 수준에서는 **대시보드 및 CLI**로 작업하지만, 생성 및 관리하는 자원 규모가 커지는 경우에는 더 이상 대시보드로 관리가 어려운 부분이 있다.

또한, 대다수 자원에 대한 정보를 **스프레드시트 기반**으로 관리하고 있기 때문에 보안적으로 문제가 많이 발생한다.

# 자동화(IaaC)

## INFRA STRUCTURE AS CODE

이러한 문제를 IaaC(Infrastructure as a Code)로 구성 시, 이러한 문제를 좀 더 효율적으로 접근 및 해결이 가능하다.

IaaC를 지원하는 도구는 대표적으로 다음과 같다.

1. Ansible
2. Terraform(OpenTofu)
3. Pulumi
4. Salt

# 자동화(IaaC)

## INFRA STRUCTURE AS CODE

| 커뮤니티 버전 | 라이선스 | 상용 버전 | 오픈소스 여부 |
|---|---|---|---|
| Ansible Core | GPLv3 | Ansible Automation Platform | 완전 오픈소스 |
| SaltStack | Apache 2.0 | Salt Enterprise (VMware) | 커뮤니티판 오픈소스 |
| | BUSL | HashiCorp Terraform Cloud | 부분 폐쇄 |
| OpenTofu | MPL 2.0 | 없음 | 완전 오픈소스 |
| Pulumi | Apache 2.0 | Pulumi Cloud | 코어는 오픈소스 |

# 자동화 도입 이유

## IAC BETTER THAN SCRIPT

자동화 도구는 한국 기준, 2022년도부터 기업들이 점차적으로 도입을 시작하였다. 도입 이유는 보통 다음과 같다.

1. 잦은 반복 작업 수행
2. 잦은 반복적인 작업 실수
3. 수동 작업 수행 시, 품질 유지 문제

IaC 도입 후, 사용하는 경우 보통 이러한 문제가 현장에서 사라지거나 혹은 높은 확률로 장애 및 작업 실수가 줄어든다.

https://www.forbes.com/councils/forbestechcouncil/2024/03/14/using-infrastructure-as-code-to-save-time-and-money/

# 자동화

## ABOUT ANSIBLE

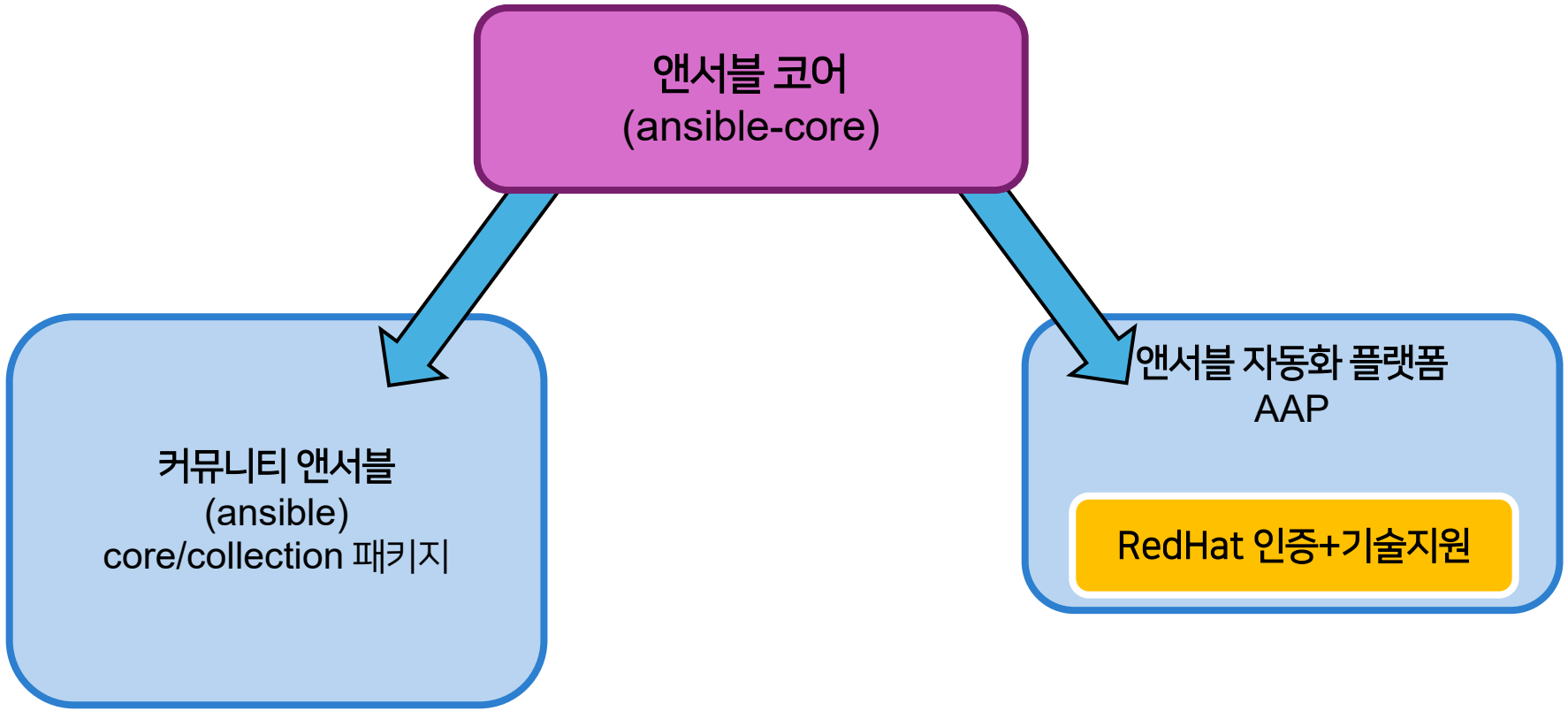
Ansible은 원래 Ansible, Inc.에서 개발되었으며, 2015년에 Red Hat에 인수. 현재 Ansible은 두 가지 릴리즈로 나뉘어 유지.

1. **Ansible Core**: CLI 명령어와 기본 모듈 등 핵심 기능만 포함된 **최소 단위 배포판**
2. **Ansible (Community Distribution)**: 다양한 **공식/비공식 컬렉션**을 포함한 **전체 배포판**

참고로, 과거에는 Ansible을 **Ansible Engine**이라고 부르기도 했으나, 현재는 해당 명칭은 더 이상 공식적으로 사용되지 않는다. 기업용으로는 Red Hat의 **Ansible Automation Platform**이 별도 제공.

# 자동화

## 앤서블 소개


앤서블 코어
(ansible-core)
커뮤니티 앤서블
(ansible)
core/collection 패키지
앤서블 자동화 플랫폼
AAP
RedHat 인증+기술지원

# 자동화

## 앤서블 소개

코어는 앤서블 핵심 모듈로 구성이 되어 있으며, 그 이외 확장으로 `collection, roles`로 통해서 확장이 가능하다. 현재 앤서블은 **앤서블 코어**로 통합 및 배포가 되고 있다.

각 확장은 제 3자 제공 혹은 커뮤니티로 나누어져 있고, 주요 코어 모듈 혹은 확장 모듈 경우 Redhat기여 혹은 제공을 하고 있음. 코드상 차이는 크게 없음.

1. 기본적으로 많이 사용하는 모듈(제공) `ANSIBLE CORE`에서는 `POSIX/BUILT-IN`이다.
2. 그 이외 나머지 기능들은 `collection(community, vender)`로 확장.

# 자동화

## 앤서블 프로그램

또한 앤서블 코어는 두 가지 릴리즈 방식이 있다.

**ansible-core**

- 앤서블 코어는 앤서블 인터프리터 + 코어 모듈
- 바이너리 프로그램

**ansible-project**

- 앤서블 코어 + 추가적인 컬렉션 구성

# 자동화

## 앤서블 프로그램

### ansible-navigator

- 앤서블 통합 도구. 해당 명령어로 플레이북 관리 및 실행이 가능하다. 다만, 현재 레드햇 배포판 제외하고 다른 모든 리눅스 배포판에서는 PIP를 통해서 설치해야 한다.
- 네비게이터를 사용하기 위해서는 컨테이너 런타임이 설치가 필요하다.
- 포드만 혹은 도커 기반으로 실행

### ansible-builder

ansible-builder는 **Ansible 실행 환경(Execution Environment, EE)** 을 **자동으로 빌드**해주는 도구.
쉽게 말해, **Ansible이 돌아가는 맞춤형 컨테이너 이미지**를 만드는 도구.

# 자동화

## 앤서블 프로그램

### ansible-navigator

- 앤서블 통합 도구. 해당 명령어로 플레이북 관리 및 실행이 가능하다. 다만, 현재 레드햇 배포판 제외하고 다른 모든 리눅스 배포판에서는 PIP를 통해서 설치해야 한다.
- 네비게이터를 사용하기 위해서는 컨테이너 런타임이 설치가 필요하다.
- 포드만 혹은 도커 기반으로 실행

### ansible-builder

ansible-builder는 **Ansible 실행 환경(Execution Environment, EE)** 을 **자동으로 빌드**해주는 도구.
쉽게 말해, **Ansible이 돌아가는 맞춤형 컨테이너 이미지**를 만드는 도구.

# 자동화

## 앤서블 프로그램

YAML은 파이썬과 비슷하게 아래와 같은 조건으로 작성이 되어 있음.

- 들여쓰기 2칸
- 탭은 빈 공간(space)으로 전환
- 자동 들여쓰기 활성화

# 자동화

## 앤서블 프로그램

vi/vim를 위해서 기능 강화.

```
eploy# touch .vimrc
```

```
deploy# dnf install vim vim-enhanced neovim yamllint
```

레드햇 계열 배포판을 사용하는 경우, 아래와 같이 설치.

```
deploy# dnf install vim-ansible
```

# 자동화

## 앤서블 프로그램

리눅스

```
# dnf install nano wget curl

# curl
https://raw.githubusercontent.com/scopatz/nanorc/master/install.sh |
sh
```

# 자동화

## 앤서블 프로그램

```
# vi ~/.nanorc
color brightwhite "#.*$"
color ,red ":\w.+$"
color ,red ":'.+$"
color ,red ":".+$"
color ,red "\s+$"
color ,red "['\"][^['\"]]*$"
color yellow "['\"].*['\"]"
color brightgreen ":( |$)"
set tabsize 2
set tabstpotospacs
```

# 표준 데이터 파일

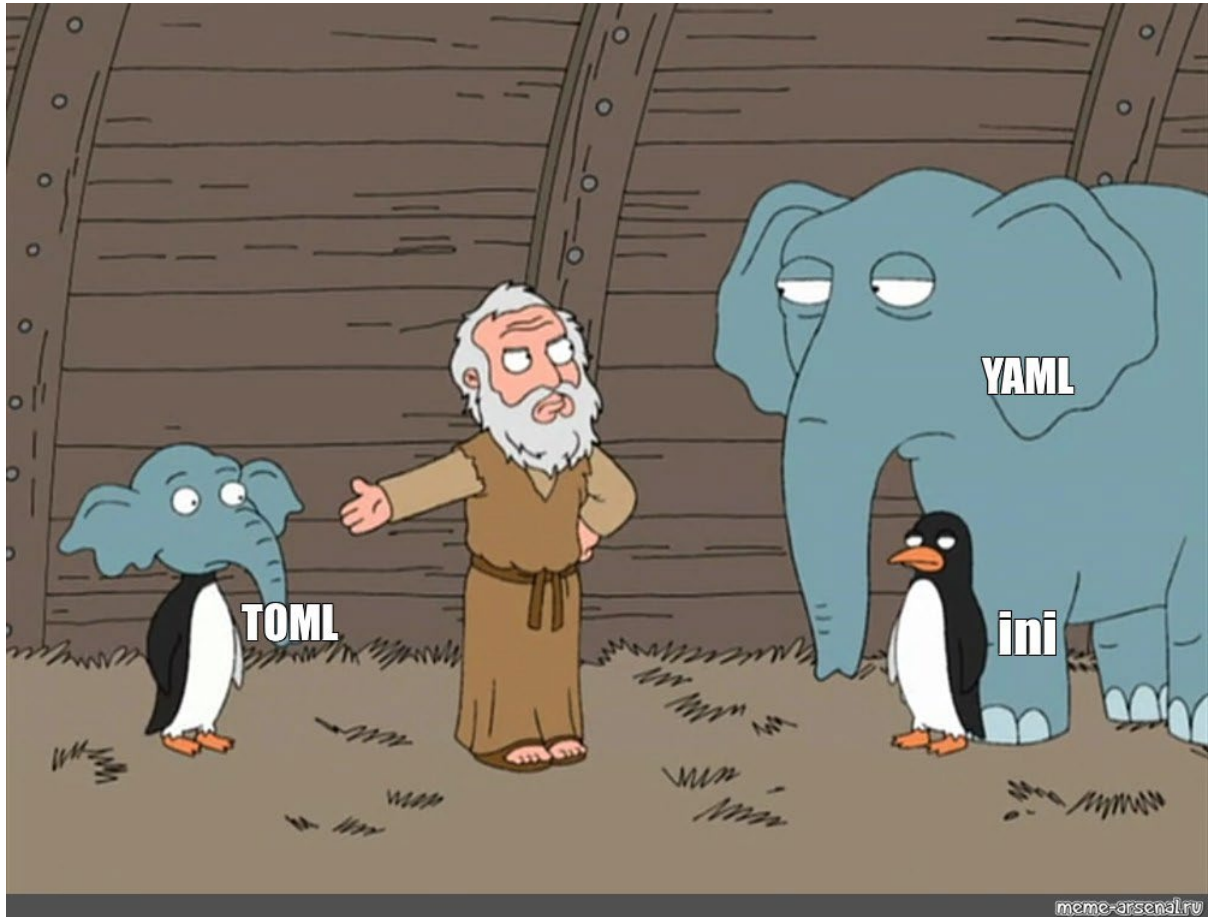
현재 오픈소스는 데이터 형식에 대해서 표준화를 진행하고 있다. 현재, 오픈소스에서 다음과 같이 자원 표준화를 진행하고 있다.

1. TOML
2. YAML
3. JSON

앤서블은 TOML/YAML/JSON를 전부 사용하고 있다. 자동화를 위한 **작업 데이터 셋**은 YAML으로 작성이 되며, 앤서블 내부적으로 데이터 핸들링은 JSON으로 데이터를 다룬다.

인벤토리와 같은 설정형식은 기존 INI에서 TOML으로 변환하고 있다. TOML를 앤서블 코어 2.8 이후부터 지원한다.

# YAML/JSON/TOML



# YAML 조건

## YAML RULE

앤서블에서 사용하는 문법을 작성하기 위해서는 다음과 같은 조건을 만족해야 한다.

1. 최소 한 칸 이상의 띄어쓰기(권장은 2칸)
2. 탭 사용시 반드시 빈 공간으로 전환
3. 블록 구별은 -(대시)로 반드시 처리

# YAML?

## YAML RULE

```
- name: simple playbook
  hosts: all
  become: false
  tasks:
    - name:
      module
        args1:
        args2:
```

# YAML??

## YAML RULE

YAML를 형식(format)이 정해져 있다. 다만, 문제가 들여쓰기(ident)가 조금은 명확하지 않는 부분이 있다. 예를 들어서 리스트 혹은 딕셔너리는 선언 기호인 - 경우에는 다음과 같은 문제가(?) 있다.

```
tasks:
- test
    parameter
```

```
tasks:
  - test
      parameter
```

위의 두 개는 들여쓰기가 다르지만, 결과적으로 같은 파싱(parsing)결과가 나온다. 결국 사람마다 다른 형식으로 작성하기 때문에, 적절하게 작업자끼리 형식에 대한 합의가 필요하다.

# 자동화

## 앤서블 예제

위의 내용을 아래와 같이 작성한다.

```
---
- name: Ansible Basic Practice
  hosts: node1,node2
  become: yes
  tasks:
    # 1. /tmp/test 파일 생성
    - name: Create /tmp/test file
      ansible.builtin.file:
        path: /tmp/test
        state: touch
        owner: ansible
        group: ansible
        mode: '0644'
```

# 자동화

## 앤서블 예제

위의 내용 계속 이어서…

```
    # 2. 사용자 생성
    - name: Create users test1 and test2
      ansible.builtin.user:
        name: "{{ item }}"
        password: "{{ 'test' | password_hash('sha512') }}"
        groups: adm,wheel
        state: present
      loop:
        - test1
        - test2
```

# 자동화

## 앤서블 예제

위의 내용 계속 이어서…

```
    # 3. 패키지 설치
    - name: Install web and ftp packages
      ansible.builtin.dnf:
        name:
          - httpd
          - vsftpd
          - squid
        state: present
      register: package_result
```

# 자동화

## 앤서블 예제

위의 내용 계속 이어서…

```
    # 4. 서비스 시작
    - name: Start httpd/vsftpd/squid services
      ansible.builtin.service:
        name: "{{ item }}"
        state: started
        enabled: yes
      loop:
        - httpd
        - vsftpd
        - squid
```

# 자동화

## 앤서블 예제

위의 내용 계속 이어서…

```
    # 5-1. 설치한 패키지 제거
    - name: Remove installed packages
      ansible.builtin.dnf:
        name:
          - httpd
          - vsftpd
          - squid
        state: absent
```

# 자동화

## 앤서블 예제

위의 내용 계속 이어서…

```
    # 5-2. 생성한 사용자 제거
    - name: Remove created users
      ansible.builtin.user:
        name: "{{ item }}"
        state: absent
        remove: yes
      loop:
        - test1
        - test2
```

# 자동화

## 앤서블 예제

위의 내용 계속 이어서…

```
    # 6. ansible-doc 확인 (도움말 확인은 설명으로 대체)
    - name: Display dnf module info (example)
      ansible.builtin.debug:
        msg:
          - "명령어 예시: ansible-doc dnf"
          - "도움말을 확인한 뒤 아래와 같이 httpd 설치 가능"
          - "ansible node1 -m dnf -a 'name=httpd state=present'"
```

# VDC

데이터센터

# 기술 활용(오픈스택)

## 오픈스택 소개

1. 오픈스택은 본래 미국 NASA에서 프로젝트용 웹 사이트 관리를 효율적으로 하기 위해서 Rackspace에 위탁 및 운영.
2. Rackspace는 여러 프로젝트 및 자원의 회수 및 재활용을 효율적으로 하기 위해서 리눅스 기반으로 관리도구를 만들기 시작.
3. 도구가 점점 발전이 되면서 최종적으로 **"오픈스택"**이라는 이름으로 커뮤니티에 기여 및 공개가 되었다.

자세한 내용은 아래 사이트에서 확인이 가능하다.

`https://docs.openstack.org/project-team-guide/introduction.html`

# 기술 활용(오픈스택)

## 오픈스택 소개

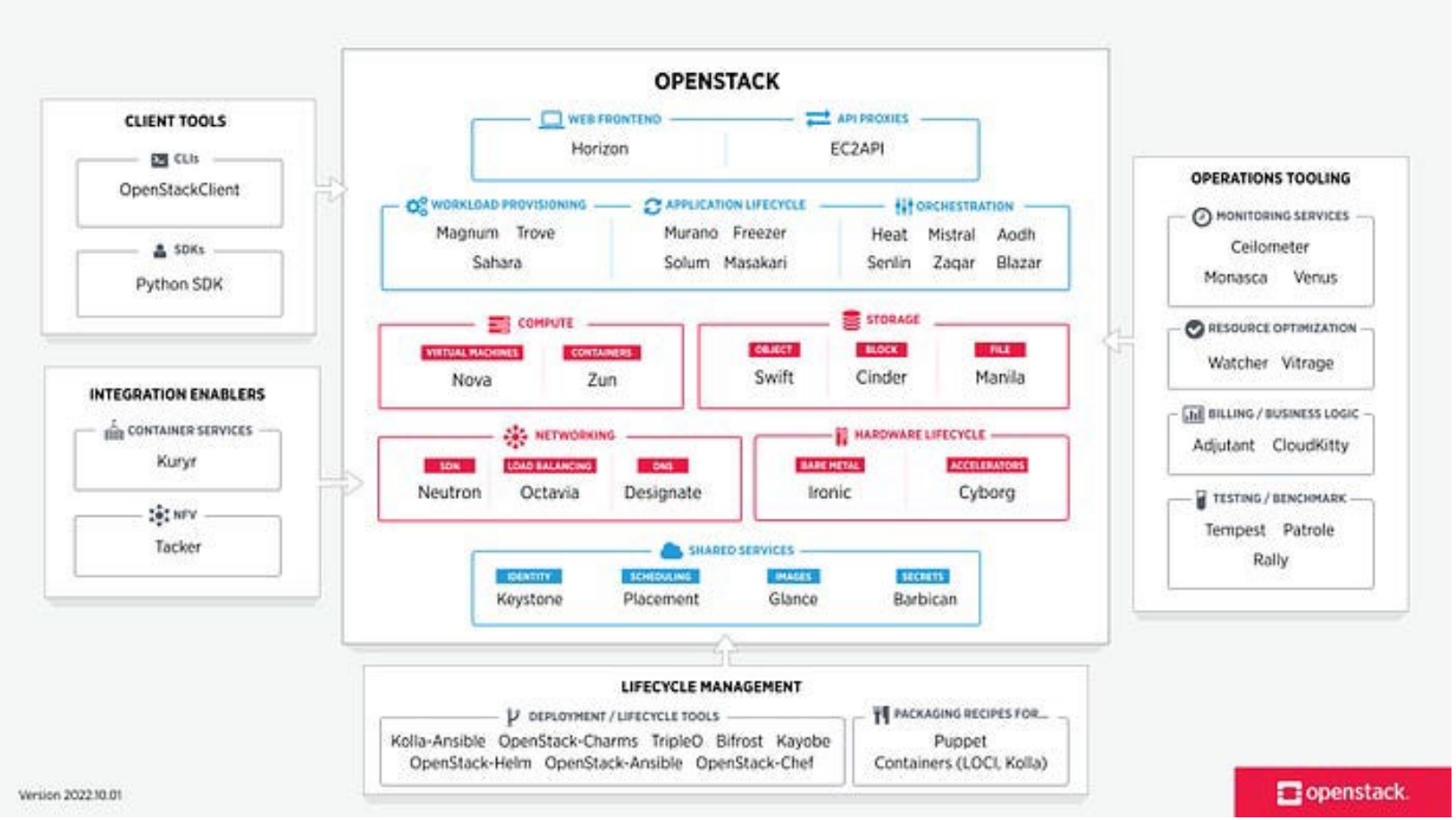
오픈스택은 파이썬 기반으로 작성이 되어 있으며, 이때 오픈스택은 Nova밖에 없었다.

이를 통해서 **네트워크/볼륨/이미지/가상머신/보안**과 같은 자원을 모두 다루다가 하나씩 서비스가 분리가 되었다. 현재 오픈스택 핵심 서비스 다음과 같다.

1. Nova
2. Keystone
3. Cinder
4. Glance
5. Heat
6. Horizon

# 오픈스택 아키텍처

## OPENSTACK ARCHITECHTURE



# 오픈스택 릴리즈

## OPENSTACK VERSION

현재 오픈스택은 **코드/년도 기반 릴리즈 이름**을 따르고 있다.

릴리즈 주기는 보통 6개월에 한번씩 릴리즈 된다.

1. **2025.1년도 첫 번째** 릴리즈
2. **2025.2년도 두 번째 릴리즈**

코드 네임은 알파벳 A부터 **도시/동물 이름**으로 시작. 현재는 보조적으로 사용하고 **년도 기반 릴리즈**가 관리 번호로 명명 및 명시되고 있다.

레드햇 오픈스택 경우에는 별도로 버전 릴리즈 관리 및 운영하고 있다.

# 오픈스택 릴리즈

## OPENSTACK VERSION

| 연도 기반 버전 | 코드네임 | 릴리즈 시기 |
|---|---|---|
| 2018.1 | Queens | 2018년 2월 |
| 2018.2 | Rocky | 2018년 8월 |
| 2019.1 | Stein | 2019년 4월 |
| 2019.2 | Train | 2019년 10월 |
| 2020.1 | Ussuri | 2020년 5월 |
| 2023.2 | Bobcat | 2023년 10월 |
| 2024.1 | Caracal | 2024년 4월 |
| 2024.2 | Dalmatian | 2024년 10월 |
| 2025.1 | Flamingo | 2025년 4월 (예정) |
| 2025.2 | G… (미정, G 시작 이름 예정) | 2025년 10월 (예정) |

# 오픈스택 릴리즈

## 오픈스택 버전

| 연도 | RHEL 메이저 GA(있을 때) | OpenStack 상반기 | OpenStack 하반기 |
|---|---|---|---|
| 2022 | **RHEL 9** — 2022-05-17 | **Yoga** — 2022-03 | **Zed** — 2022-10 |
| 2023 | — | **Antelope** — 2023-03-22 | **Bobcat** — 2023-10 |
| 2024 | — | **Caracal** — 2024-04 (SLURP) | **Dalmatian** — 2024-10-02 |
| 2025 | **RHEL 10** — 2025-05 (10.0) | **Epoxy** — 2025-04-02 (SLURP) | **Flamingo** — 2025-10-01(예정) |

# 오픈소스에서 IDC

## 오픈소스 기반 IDC

오픈스택은 퍼블릭 클라우드에서 대표적인 AWS/GCP와 같은 API기반의 플랫폼 구성 및 운영이 가능하다.

서비스 확장은 사용자 혹은 오픈스택 릴리즈에 맞추어서 가능하며, 이를 통해서 다양한 서비스를 사용자에게 제공이 가능하다.

오픈스택은 플랫폼 기반으로 서비스가 확장이 되는 구성을 가지고 있다.

이를 통해서 IaaS 및 XaaS서비스 개념을 구현 및 구성하고 있다. 또한, 오픈스택은 OpenInfra의 핵심 구성원이다. 오픈 소스에서는 인프라 및 응용 분야에 대해서 보통 두 가지 영역으로 구별하고 있다.

1. Open-Infra
2. CNCF

OpenInfra는 말 그대로 열린 인프라 아키텍처를 지향하고 있으며, CNCF는 인프라 위에서 동작하는 컴퓨팅 서비스에 대해서 정의하고 있다.

# 오픈스택 설치 도구

## 오픈스택 설치 기술 비교

현재 오픈스택은 다양한 방법으로 설치를 제공 및 지원하고 있다.

1. kolla-ansible
2. OpenStack Kayobe
3. OpenStack Bifrost
4. Helm-OpenStack

일반적으로 많이 사용하는 구성은 **`Kolla-Ansible`** 기반으로 많이 사용한다. 하드웨어 및 운영체제 관리까지 포함이 되는 경우 **`Kayobe, Bifrost`**를 많이 사용한다.

좀 더 유연한 컨트롤러 및 쿠버네티스와 통합을 위해서 **`helm-openstack`** 사용이 가능. 레드햇 오픈 시프트를 제외하고 아직 실 환경에서 사용하는 사례는 드물다. (2025년 기준)

# 오픈스택 설치 도구

## 오픈스택 설치 기술 비교

베어메탈 기반으로 구성 시, 자동화 도구가 필요한데 오픈스택은 다음과 형식으로 제공.

1. Ironic
2. Bifstor (ansible+Ironic)
3. Kayobe

레드햇 오픈스택 경우에는 현재 Director 및 OpenShift 기반으로 설치 지원.

커뮤니티 버전은 대용량 및 프로덕트 용도로 사용을 원하는 경우 Kayobe기반으로 물리적 인프라 구성 및 컨테이너화 된 오픈스택 서비스 배포 사용을 권장.

# 오픈스택 설치 도구

## 오픈스택 설치 기술 비교

| 구분 | Bifrost | Kayobe |
|---|---|---|
| 정의 | Ironic(베어메탈 서비스)를 **스탠드얼론**으로 설치·활용하는 **Ansible 플레이북 모음** | **Kolla-Ansible**을 기반으로 컨테이너화된 OpenStack을 **베어메탈에 풀스택 배포·운영**하는 오케스트레이션 |
| 주 목적 | 베어메탈 **OS 프로비저닝**(디스커버리, 인스펙션, 이미지 디플로이) 중심의 경량 자동화 | **컨트롤 플레인/워크로드 노드 OS 프로비저닝 + 컨테이너화 OpenStack 배포 + Day2 운영 작업**까지 포괄 |
| 핵심 구성 | Ansible Roles/Playbooks + Ironic **standalone** | Seed/Seed-hypervisor, **bifrost(내장 컨테이너)**, Kolla/Kolla-Ansible, 호스트 구성(네트워크·OS) |
| 동작 방식 | Ironic을 단독으로 올려 **알려진 하드웨어**에 베이스 이미지 배포 | Seed 호스트에서 **bifrost 컨테이너로 노드 등록·프로비저닝 → Kolla-Ansible로 OpenStack 서비스 컨테이너 배포** |
| 의존/연계 | Ironic, Ansible. OpenStack의 다른 컴포넌트 없이 독립 운용 가능 | 내부적으로 **bifrost를 호출**해 오버클라우드 노드를 PXE로 발견/등록 후, **Kolla-Ansible**로 서비스 배포 |

# Kolla-ansible

설치(어려움)

# kolla-ansible 설치

## kolla-ansible 준비 및 설치

아래와 같이 설치를 진행한다.

```
# dnf install -y git dbus-devel glib2-devel python3.11 python3.11-
devel python3.11-pip cmake
# python3.11 -m ven openstack
# source ~/openstack/bin/active
# pip3.11 install dbus-python docker ansible==11.* kolla-ansible
kolla
```

# kolla-ansible 설치

## 인벤토리 구성

AIO 기반으로 설치 시, 아래와 같이 구성 및 설정한다. AIO는 인벤토리 내용 변경이 필요 없다.

기본값으로 제공하는 설정 그대로 사용한다. 실 서비스 구성 시에는 절대로 all-in-one를 사용하지 않는다.

```
# mkdir -p /etc/kolla/ansible/inventory/
# cp ~/openstack/share/kolla-ansible/ansible/inventory/all-in-one
/etc/kolla/ansible/inventory/
```

# kolla-ansible 설치

## globals.yml

설치를 위해서 아래와 같이 구성한다.

```
kolla_base_distro: "rocky"
openstack_release: "2024.2"

node_custom_config: "/etc/kolla/config"
config_strategy: COPY_ALWAYS

network_interface: "ens8"
api_interface: "ens8"
storage_interface: "ens8"
```

# kolla-ansible 설치

**globals.yml**

위의 내용 계속 이어서…

```
neutron_external_interface: "ens9"
neutron_plugin_agent: "openvswitch"

enable_haproxy: "no"
enable_keepalived: "no"
enable_proxysql: "no"
enable_loadbalancer: "no"
```

# kolla-ansible 설치

**globals.yml**

위의 내용 계속 이어서…

```
kolla_enable_tls_external: "no"
kolla_enable_tls_internal: "no"

enable_cinder: "no"
enable_cinder_backup: "no"
enable_heat: "no"
enable_magnum: "no"
enable_octavia: "no"
enable_barbican: "no"
```

DUSTBOX

# kolla-ansible 설치

**globals.yml**

위의 내용 계속 이어서…

```
kolla_internal_vip_address: "192.168.10.68"
kolla_external_vip_address: "192.168.10.68"

kolla_internal_fqdn: "aio.local"
kolla_external_fqdn: "aio.local"
```

DUSTBOX

# kolla-ansible 설치

## kolla-ansible 준비 및 설치

아래와 같이 설치를 진행한다.

```
# kolla-ansible install-deps
# kolla-genpwd
# kolla-ansible bootstrap-servers
# kolla-ansible pull
# kolla-ansible certificates
# kolla-ansible prechecks
# kolla-ansible deploy
# kolla-ansible post-deploy
```

# kolla-ansible 설치

## 설치 확인

설치가 올바르게 되었는지 확인하기 위해서 아래와 같이 작업을 수행한다.

```
# source /etc/kolla/admin-openrc.sh
# openstack token issue
# openstack server list
# openstack user list
# openstack project list
```

# Packstack

설치(쉬움)

# Packstack 설치

## AIO IN SINGLE NODE

단일 노드에서 오픈스택 설치 시, 상황에 따라서 RPM 기반으로 설치가 필요하다.

Kolla-Ansible 기반으로 설치 시, 시스템 요구사항이 높기 때문에 저 사양에서는 Packstack 기반으로 설치를 권장한다.

설치가 어려운 경우, RDO기반으로 오픈스택 설치를 진행한다. 다만, RDO는 프로덕트 용도로는 사용을 권장하지 않는다.

RDO 설치 관련된 내용은 아래 사이트에서 확인이 가능하다.

https://www.rdoproject.org/deploy/packstack/

# Packstack 설치

## AIO IN SINGLE NODE

```
# hostnamectl set-hostname osp.packstack
# dnf install -y centos-release-openstack-caracal;
# setenforce 0;
# dnf install -y openstack-packstack;
# packstack --allinone
```

# 오픈스택에서 VDC

## OPENSTACK IN VDC CONCEPT

오픈스택에서 VDC 개념을 비교하면 다음과 같다.

| 구분 | OpenStack Project | VDC (Virtual Data Center) |
|---|---|---|
| **역할** | 사용자·자원의 논리적 격리 단위 (테넌트/조직/팀) | 하나의 완전한 가상 데이터센터 (Compute + Network + Storage + Policy 집합) |
| **자원 범위** | 인스턴스, 네트워크, 볼륨, 이미지, 로드밸런서 등 | 위 모든 자원 + 정책(보안/라우팅) + 템플릿(Heat) + 관리 포털 |
| **격리 수준** | 논리적 격리 (RBAC, 네트워크/보안 그룹으로 분리) | 데이터센터 수준의 논리 독립 (테넌트별 완전 독립된 DC처럼 구성) |

# 오픈스택에서 VDC

## OPENSTACK IN VDC CONCEPT

위의 내용 계속 이어서…

| 구분 | OpenStack Project | VDC (Virtual Data Center) |
|---|---|---|
| **네트워크** | 네트워크는 프로젝트 단위로 생성/소유 | VDC 내 서브넷, 라우터, 외부망, 보안정책까지 포함 |
| **관리 단위** | 주로 Keystone(인증)에서 관리 | 상위 포털 또는 SDN 컨트롤러에서 관리 (프로젝트 연동 포함) |
| **스코프** | IaaS 내부의 논리 단위 | IaaS + SDN + SDS + 관리 포털까지 확장된 논리 단위 |
| **예시** | “project = dev-team” | “VDC = dev-team의 클라우드 환경 전체” |

DUSTBOX

# 오픈스택 코어 서비스

## OPENSTACK CORE SERVICE

오픈스택 코어 서비스는 다음과 같다. 이를 통해서 추상적인 컴퓨팅 자원을 구성한다.

1. keystone
2. nova
3. glance
4. cinder

# 랩 진행 전 쉘 변수 선언

**시작 전…**

만약, RDO 기반으로 구성하는 경우 아래 변수 선언은 필요 없다. Kolla-ansible 기반으로 설치하는 경우 쉘 변수 선언 후 진행한다.

# 랩 진행 전 쉘 변수 선언

## LAB PREPARE

```
PHYSNET=physnet1
EXT_NET=public
EXT_SUBNET=public-subnet
EXT_CIDR=192.168.90.0/24
EXT_GW=192.168.90.1
EXT_POOL_START=192.168.90.100
EXT_POOL_END=192.168.90.200
EXT_DNS=8.8.8.8
```

# 랩 진행 전 쉘 변수 선언

**LAB PREPARE**

```
PRIV_NET=private

PRIV_SUBNET=private-subnet

PRIV_CIDR=192.168.100.0/24

PRIV_DNS=8.8.8.8
```

# 랩 진행 전 쉘 변수 선언

**LAB PREPARE**

```
ROUTER=r1
IMAGE_NAME=cirros-0.6.2
FLAVOR=ci.tiny
SECGRP=allow-ssh-icmp
KEY_NAME=lab-key
VM_NAME=vm1
```

# 키스톤 설명

## ABOUT KEYSTON

키스톤 서비스는 오픈스택에서 제공하는 **프로젝트/사용자/역할/그룹**과 같은 자원을 생성 및 관리한다.

AWS와 마찬가지로 모든 서비스는 **엔드포인트(Endpoint)**가 존재하며, 이를 통해서 각 서비스에 접근이 가능하다.

프로젝트 생성 후 인스턴스 생성을 위한 자원 생성 및 구성한다.

# 프로젝트 생성

## CREATE PROJECT

프로젝트 생성.

```
# openstack project create vlab-project
# openstack project list
```

# 사용자 생성

## CREATE USER

사용자 생성.

```
# openstack user create --password 'Training123!' vlab-user
# openstack user list
```

# 역할 구성

## CREATE ROLE

역할 구성. 역할은 **사용자/프로젝트 연결 및 권한 할당**한다.

```
# openstack role add --project vlab-project --user vlab-user1 student
# openstack role assignment list --user vlab-user1 --project vlab-
  project --names
```

# 글랜스 설명

## ABOUT GLANCE

오픈스택에서 인스턴스 생성 시 사용하는 이미지. 이 이미지를 통해서 가상머신을 생성 및 구성한다.

테스트를 위해서 저 사양 이미지 cirros를 사용해서 구성한다. 오픈스택 이미지는 아래 주소에서 다운로드가 가능하다.

https://docs.openstack.org/image-guide/obtain-images.html

# 이미지 업로드

## CREATE IMAGE

이미지 업로드 및 확인 명령어. 이미지는 cirros를 업로드 한다.

```
# openstack image create cirros \
  --file /var/images/cirros.qcow2 \
  --disk-format qcow2 \
  --container-format bare
# openstack image list
# openstack image show rocky9-base
# curl -L -o /tmp/cirros.qcow2 https://download.cirros-
cloud.net/0.6.2/cirros-0.6.2-x86_64-disk.img
# openstack image create "$IMAGE_NAME" --public --disk-format qcow2 -
-container-format bare --file /tmp/cirros.qcow2
```

# 뉴트론 설명

## ABOUT NEUTRON

뉴트론 서비스는 오픈스택에서 다음과 같은 기능 제공한다.

1. Software Defined Network(SDN, OVS)
2. Network Function Virtualization(NFV,OVN)

뉴트론에서 NFV를 완벽하게 사용하려면 **OVN(Open Virtual Network)**기반으로 구성을 권장한다. 여기는 간단한 네트워크 구성만 사용하기 때문에 **OpenVSwitch** 기반으로 구성한다.

# 네트워크 생성

## CREATE NETWORK RESOURCES

외부 네트워크 생성 및 확인.

```
# openstack network create "$EXT_NET" \
  --share --external \
  --provider-network-type flat \
  --provider-physical-network "$PHYSNET"
# openstack subnet create "$EXT_SUBNET" \
  --network "$EXT_NET" \
  --subnet-range "$EXT_CIDR" \
  --gateway "$EXT_GW" \
  --allocation-pool start=$EXT_POOL_START,end=$EXT_POOL_END \
  --dns-nameserver "$EXT_DNS" \
  --no-dhcp
```

DUSTBOX

# 네트워크 생성

## CREATE NETWORK RESOURCES

내부 네트워크 생성 및 확인.

```
# openstack network create "$PRIV_NET"
# openstack subnet create "$PRIV_SUBNET" \
  --network "$PRIV_NET" \
  --subnet-range "$PRIV_CIDR" \
  --dns-nameserver "$PRIV_DNS"
```

DUSTBOX

# 네트워크 생성

## CREATE NETWORK RESOURCES

라우터 및 게이트웨이 설정.

```
# openstack router create "$ROUTER"
# openstack router set "$ROUTER" --external-gateway "$EXT_NET"
# openstack router add subnet "$ROUTER" "$PRIV_SUBNET"
```

# 노바설명

## ABOUT NOVA

노바 서비스는 컴퓨팅 서비스. 이를 통해서 다음과 같은 서비스 구성이 가능하다.

1. 가상머신(Instance, VM)
2. 컨테이너 서비스
3. 쿠버네티스 서비스

이와 같은 서비스들은 노바를 통해서 구성된다. AWS와 비교하면 EC2 서비스와 동일한 서비스.

# 보안그룹 생성

## CREATE SECURITY GROUP

보안그룹 생성.

```
# openstack security group create "$SECGRP"
# openstack security group rule create --proto icmp "$SECGRP"
# openstack security group rule create --proto tcp --dst-port 22:22
"$SECGRP"
```

# 키페어 생성

## CREATE KEYPAIR

키페어 생성.

```
# openstack keypair create "$KEY_NAME" > ~/.ssh/$KEY_NAME.pem
# chmod 600 ~/.ssh/$KEY_NAME.pem
```

# 플레이버 생성

## CREATE FLAVOR

플레이버 생성.

```
# openstack flavor create "$FLAVOR" --ram 256 --disk 1 --vcpus 1
```

# 인스턴스 생성

## CREATE INSTANCE

인스턴스 생성.

```
# openstack server create "$VM_NAME" \
  --flavor "$FLAVOR" \
  --image "$IMAGE_NAME" \
  --nic net-id=$(openstack network show "$PRIV_NET" -f value -c id) \
  --security-group "$SECGRP" \
  --key-name "$KEY_NAME"
```

# VDC(서비스 패키징 및 배포)

CONTAINER

KUBERNETES

# 포드만

고급 컨테이너

# 기술 활용(컨테이너)

## ABOUT CONTAINER

리눅스 기반의 컨테이너 기술은 다음과 같은 물음으로 시작이 되었다.

**"가상머신보다 가볍게 격리 및 운영"**이라는 요구사항에서 시작이 되었다. x86 시스템에서 최초의 컨테이너 개념은 FreeBSD의 Jail에서 시작이 되었으며, 그 기술 중 chroot라는 구성 요소가 리눅스에 도입이 되었다.

초기 리눅스는 chroot로 사용하였으나, 패키징 및 분리 운영에 어려운 부분이 많았으며, 이를 해결하기 위해서 여러가지 도구를 개발하기 시작하였다.

그 결과 우리가 많이 사용하는 Docker가 릴리즈가 되었으며, 그 이후에 Podman으로 고수준 런타임 애플리케이션들이 출시가 되었다.

# 기술 활용(컨테이너)

## 컨테이너 활용

현재 오픈소스 컨테이너 엔진은 포드만으로 전환 중. **docker-ee, docker-ce**는 여전히 사용이 가능하나, 오픈소스 표준 엔진 및 런타임을 제공하는 포드만을 권장. 포드만은 다음과 같은 표준형식을 제공한다.

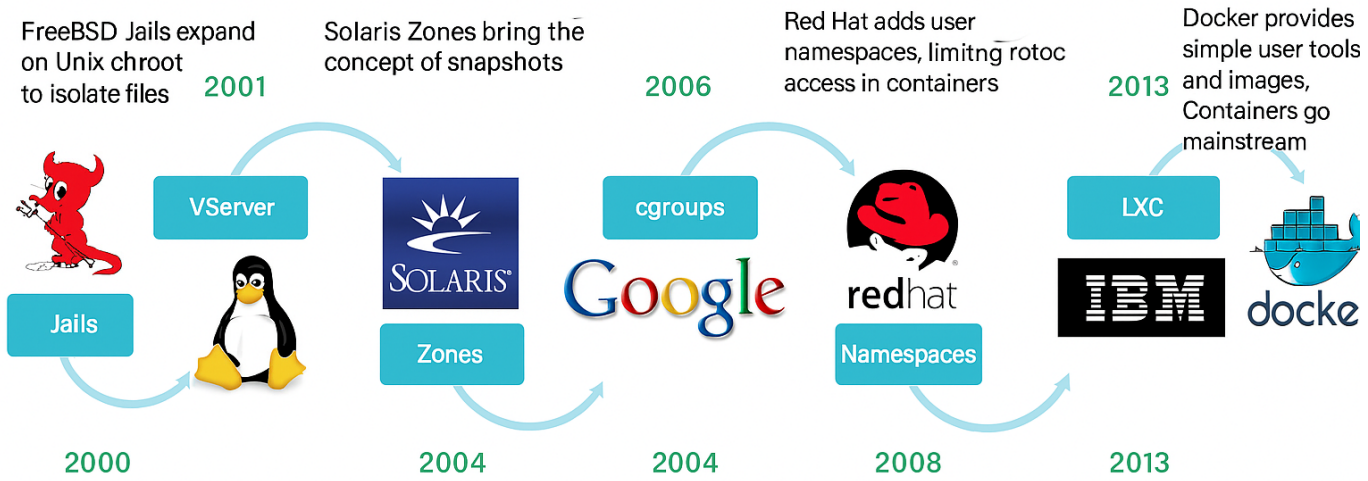
- OCI(open container initiative)
- CNI(container network interface)
- CRI(container runtime interface)

도커는 도커에서 사용하는 관리 방법 및 구조를 사용하고 있기 때문에, OCI형식으로 이미지는 제공하지만, 네트워크 및 런타임 인터페이스는 도커 자체 구조를 사용한다. 이 부분에 대해서는 뒤에서 더 다루도록 한다.

앞으로 Podman은 포드만이라고 표현한다.

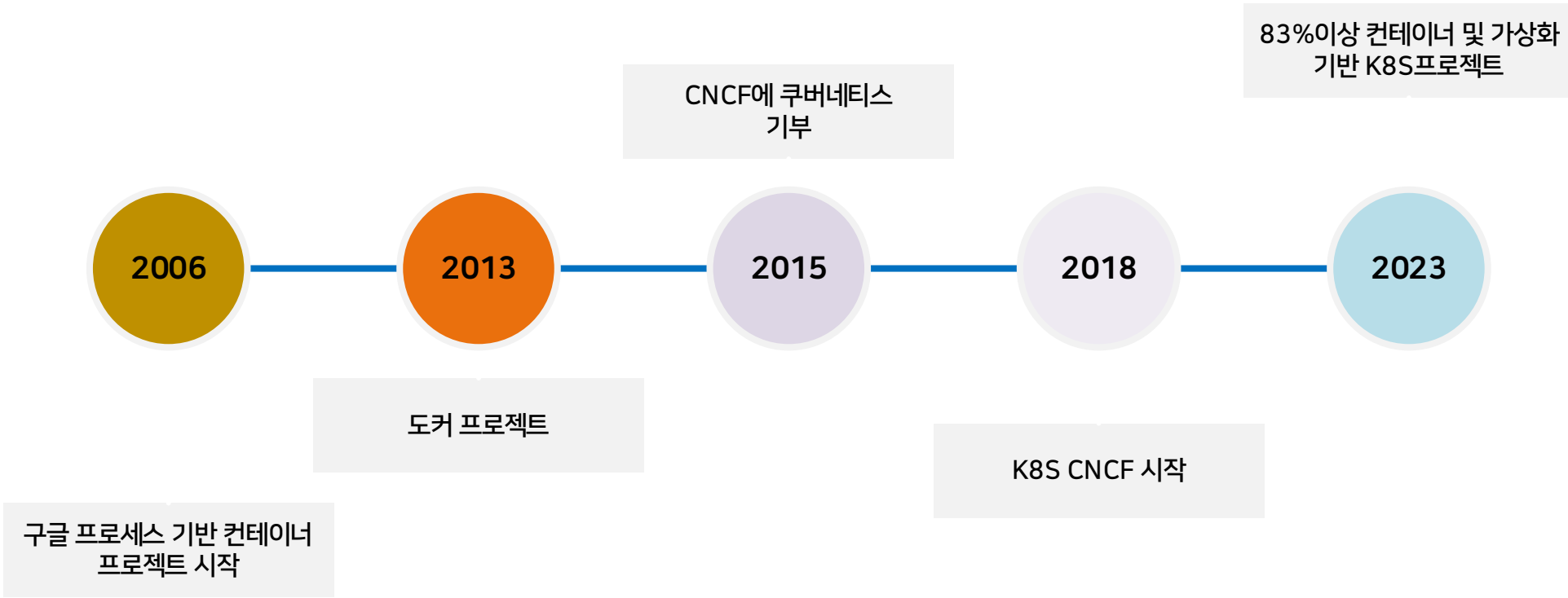
# 기술 활용(컨테이너)

## ABOUT CONTAINER



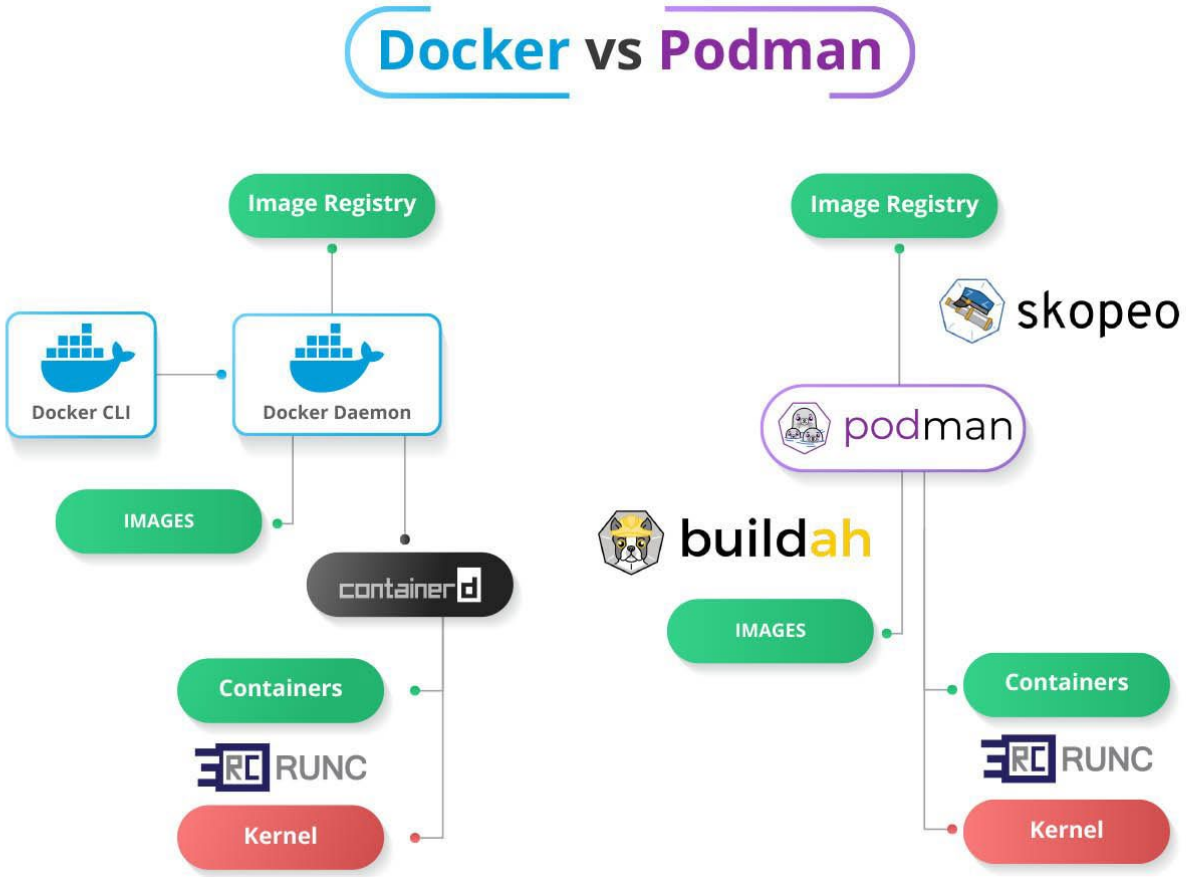
# 기술 활용(컨테이너)

## ABOUT CONTAINER



# 기술 활용(컨테이너)

## ABOUT CONTAINER






2025-12-26

# 명령어

## PODMAN COMMAND

이미 도커를 사용한 경험이 있는 사용자는, 기본 명령어 부분은 크게 차이가 없기 때문에 아래 내용에 대해서 학습은 따로 필요 없다. 하지만, 몇몇 부분에서 다르게 동작하기 때문에 해당 부분에 대해서 한번 더 확인한다.

현재 표준 컨테이너는 다음과 같은 패키지로 나누어 진다.

1. 표준 컨테이너 런타임 엔진 포드만
2. 표준 컨테이너 도구(skopeo, buildah…)

# INSTALLATION PODMAN

## 포드만 설치

포드만 설치는 RHEL기반의 클론 버전인 ROCKY LINUX에서 설치한다.

```
$ dnf search container-tools podman
$ sudo dnf install podman epel-release -y
$ podman images
$ podman pods ls
$ podman containers ls
```

# Podman POD

## POD 개념

**Pod**는 배포판 조금씩 다를 수 있다. 여기서 말하는 POD 혹은 Pod는 쿠버네티스에서 말하는 개념이다. 레드햇 계열은 RHLE 7이후부터는 포드만에서 **kata라는 Pod 애플리케이션을** 사용하고 있으며, 이전 버전들은 쿠버네티스에서 사용하는 /**pause**기반으로 사용하고 있다.

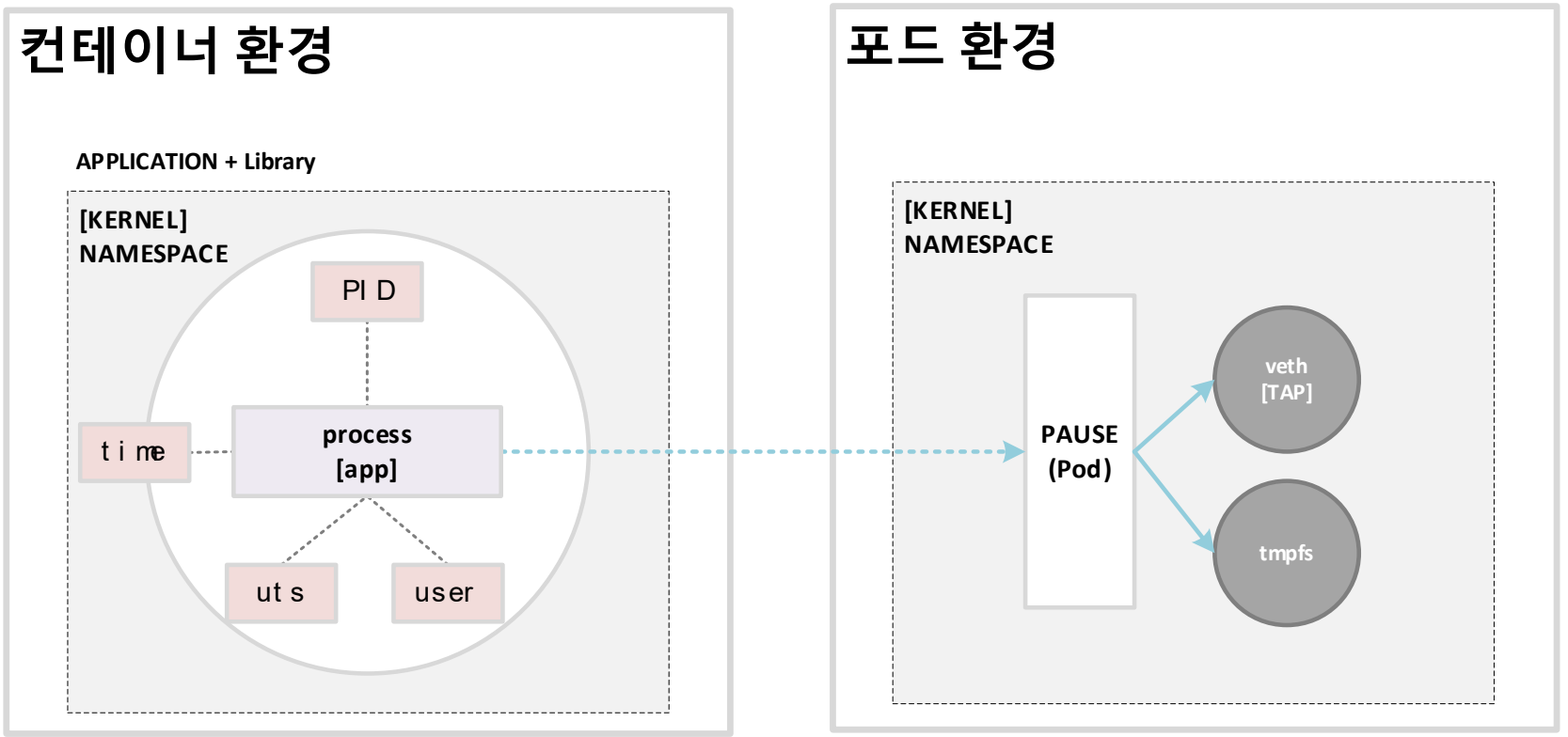
Pod는 쿠버네티스에서 제시한 개념이며, pause는 Pod를 구현하기 위해서 사용하는 컨테이너 애플리케이션이다.

/pause 프로그램이 동작하는 컨테이너 자원은 "인프라 컨테이너(infra-container)"라고 불리기도 한다. 이 목차에서는 간단하게 생성 및 실행만 하고, 컨테이너와 연동해서 활용하는 부분은 후에 진행하도록 한다.

# POD 개념

## POD 개념

포드는 뒤에서 좀 더 다루기 때문에 Pod개념에 대해서 간단하게 설명한다. 아래 그림을 참고 한다.



# command:pod

## POD 생성

생성 및 동작중인 Pod를 확인한다. 일반적으로 사용하는 Pod는 쿠버네티스에서 사용하는 /pause기반으로 POD를 구성한다.  아래 명령어로 간단하게 Pod생성 및 구성이 가능하다.

```
$ podman pod create
$ podman pod ls
$ podman pod create test
```

# command:pod

## POD 생성

생성한 Pod는 아래와 같이 실행한다. Pod는 애플리케이션 컨테이너가 연결이 되지 않으면, 시작하지 않는다.

```
$ podman pod start <ID> or <NAME>
POD ID        NAME                STATUS      CREATED         INFRA ID
# OF CONTAINERS
5de2fc474c1c  gallant_mahavira    Running     2 minutes ago
e223bc8ef0b7  1
```

# command:pod

## POD 구성

실행된 Pod는 아래와 프로세스 확인이 가능하다. 또한, 이미지를 파일로 저장해서 어떻게 구성이 되어 있는지 내부 구성원을 확인한다.

```
$ ps -ef | grep -i pause
pause
$ podman save <POD_CONTAINER_IMAGE> -o pod.tar
$ mkdir kata
$ tar xf pod.tar -C kata/
$ tar xf <FILENAME>.tar
```

# command:run

## 컨테이너 실행

컨테이너 런타임에서 컨테이너를 생성한다. 명령어 및 옵션은 아래처럼 사용한다.

| | |
|---|---|
| `--rm` | 컨테이너가 중지가 되면, 컨테이너를 제거한다. |
| `-v` | 디렉터리를 바인딩 하는 옵션. 일반적으로 컨테이너 내부의 프로그램이 저장하는 외부 장치에 이 옵션을 같이 사용한다. 혹은 런타임에서 사용하는 볼륨을 사용하기도 한다. |
| `-d` | detach 옵션은 사용자 영역에서 실행하는 프로그램을 사용자 영역(userspace)에서 분리한다. |
| `-i` | interactive 옵션은 사용자가 대화형으로 stdout/stderr를 podman 명령어를 통해서 주고 받는다. |
| `-t` | 컨테이너에 tty장치를 생성해서 터미널을 구성한다. 여기에서 생성하는 tty는 pseudo-tty장치이며, 실제로 존재하지 않는 가상의 터미널이다. |
| `-p` | 컨테이너 애플리케이션이 사용하는 포트를 내부에서 외부로 바인딩하는 옵션. |

```
$ podman run -ti --rm centos:stream9-minimal bash
```

# command:run

## 컨테이너 실행

컨테이너를 실행하는 명령어. 컨테이너 런타임을 통해서 컨테이너 생성 및 실행한다. 컨테이너 실행 시, 백그라운드로 실행 혹은 포그라운드로 실행이 가능하다.

```
$ podman run -d centos:stream9-minimal
$ podman run -it --rm centos:stream9-minimal bash
[root@9bab61b80300 /]# ip link
bash: ip: command not found
[root@9bab61b80300 /]# ls /bin | wc -l
185
[root@9bab61b80300 /]# cd /dev/
[root@9bab61b80300 dev]# ls
console  core  fd  full  mqueue  null  ptmx  pts  random  shm  stderr
stdin  stdout  tty  urandom  zero
[root@9bab61b80300 dev]#
```

DUSTBOX

# command:port

## 컨테이너 포트 확인

컨테이너에서 사용하는 포트를 호스트로 바인딩한다. 왼쪽은 '**컨테이너 포트**', 오른쪽은 '**호스트 포트**'를 명시한다. 도커는 docker-proxy를 통해서 컨테이너 포트를 구성한다.

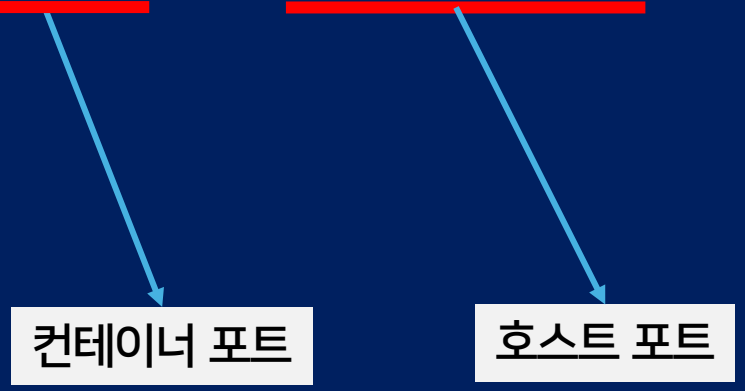
컨테이너 이미지가 사용하는 포트 정보를 확인하고 싶으면 아래와 같이 실행.

```
$ podman container run -d -p 8080:8080 --rm --name centos-httpd
centos7/httpd-24-centos7
$ podman image inspect centos7/httpd-24-centos7
> Ports:
```

# command:RUN(PORT)

## 컨테이너 포트 확인

```
# podman port -l or port <CONTAINER ID/NAME>
80/tcp -> 0.0.0.0:8080
```






2025-12-26

# command:STOP

## 컨테이너 중지

컨테이너를 중지하기 위해서 stop명령어를 실행한다. 중지가 잘 되었는지 ps명령어를 통해서 확인한다.

```
$ podman stop <CONTAINER ID> <CONTAINER NAME>
$ podman container stop
$ podman ps
$ podman container ps
```

# command:STOP

## 컨테이너 중지

실행중인 모든 컨테이너를 중지하기 위해서는 `--all` 명령어 옵션을 통해서 중지가 가능하다. 올바르게 컨테이너가 중지되지 않으면 직접적으로 중지 신호를 명시 할 수 있다.

```
$ podman container stop --all
$ podman container stop -t <WAIT_FOR_TIME> <CONTAINER ID OR NAME>
$ podman container kill -t <SIGKILL NUMBER> <CONTAINER ID OR NAME>
```

# command:START

## 컨테이너 시작

중지된 컨테이너를 다시 실행하기 위해서 위의 명령어로 실행한다.

```
$ podman container start <ID> <CONTAINER NAME>
```

# command:INSPECT

## 컨테이너 검사

- 동작중인 pod/container의 정보를 확인한다.
- 정보는 JSON형태로 제공한다.
- 이미지도 역시 inspect명령어로 확인이 가능하다.

컨테이너 런타임별로 동작 방법이 조금 다를 수 있으니 주의가 필요하다. 대다수 컨테이너 정보는 'inspect'를 통해서 확인.

1. 도커와 포드만에서 출력되는 내용은 조금은 다름.
2. 도커 경우에는 사용하는 버전 및 런타임 수준에 따라서 명령어가 다를 수 있다.

```
$ podman image inspect <ID> <NAME>
          container == podman ps
          pod       == podman ps -a
```

# command:INSPECT

## 컨테이너 검사

JSON에서 특정 딕 및 리스트에서 값을 찾아서 화면에 출력한다.

```
$ podman container inspect centos-httpd --format '{{ .Config.Cmd }}'
[/usr/bin/run-httpd]
$ podman container inspect centos-httpd --format
'{{ .Config.StopSignal }}'
15
$ podman container inspect centos-httpd --format
'{{ .NetworkSettings.Ports }}'
map[8080/tcp:[{ 80}] 8443/tcp:[]]
```

# command:PS

## 컨테이너 목록

둘 다 동일한 컨테이너 목록을 출력. 권장은 **podman container ls**명령어 사용을 권장.

```
$ podman container ls
$ podman ps
```

# command:PS

## 컨테이너 목록

컨테이너화 되어서 동작하는 프로세스 목록을 확인한다.

```
$ podman container ps
a98f2a616cfa  quay.io/xinick/containerlab/httpd:latest  /bin/sh -c
httpd ...  2 hours ago     Up 2 hours ago     0.0.0.0:8080->80/tcp
gallant_Mahavira
$ podman container ls
$ podman pod ls
$ podman ps
```

# command:PS

## 컨테이너 목록

ps 명령어는 process의 약자이며, 컨테이너에서 동작중인 프로세스 목록을 확인한다.

| 옵션 | 설명 |
|---|---|
| `--size` | 현재 사용중인 컨테이너의 크기를 확인한다. 다만, 시간이 좀 걸린다. |
| `--sort` | 정렬 기준의 필드를 선택합니다. |
| `--noheading` | 맨 위의 제목 필드를 출력하지 않는다. |
| `--latest(-l)` | 마지막에 생성된 컨테이너를 출력한다. |
| `--watch(-w)` | 실시간으로 프로세스 목록을 명시된 초만큼 갱신한다. |

# command:RM

## 컨테이너 목록

컨테이너를 제거한다. 다만, 컨테이너 제거 시, 컨테이너에서 사용하는 프로세스는 반드시 중지가 되어 있어야 한다.

```
$ podman container rm centos-httpd
          pod
```

| 옵션 | 설명 |
|---|---|
| `--all` | 모든 컨테이너를 제거한다. |
| `--force` | 컨테이너를 강제로 제거한다. 다만, 사용 중이면, 컨테이너가 중지가 될 때 제거가 된다. |

# command:RMI

## 이미지 관리

런타임 엔진에서 관리하는 이미지를 제거한다.

```
$ podman images
$ podman rmi <IMAGE NAME OR ID>
$ podman rmi --all
```

| 옵션 | 설명 |
|---|---|
| **--all** | 모든 이미지를 한번에 제거한다. 다만, 사용중인 경우 제거가 되지 않을 수 있으며, 사용 중에 강제로 제거하면 컨테이너가 올바르지 않는 동작을 한다. |

# command:EXEC

## 명령어 실행

컨테이너 내부에 명령어 실행. 컨테이너에서 생성된 파일이나 혹은 바인딩된 디렉터리를 확인 시 사용한다.

```
$ podman exec -i centos-httpd bash -c 'cat >
/var/www/html/index.html' << _EOF
Hello World
_EOF
```

# command:EXEC

## 명령어 실행

대화형 모드는 반드시 '-i', '-t'옵션을 통해서 접근이 가능하다.

```
$ podman exec centos-httpd cat /var/www/html/index.html
$ curl localhost:8080
```

# command:RUN(EXEC)

**명령어 실행**

컨테이너 실행자마다 다르지만, 컨테이너 실행자는 'tty', 'pts'를 지원하지 않는다.

```
$ podman exec -it <POD_ID> /bin/bash
```

# command:commit

## 런타임 이미지 커밋

현재 사용중인 컨테이너를 이미지로 저장한다. 기존에 사용하고 있는 컨테이너를 이미지로 커밋을 원하는 경우, 가급적이면 컨테이너 중지 후 커밋을 권장. 절대적으로 테스트 용도로만 사용.

```
$ podman run -d -it --rm --name centos-9-httpd-mariadb -p 8081:80
quay.io/centos/centos:stream9 bash
$ podman exec -it centos-9-httpd bash
> dnf install httpd mariadb-server -y
$ podman stop centos-9-httpd-mariadb
$ podman commit centos-9-httpd-mariadb centos-9-httpd-mariadb-
commited
Getting image source signatures
Copying blob 53498d66ad83 skipped: already exists
Copying blob e41246bc487f skipped: already exists
Copying blob d320f5da4f70 skipped: already exists
```

# command:logs

## 로그

컨테이너 로그를 확인한다. 컨테이너가 시작하면 출력되는 **표준 출력/오류**를 런타임으로 전달한다.

```
# podman container logs centos-httpd
=> sourcing 10-set-mpm.sh ...
=> sourcing 20-copy-config.sh ...
=> sourcing 40-ssl-certs.sh ...
---> Generating SSL key pair for httpd...
AH00558: httpd: Could not reliably determine the server's fully qualified
domain name, using 10.88.0.19. Set the 'ServerName' directive globally to
suppress this message
[Sun Feb 05 06:47:31.399659 2023] [ssl:warn] [pid 1] AH01909:
10.88.0.19:8443:0 server certificate does NOT include an ID which matches the
server name
```

# command:logs

**로그**

혹은 **journalctl**명령어로 확인이 가능하다. 컨테이너 런타임이 동작하면서 발생하는 정보 및 오류내용을 확인한다.

```
# podman container ls
> e553c7a47eb4
> 743aa7b01a52
# journalctl PODMAN_ID=743aa7b01a52
> EDT m=+0.038686754 container create
> EDT m=+0.132793251 container init
```

# command:mount

## 오버레이 디스크 마운트 확인

컨테이너에 **마운트** 혹은 **바인딩(mount --bind)**이 된 자원을 확인한다.

```
$ podman container mount
77da2c1e24e8
/var/lib/containers/storage/overlay/1b003f63da8767dce7dacdcc431045ef46264a003047efeb9c032061e1e7506c/merged
a98f2a616cfa
/var/lib/containers/storage/overlay/9e781c2758890d0d7e9328cd6c69674cec7a1c7840ed06e379d847637507abc1/merged
```

# command:pause/unpause

## 일시중지

동작중인 컨테이너를 일시적으로 중지한다. 다시 정상적으로 동작하기 위해서 **unpause**로 **pause**상태를 원상복구 한다.

```
$ podman container pause centos-httpd
77da2c1e24e84283547e6b7defcdf80231b2833d28c18fa19bf5a0b222c88e85

$ podman container ls
a98f2a616cfa  quay.io/xinick/containerlab/httpd:latest  /bin/sh -c httpd ...
3 hours ago  Up 3 hours ago  0.0.0.0:8080->80/tcp  gallant_mahavira
$ podman container unpause
```

# command:prune

## 사용하지 않는 이미지 제거

사용하지 않는 컨테이너를 모두 제거한다.

```
$ podman container prune
WARNING! This will remove all non running containers.
Are you sure you want to continue? [y/N]
```

# command:rename

## 이름 변경

컨테이너 이름을 변경한다.

```
$ podman container rename centos-httpd centos-apache
$ podman container ps
> /bin/sh -c httpd ...  29 seconds ago  Up 29 seconds ago  0.0.0.0:8080-
>80/tcp  centos-apache
```

# command:restart

## 재시작

동작중인 컨테이너를 중지 후 다시 시작한다. 이 명령어는 'stop', 'start'가 동시에 시작하는 명령어.

```
$ podman container restart centos-apache
842293d27f7d4bfe62c433fb50bb1a477944335cd65a8d94ee97742be46b1b2b
```

# command:checkpoint

## 시점저장

현재 상태를 체크포인트로 저장 합니다. 다만, 체크포인트 생성을 하려는 컨테이너는 반드시 `--rm`명령어를 사용하면 안된다.

```
$ podman container run -d --name checkpoint-httpd -p 8088:8080
quay.io/fedora/httpd-24:latest
$ podman container checkpoint --keep checkpoint-httpd
1dcb10cfe96a65ffe63f2a8fba97d93ad7a1133b09372e5f4d347c4a5cc737bf
$ podman container checkpoint --export=checkpoint-httpd.tar.gz
checkpoint-httpd
podman container checkpoint --export=checkpoint-httpd.tar.gz --all
```

# command:restore

## 시점복구

체크포인트로 중지된 컨테이너를 실행한다. 기존의 체크포인트 컨테이너를 실행하거나 혹은 이미지를 불러와서 복구한다.

```
$ podman container restore checkpoint-httpd
1dcb10cfe96a65ffe63f2a8fba97d93ad7a1133b09372e5f4d347c4a5cc737bf
$ podman container restore --import checkpoint-httpd.tar.gz
```

# command:stats

## 컨테이너 상태

컨테이너 상태를 확인 합니다. 이 정보는 systemd에서 cgroup으로 모니터링한다. 매 5초마다 정보를 갱신한다.

```
$ podman container stats
```

```
ID            NAME          CPU %       MEM USAGE / LIMIT  MEM %       NET IO           BLOCK IO      PIDS        CPU TIME    AVG CPU %
1dcb10cfe96a  centos-httpd  0.04%       32.48MB / 2.728GB  1.19%       796B / 1.836kB   32.77kB / 0B  213         179.744ms   0.09%
```

# command:top

## 컨테이너 상태

네임스페이스에서 동작하는 프로세스의 CPU 사용률 및 동작 시간을 확인한다.

```
$ podman container top <CONTAINER_ID> <NAME>
```

```
[root@container ~]# podman container top centos-httpd
USER        PID         PPID        %CPU        ELAPSED          TTY         TIME        COMMAND
root        1           0           0.000       7m2.357571898s   ?           0s          httpd -DFOREGROUND
apache      2           1           0.000       7m2.357666755s   ?           0s          httpd -DFOREGROUND
apache      3           1           0.000       7m2.357688937s   ?           0s          httpd -DFOREGROUND
apache      4           1           0.000       7m2.35770685s    ?           0s          httpd -DFOREGROUND
apache      5           1           0.000       7m2.357724764s   ?           0s          httpd -DFOREGROUND
```

# command:mount

## 바인딩 해제

바인딩된 루트 **파일시스템**(root filesystem, /var/lib/containers/storage)디렉터리를 컨테이너에서 해제한다. 컨테이너 재시작은 필요 없으며, 네임스페이스 공간을 통해서 연결이 해제된다.

```
$ podman container unmount
$ podman inspect centos-httpd-volume --format
'{{ .HostConfig.Binds }}'
[htdocs:/var/www/html/:rw,rprivate,nosuid,nodev,rbind]
$ podman container unmount centos-httpd-volume
```

# command:wait

## 상태코드 확인

컨테이너의 실행 상태를 코트로 화면에 출력한다. 시스템에서 사용하는 시스템 리턴 코드와 동일하게 0,1 그리고 125와 같은 숫자로 표현한다.

```
$ podman container wait
$ podman container wait --condition=running centos-httpd
-1
```

# 쿠버네티스

컨테이너 오케스트레이션

# 쿠버네티스

## THE KUBERNETES

오픈소스 커뮤니티에서 **포드만 컨테이너 런타임** 사용 이유는 다음과 같다.

1. 쿠버네티스에서 사용하는 이미지 빌드
2. 표준 런타임 기반으로 컨테이너 애플리케이션 테스트
3. Pod기반으로 구성하여 쿠버네티스에서 동작이 가능한지 확인
4. systemd의 service형태로 컨테이너 구현을 원하는 경우(derecated)

# 쿠버네티스

## THE KUBERNETES

쿠버네티스 본래 구글에서 내부 서비스 관리 용도로 사용함. 구글에서 **Borg**라는 이름으로 2014년도에 릴리즈 하였음. 구글에서는 대략 2006년도에서 내부적으로 서비스 및 컨테이너 관리 방법에 대해서 논의가 됨.

구글은 2014년도에 깃-허브에 처음 코드를 릴리즈 하였으며, 현재 기준 11년 동안 꾸준히 릴리즈가 됨.

현재 CNCF에서는 **Serverless** 및 **Knative**를 구현하기 위해서 쿠버네티스를 사용하고 있으며, 현재는 가상머신도 같이 컨테이너와 동일하게 POD기반으로 관리한다.

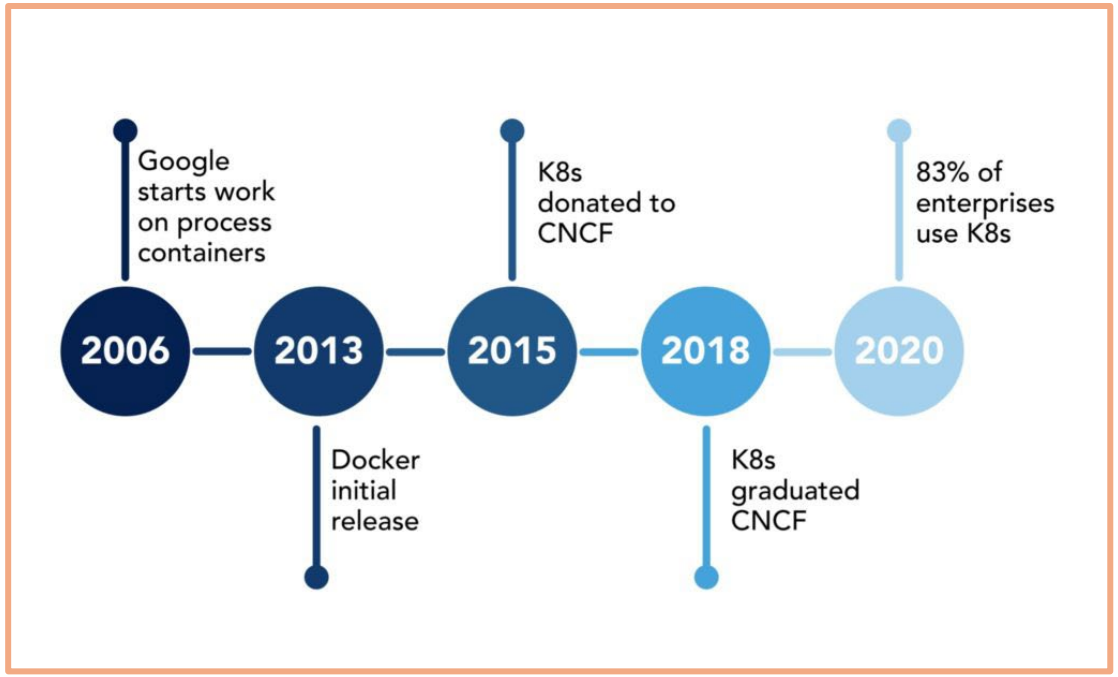
오픈스택은 쿠버네티스를 API기반으로 지원 및 자원 연동을 할 수 있도록 지원하고 있으며, 현재는 STARINGX를 OpenInfra를 통해서 지원하고 있다.

# 쿠버네티스

## THE KUBERNETES

쿠버네티스는 구글에서 **Borg**라는 이름으로, 내부적으로 클러스터 관리 및 배포 용도로 사용하였다.

초기 **Borg**는 **C++**로 작성이 되었으나, 쿠버네티스로 넘어 오면서 Go언어 기반으로 작성이 되었다. 초기 릴리즈는 2015년도, 발표는 2014년도에 이루어 졌다.



# 가상화/컨테이너

## THE KUBERNETES

쿠버네티스는 자체적으로 런타임을 지원하지 않는다. 여기서 말하는 런타임은(runtime) 컨테이너 이미지에 있는 내용을 메모리에 불러와서 프로세서를 격리 및 추적하여, 호스트와 분리 운영이 가능하도록 환경을 구성을 한다.

이러한 이유 때문에 가상머신과 많은 혼동이 있는데, 둘은 엄연히 큰 차이가 있다. 가상머신은 **링 구조(ring structure)**를 생성 및 재구성하기 때문에, 더 복잡하고 독립적인 자원을 가지고 동작한다.

가상머신은 런타임에서 동작하는 구조가 아닌, 일반적으로 하이퍼바이저(hypervisor)나 혹은 가속기(kernel module level)에서 동작한다.

컨테이너는 이와 반대로 가상화 같은 하드웨어 수준의 기술이 필요하지 않으며, 커널 수준에서 격리 및 추적 기술 기반으로 프로세스를 관리한다.

이러한 이유로, 가상머신에서 사용하던 기능을 전부 컨테이너로 마이그레이션이 가능하지 않기 때문에, 서비스 아키텍트 및 구성에 따라서 플랫폼을 선택 및 구성하면 된다.

# 자원 구성

## 쿠버네티스 자원

쿠버네티스가 사용하는 물리적 자원은 다음과 같다.

1. Controller(AKA Master)
2. Compute(AKA minion/worker)
3. Infra Node(Optional)

위의 두 개 용어, master, minion, worker는 오랫동안 IT계열에서 많이 사용하였지만, 차별적인 단어로 현재는 취급이 되고 있어서 통상적으로 controller, compute와 같은 단어로 대체가 되어가고 있다.

# 자원 구성

## 쿠버네티스 자원

쿠버네티스는 오래전 버전은 모든 서비스가 hosted형태로 구성이 되어 있었지만, 현재 쿠버네티스는 kubelet제외한 나머지 서비스는 containerized가 되었다. 이 말은 kubelet은 Proxy/Bootstrap/Monitor를 hosted상태에서 담당하고, 나머지는 서비스는 Static-POD가 되었다.

Static-POD는 우리가 알고 있는 POD와 동일하지만, 쿠버네티스에서 구성 시 기본적으로 생성하는 POD 서비스 이다. Static-POD는 다음과 같다.

- kube-scheduler
- kube-apiserver
- kube-ControllerManager
- CoreDNS
- ETCD

# 릴리즈

## 릴리즈 규칙

The Kubernetes project maintains release branches for the most recent three minor releases (1.32, 1.31, 1.30). Kubernetes 1.19 and newer receive approximately 1 year of patch support. Kubernetes 1.18 and older received approximately 9 months of patch support.

Kubernetes versions are expressed as x.y.z, where x is the major version, y is the minor version, and z is the patch version, following Semantic Versioning terminology.

자세한 내용은 "https://kubernetes.io/releases/"에서 확인이 가능하다.

# 쿠버네티스

## 설치

```
swapoff -a
sed -i '/swap/d' /etc/fstab

setenforce 0
sed -i 's/^SELINUX=enforcing/SELINUX=disabled/' /etc/selinux/config

systemctl stop firewalld && sudo systemctl disable firewalld
```

## 설치

```
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF

modprobe overlay
modprobe br_netfilter
```

# 쿠버네티스

## 설치

```
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables  = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward                 = 1
EOF
sysctl --system
```

# 쿠버네티스

## 설치

```
KUBERNETES_VERSION=v1.32

CRIO_VERSION=v1.32
```

## 설치

```
cat <<EOF | tee /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.k8s.io/core:/stable:/$KUBERNETES_VERSION/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.k8s.io/core:/stable:/$KUBERNETES_VERSION/rpm/repodata/repomd.xml.key
EOF
```

# 쿠버네티스

## 설치

```
cat <<EOF | tee /etc/yum.repos.d/cri-o.repo
[cri-o]
name=CRI-O
baseurl=https://download.opensuse.org/repositories/isv:/cri-o:/stable:/$CRIO_VERSION/rpm/
enabled=1
gpgcheck=1
gpgkey=https://download.opensuse.org/repositories/isv:/cri-o:/stable:/$CRIO_VERSION/rpm/repodata/repomd.xml.key
EOF
```

# 쿠버네티스

## 설치

```
dnf install -y container-selinux

dnf install -y cri-o kubelet kubeadm kubectl

systemctl start crio.service
```

# 기본명령어

쿠버네티스

# 쿠버네티스 활용

## 기본 명령어

쿠버네티스 운영에 필요한 간단한 명령어를 학습한다.

# 설명

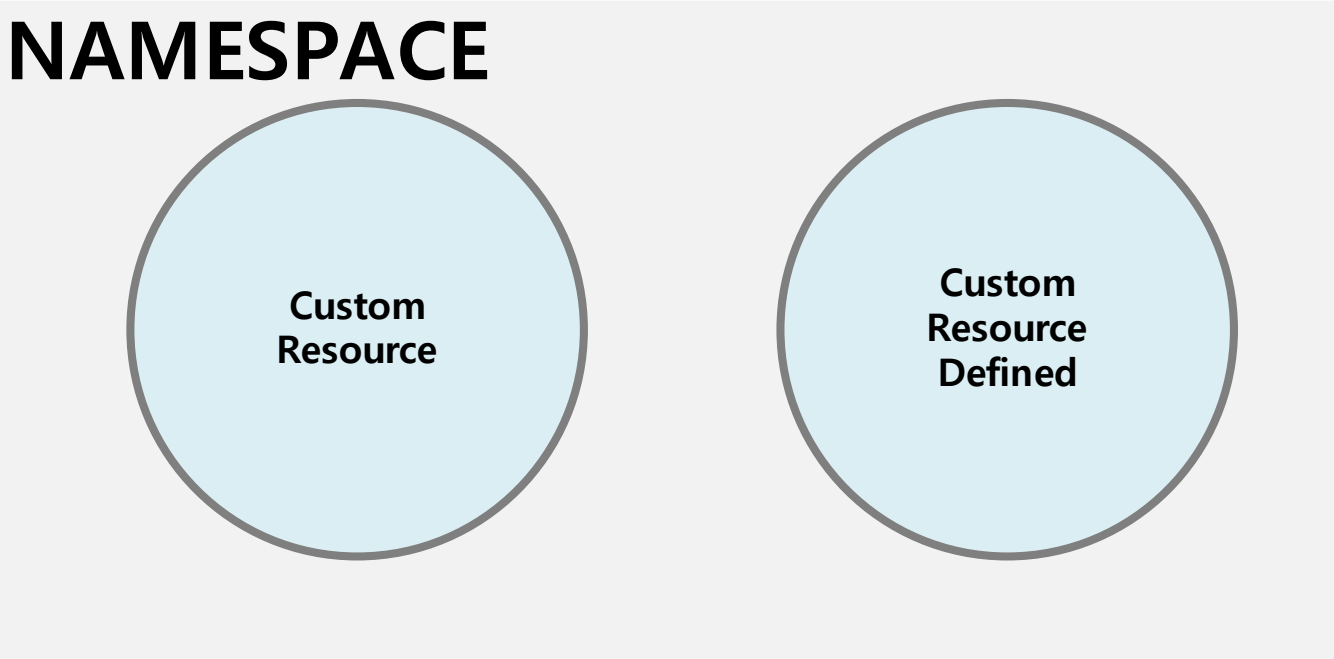
## 네임스페이스 설명

기본적으로 명령어 사용 및 활용방법에 대해서 이야기 한다. 여기서 다루는 자원은 쿠버네티스에서 매우 중요하며, 꼭 필요한 자원 및 개념이다.

1. 네임스페이스
2. POD
3. YAML/JSON

# NAMESPACE

## 네임스페이스 설명



# NAMESPACE

## 네임스페이스 생성

오픈 시프트에서는 프로젝트, 쿠버네티스는 네임스페이스라고 부른다. 자원을 격리를 하기 위해서 사용하는 개념. 다음과 같은 명령어로 생성이 가능.

```
# kubectl create namespace testnamespace
```

YAML파일로 생성하기 위해서 다음과 같이 실행한다.

```
# kubectl create namespace testnamespace --dry-run=client --output=yaml > testnamespace.yaml
```

# NAMESPACE

## 네임스페이스 제거

네임 스페이스가 올바르게 제거가 되지 않는 경우, 아래와 같이 제거가 가능하다.

```
# REMOVENS=<NAMESPACE_NAME>
# kubectl get namespace "$REMOVENS" -o json \
  | tr -d "\n" | sed "s/\"finalizers\": \[[^]]\+\]/\"finalizers\": []/" \
  | kubectl replace --raw /api/v1/namespaces/"$REMOVENS"/finalize -f -

# kubectl delete namespace $REMOVENS
```

# POD

## 개념 설명

POD의 뜻은 **"고래 때"**를 가지고 있으며, 여러 컨테이너를 POD로 하나로 묶어서 서비스를 구성한다.
POD 개념을 구성하기 위해서, **`/pause`**라는 애플리케이션을 사용한다. 이 애플리케이션 각 회사마다 다르게 사용하며, 레드햇 경우에는 **`cataonit`**를 기본으로 사용한다.

**`/pause`** 애플리케이션은 호스트 시스템의 /bin/init 혹은 /bin/systemd와 동일한 기능을 하며, 애플리케이션에 요청하는 **시스템 콜 및 라이브러리 콜을 커널 네임스페이스(Kernel Namespace)를 통해서 구성**한다.

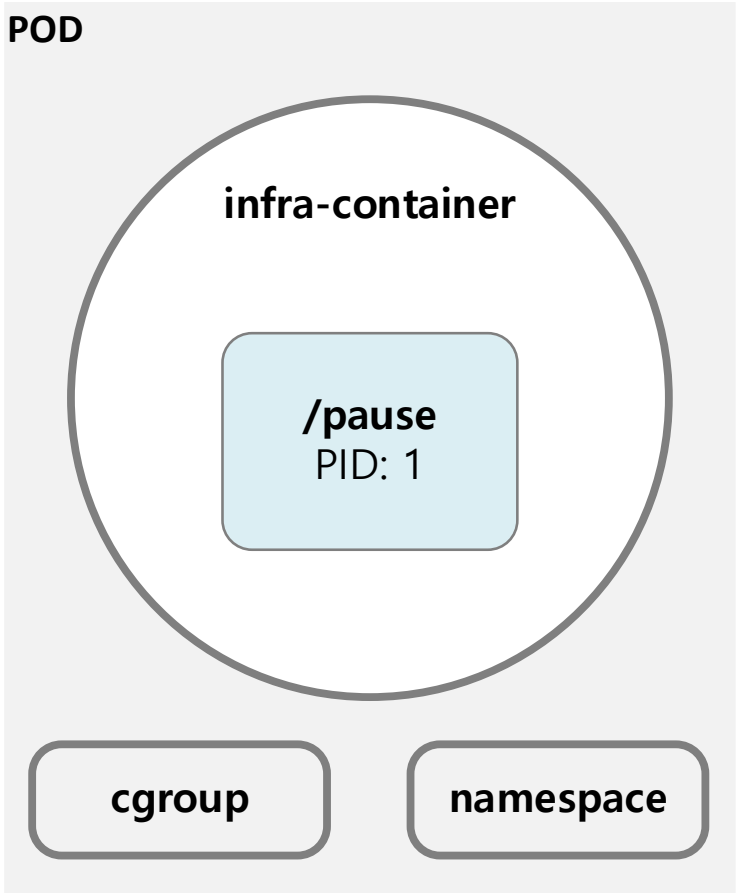
POD에서 사용하는 자원은 커널에서 C-Group를 통해서 **CPU/MEM/DISK/NET**와 같은 자원을 제어한다. 현재 쿠버네티스는 CPU/MEM만 지원하며, 나머지 자원 영역에 대해서는 지원하지 않는다.

# POD

## 개념 설명

# POD

## 컨테이너에서의 고래 때(?)



# POD/CGROUP

## 애플리케이션 실행

CGROUP부분을 정리하면 다음과 같다.

| 항목 | 내용 |
|---|---|
| **표준화 문제** | 디스크, 네트워크 자원은 하드웨어 및 커널 구현 방식이 다양해 표준화가 어려움 |
| **기술적 복잡성** | 디스크 I/O 및 네트워크 대역폭의 계량화와 격리는 애플리케이션별 요구가 달라 어려움 |
| **커뮤니티와 연구 동향** | Extended Resources를 통한 GPU 지원처럼, 관련 기능에 대한 논의중 |
| **결론** | 추후 디스크와 네트워크 자원에 대한 지원 가능성은 있으나, 표준화와 기술적 도전 |

# RUN

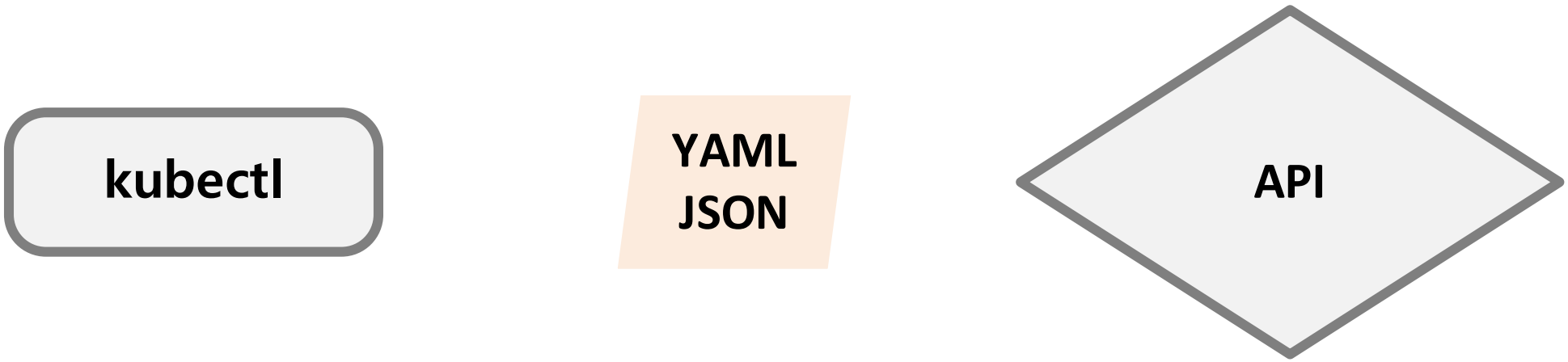
## 애플리케이션 실행

RUN명령어는 자원 실행하는 옵션은 다음과 같이 지원한다.

| 옵션 | 설명 |
|---|---|
| --image nginx pod-nginx | 간단하게 POD실행한다. 보통 맨 뒤쪽은 POD의 이름으로 사용한다. |
| --dry-run=client<br>--output=yaml pod-nginx | 실제로 생성하지 않고, kubectl명령어에서 YAML형태로 자원을 생성 후 화면에 출력한다. |
| --stdin(-i)<br>--tty | 컨테이너를 실행하면서 가상 TTY를 통해서 대화형으로 시작한다. |
| --port | POD에서 사용할 포트번호를 명시한다. |
| --expose | POD를 SERVICE로 노출한다. 기본적으로 ClusterIP형태로 되며, 이를 사용하기 위해서 앞서 이야기한 --port가 선언이 되어 있어야 한다. |
| --rm | Podman의 --rm옵션과 똑같다. 컨테이너 혹은 POD가 중지가 되면 제거가 된다. |

# RUN

## 애플리케이션 실행



# RUN

## 애플리케이션 실행

명령어는 다음과 같이 사용한다.

```
# kubectl run --image=node1.example.com/testapp/php-blue-green:v1
php-pod
# kubectl get pod
# kubectl delete pod php-pod
# kubecrl get pod
# kubectl run --image=node1.example.com/testapp/php-blue-green:v1 --
dry-run=client --output=yaml php-pod-v1 > php-pod-v1.yaml
# ls -l php-pod.yaml
```

# 설명

## 생성 방법 차이

create/apply그리고 API의 차이점은 다음과 같다.

| 방법 | 주요 특징 및 차이점 | 사용 시나리오 |
|---|---|---|
| kubectl create | ① 선언적 구성이 아니라 즉시 생성<br>② 이미 존재하는 경우 에러 발생<br>③ 간단한 리소스 생성에 적합 | 초기 리소스 생성, 빠른 테스트, 임시 리소스 생성 |
| kubectl apply | ① YAML에 "last-applied-configuration" 주석을 남겨 추후 업데이트시 비교 및 병합<br>② 변경된 부분만 업데이트<br>③ 리소스가 없으면 생성, 있으면 업데이트 | 지속적인 구성 관리, GitOps 방식, 선언적 인프라 관리 |
| YAML/JSON를 API로 생성 | ① REST API를 직접 호출 (예: POST, PUT 요청)<br>② 클라이언트(스크립트, SDK 등)를 통해 세밀한 제어 가능<br>③ 자동화, 커스텀 컨트롤러에서 활용됨 | 프로그램적 리소스 생성, CI/CD 파이프라인, 커스텀 오퍼레이터 개발 |

# 설명

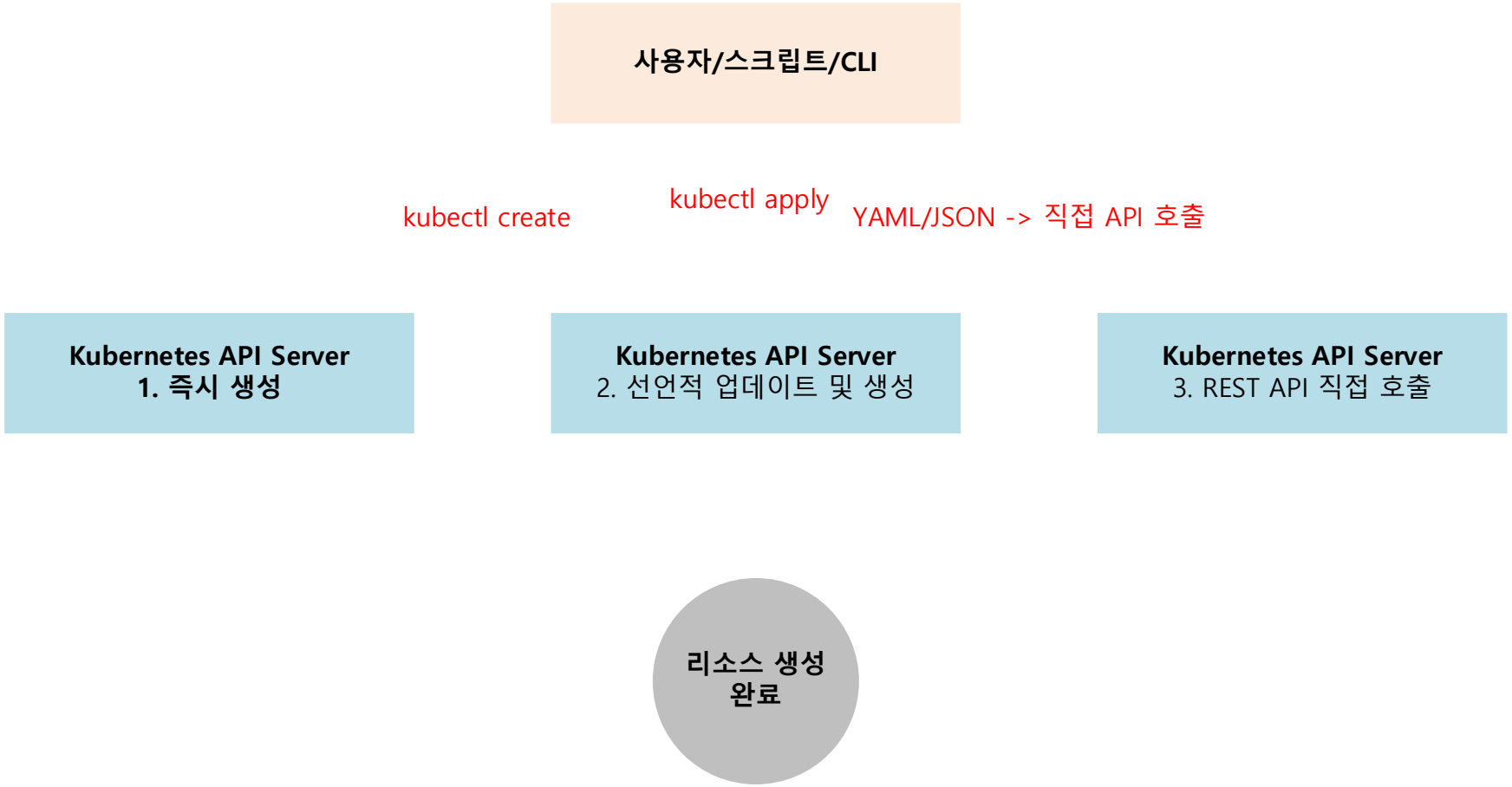
## 생성 방법 차이

앞에 내용 이어서…

| 항목 | kubectl apply -k | kubectl create -k |
|---|---|---|
| 작동 방식 | 선언적 방식으로 리소스 구성 및 업데이트<br>(last-applied-configuration 기록) | 명령 실행 시 한 번 리소스를 생성<br>(이미 존재하면 에러 발생) |
| 목적 | **선언적 관리:** 이미 생성된 리소스를 업데이트하거나, 리소스가 없으면 생성<br>지속적인 구성 관리에 유리 | **최초 생성:** 새로운 리소스를 한 번 생성할 때 사용<br>기존 리소스 업데이트에는 적합하지 않음 |
| 업데이트 처리 | 변경사항을 감지하여 패치 방식으로 업데이트<br>GitOps 등 지속적 관리에 용이 | 이미 생성된 리소스가 있으면 재생성 시 에러가 발생함 |
| 사용 시나리오 | 리소스의 장기적인 관리 및 변경 추적이 필요할 때 (선언적 구성 관리) | 단발성 리소스 생성 또는 테스트 시 사용 |

# 설명

## 생성 방법 차이




DUSTBOX

# CREATE

## 생성 방법 차이

명령어는 다음과 같이 사용한다.

```
# kubectl create -f php-pod-v1.yaml
# kubectl get pod
php-pod
# kubectl run pod --image=nginx pod-ngnix --output=yaml \
--dry-run=client > pod-nginx.yaml
# kubectl create -f pod-nginx.yaml
```

# CREATE

## 생성 방법 차이

계속 이어서…

```
# kubectl create deployment --image=nginx deploy-nginx --output=yaml
\
--dry-run=client > deploy-nginx.yaml
# kubectl create -f deploy-nginx.yaml
# kubectl create service --type=NodePort --output=yaml --dry-
run=client > svc-nginx.yaml
# kubectl create -f deploy-nginx.yaml
```

# APPLY

## 생성 방법 차이

apply명령어는 create와 거의 동일하다. 다만 차이점은 지속적으로 자원 업데이트가 가능하다. 간단하게 아래와 같이 작성한다.

```
# kubectl create deployment --image=nginx:1.26.3 deploy-nginx \
--output=yaml --dry-run=client > deploy-nginx.yaml
# kubectl apply -f deploy-nginx.yaml
# kubectl get -f deploy-nginx.yaml
# kubectl create deployment --image=nginx:1.27.4 deploy-nginx \
--output=yaml --dry-run=client > deploy-nginx.yaml
# kubectl apply -f deploy-nginx.yaml
```

# GET

## 확인

get명령어를 통해서 자원 확인이 가능하다.

```
# kubectl get pods
# kubectl get pods --namespace <NAMESPACE_NAME>
# kubectl get deployment -A
# kubectl get deployment --all-namespaces
# kubectl get pods -w
```

# GET

## 확인

앞 내용 계속 이어서…

```
# kubectl get pods -o=json
# kubectl get pod -l <LABEL_NAME>=<LABEL_VALUE>
# kubectl describe -f deploy-nginx.yaml
# kubectl create -f deploy-nginx.yaml
# kubectl describe -f deploy-nginx.yaml
```

# DELETE

## 삭제

자원 제거 시, 사용하는 명령어. YAML이나 혹은 자원을 명시하여 제거가 가능하다.

```
# kubectl delete -f deploy-nginx.yaml
# kubectl delete pod/deploy-nginx
```

# 다중 컨테이너 설명

## 개념

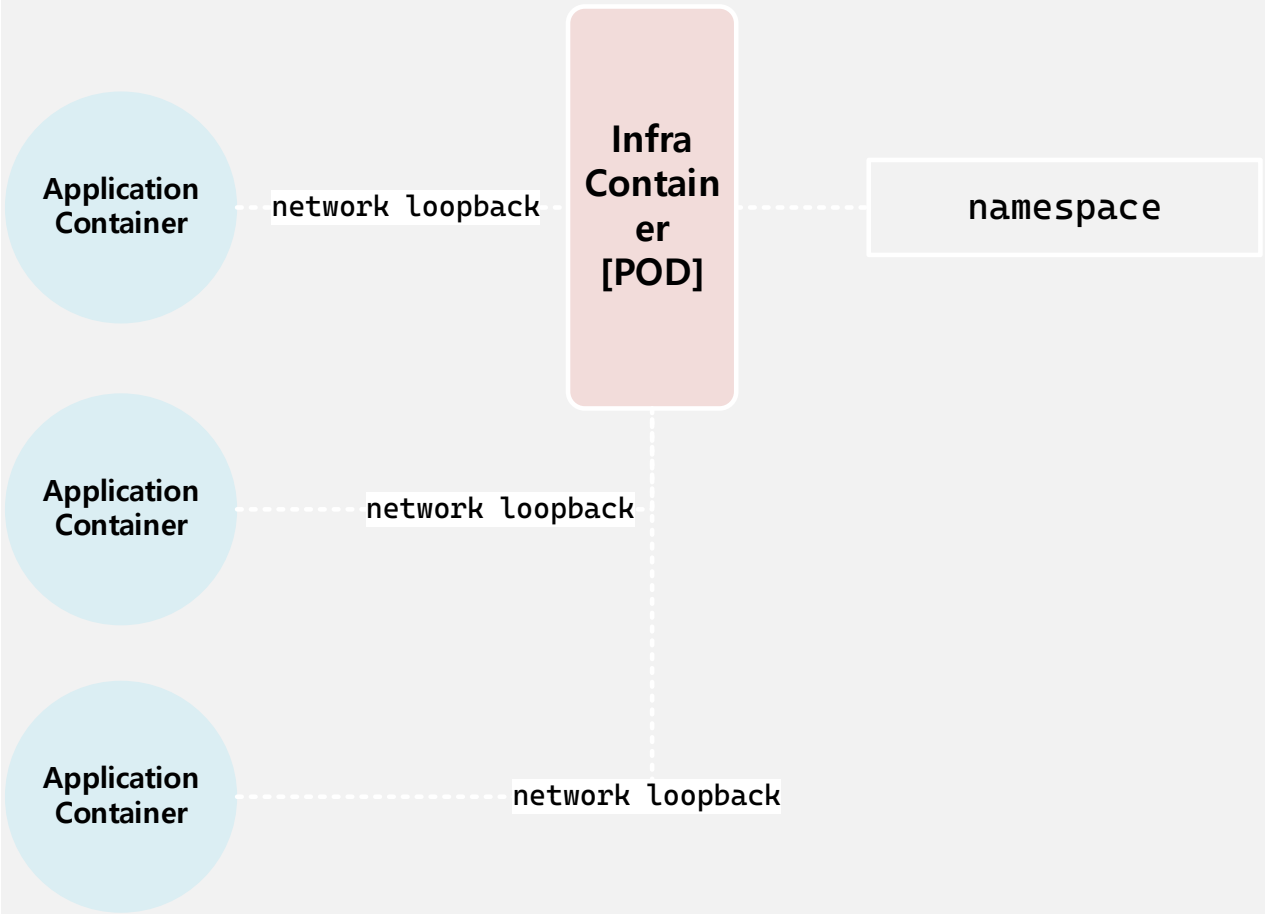
다중 컨테이너는 한 개의 POD에 하나 이상의 컨테이너가 구성이 되어있다.

POD에 연결된 컨테이너들은 자신이 사용하는 자원 **`disk, memory, cpu`** 그리고 **`network`**같은 부분들은 POD를 통해서 위임한다.

여기서 표현하는 위임은 POD가 실제 물리적 자원을 관리 하는 게 아닌, 네임스페이스를 통해서 POD가 관리하게 된다.

# 다중 컨테이너 설명

## 개념


Application Container
network loopback
Infra Container [POD]
namespace
Application Container
network loopback
Application Container
network loopback

# 다중컨테이너 구성 및 사용 방법

## 개념

다중 컨테이너 구성은 간단하다. POD생성 후, 컨테이너를 POD에 연결(attached)하면 된다. POD에 연결된 컨테이너는 POD에서 몇몇 자원을 컨테이너 대신 소유(share, own) 및 제공(provider)을 한다.

생성하기 위해서는 YAML파일 생성이 필요하다. 쉽게 생성하기 위해서 POD파일 생성 후, 설정을 추가한다.

```
# kubectl run --image=nginx wordpress --dry-run=client --output=
yaml > wordpress.yaml
# vi wordpress.yaml
```

# 다중컨테이너 구성 및 사용 방법

## 개념

다중 컨테이너 구성을 위해서 **containers:**에 다음과 같이 한 개 이상의 컨테이너 이미지를 선언한다.

```
---
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: wordpress
  name: wordpress
```

# 다중컨테이너 구성 및 사용 방법

## 개념

계속 이어서…

```
spec:
  containers:
  - image: nginx
    name: wordpress-web
  - image: mariadb
    name: wordpress-db
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
```

# 다중컨테이너 구성 및 사용 방법

## 개념 적용 및 확인

위에서 작성한 내용을 클러스터에서 생성한다.

```
# kubectl create -f wordpress.yaml
# kubectl get -f wordpress.yaml
```

# 다중컨테이너 구성

**작성 방법**

혹은 아래처럼 컨테이너에서 명령어 실행이 가능하다. 아래와 같이 파일 생성, 후 명령어를 추가해서 실행한다.

구성은 아래 슬라이드 참고를 한다

```
# kubectl run --image=alpine --dry-run=client --output=yaml run-pod --- sleep 50 > run-pod.yaml
```

DUSTBOX

# 다중컨테이너 생성 #1

**작성 방법**

다중 컨테이너 생성을 위해서 아래와 같이 구성한다.

```
# vi multi-container-1.yaml
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: run-pod
  name: run-pod
```

# 다중컨테이너 생성 #1

**작성 방법**

위의 내용 계속 이어서…

```
spec:
  containers:
  - args:
    - sleep
    - "50"
    image: alpine
    name: run-pod-1
  - args:
    - echo
    - "hello world"
    image: alpine
    name: run-pod-2
```

# 다중컨테이너 생성 #2

**작성 방법**

혹은 아래와 같이 생성이 가능하다.

```
# vi multi-container-2.yaml
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: run-pod
  name: run-pod
```

# 다중컨테이너 생성 #2

**작성 방법**

위의 내용 계속 이어서…

```
spec:
  containers:
  - args: ["-c", "sleep 10"]
    command: ["sh"]
    image: alpine
    name: run-pod-1

  - args: ["-c", "echo 'hello world'"]
    command: ["sh"]
    image: alpine
    name: run-pod-2
```

# 설명

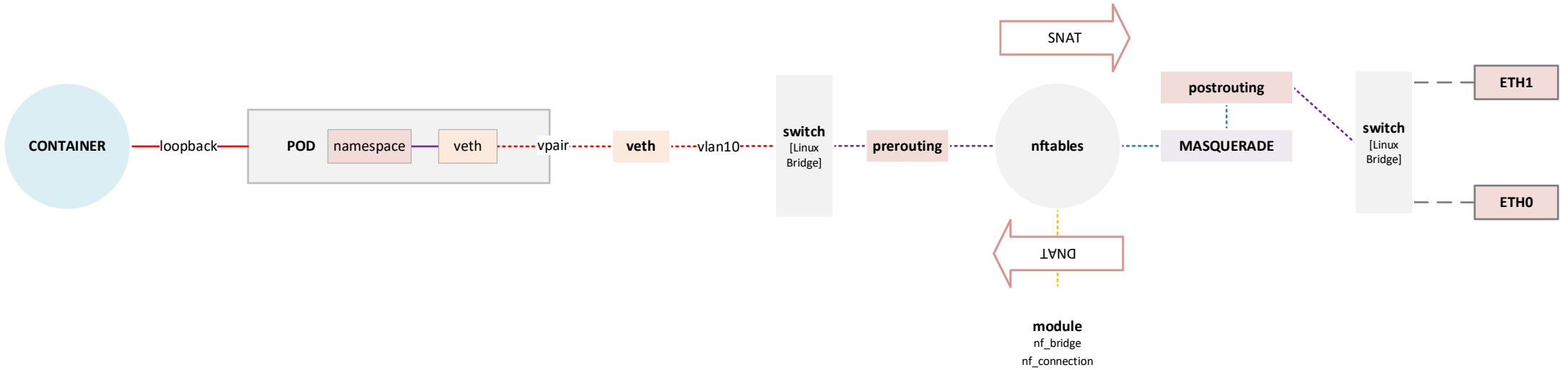
## 서비스

| 항목 | 설명 | 사용 예시 |
|---|---|---|
| ClusterIP | 클러스터 내부에서만 접근 가능한 가상 IP를 할당해 내부 통신을 지원합니다. | 내부 마이크로서비스 간 통신, 클러스터 내 API 서버 접근 |
| NodePort | 각 노드의 고정 포트를 통해 외부에서 접근할 수 있도록 노출합니다. | 외부 접근이 필요한 테스트 환경이나 개발 단계의 서비스 |
| LoadBalancer | 클라우드 제공업체의 로드 밸런서를 이용하여 외부 접근을 위한 단일 IP를 할당합니다. | 프로덕션 환경에서 안정적인 외부 접근 제공 |
| ExternalName | DNS 이름을 통해 클러스터 외부의 서비스를 참조할 수 있도록 합니다. | 외부 데이터베이스나 API를 내부 서비스처럼 사용하고자 할 때 |
| Headless | 클러스터 IP가 할당되지 않아 DNS SRV 레코드를 통해 각 Pod의 IP를 직접 반환, Pod의 개별 주소를 노출합니다. | StatefulSet, 분산 시스템 등에서 Pod 간 직접 통신이나 서비스 디스커버리 목적으로 사용 |

# 설명

## 서비스

기본적인 바닐라 상태의 서비스에서는 다음과 같은 구조로 트래픽이 전달 및 제어가 된다.






2025-12-26

# CLUSTERIP

## 서비스

클러스터 아이피는 다음과 같이 구성이 가능하다.

```
# vi my-nginx-service.yaml
apiVersion: v1
kind: Service
metadata:
name: my-nginx
labels:
    run: my-nginx
spec:
```

# CLUSTERIP

## 서비스

위의 내용 계속…

```
ports:
- port: 80
    protocol: TCP
    targetPort: 8080
selector:
    run: my-nginx
# kubectl apply -f my-nginx-service.yaml
```

# EXPOSE(ClusterIP 명시)

서비스

```
# vi service-clusterip.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-multisvc
  labels:
    run: my-nginx
```

# EXPOSE(ClusterIP 명시)

서비스

```
spec:
  ports:
  - name: normal
    port: 80
    protocol: TCP
    targetPort: 8080
```

# EXPOSE(ClusterIP 명시)

서비스

```
  - name: secure
    port: 82
    protocol: TCP
    targetPort: 8082
  - name: monitoring
    port: 83
    protocol: TCP
    targetPort: 8083
```

# EXPOSE(ClusterIP 명시)

서비스

```
  clusterIP: 10.0.23.24
  type: NodePort
  selector:
    run: my-nginx
```

# NODEPORT

## 서비스

노드 포트는 다음과 같이 구성한다.

```
# kubectl create service nodeport --tcp=8080:80 --node-port=30013 -o yaml --dry-run=client test-httpd > nodeport-test-httpd.yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: test-httpd
  name: test-httpd
```

# NODEPORT

## 서비스

```
spec:
  ports:
  - name: 8080-80
    nodePort: 30013
    port: 8080
    protocol: TCP
    targetPort: 80
  selector:
    app: test-httpd
  type: NodePort
```

# MULTIPORT

## 서비스

한 개 이상의 아이피 노출이 필요한 경우, 아래와 같이 가능하다.

```
# vi multiport-service.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: my-nginx-multisvc
  labels:
    run: my-nginx
```

# MULTIPORT

## 서비스

위의 내용 계속…

```
spec:
  ports:
  - name: normal
    port: 80
    protocol: TCP
    targetPort: 8080
```

# MULTIPORT

## 서비스

위의 내용 계속…

```
  - name: secure
    port: 82
    protocol: TCP
    targetPort: 8082
  - name: monitoring
    port: 83
    protocol: TCP
    targetPort: 8083
  selector:
    run: my-nginx
```

# 명령어(EXPOSE)

## 서비스

명령어로 처리가 가능하다.

```
# kubectl create deployment my-deployment --image=nginx --port=8080
deployment.apps/my-deployment created
# kubectl expose deployment my-deployment --port=80 --target-port=8080 --name=my-service
service/my-service exposed
# kubectl get svc my-service
NAME          TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
my-service    ClusterIP   10.96.123.45   <none>        80/TCP     2m
```

# PODMAN -> K8S

## 마이그레이션

포드만에서 구성한 자원을 K8S 노드로 마이그레이션 한다.

현재 대다수 애플리케이션은 Docker 기반으로 구성 및 실험이 되다 보니, 쿠버네티스로 애플리케이션 배포 시, 문제가 많이 발생한다.

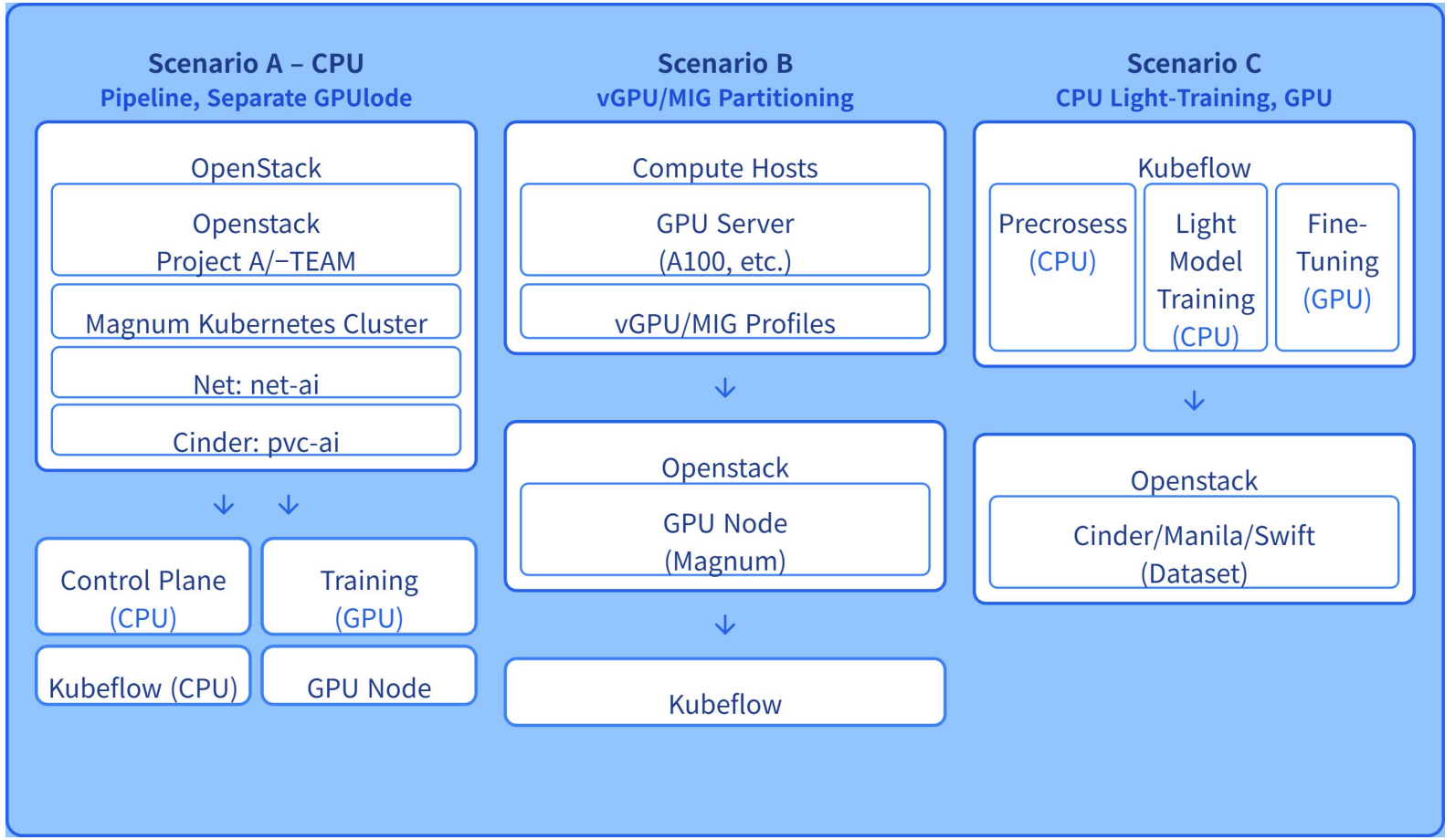
이러한 문제를 줄이기 위해서 처음부터 포드만 기반으로 컨테이너 **패키징/테스트/검증을** 권장한다.

# 폐쇠형 LLM

OpenLLM

# 컨테이너+폐쇠형 LLM

## SCENARIO FOR LLM



# 컨테이너+폐쇠형 LLM

## LLM 설명

폐쇠형 LLM은 보통 두 가지 LLM 엔진 기반으로 구성 및 구현을 한다.

1. llama.cpp
2. OpenVINO

대표적으로 많이 사용하는 LLM은 흔히 **라마**라고 부르는 llama.cpp를 보통 이야기 한다. 컨테이너 기반으로 llama.cpp 에이전트를 구성 후 사용하도록 한다.

먼저, 제일 많이 사용하는 컨테이너 기반으로 라마 서버를 구성 후 앤서블 플레이북 기반으로 오픈스택 및 쿠버네티스를 제어 및 운영이 가능하도록 한다.

# 목적

## LLM WITH VDC/SDDC

모든 인프라는 대다수가 명령어로 운영이 된다. 하지만, 해당 명령어 기반으로 시스템 관리는 사람에게는 매우 자연스럽지만, 시스템 입장에서는 매우 어렵고 불편한 영역이다.

학습 수준에 따라서 다르겠지만, 대다수 LLM 시스템은 직접적인 명령어 처리를 종종 엉뚱하게 하는 경우가 있다.

또한, 어떠한 사용자가 시스템에 명령어 작업을 LLM 통해서 진행 하였는지 확인이 어렵기 때문에 IaC기반으로 서비스 관리를 권장한다.

# 컨테이너+폐쇠형 LLM

## LLM 구성

사용하기 위해서 프로그램은 컨테이너로 패키징 한다.

```
# podman build -t localhost/llm/kiki-agent-server \
-f Containers/Containerfile.agent
# buildah build -t localhost/llm/kiki-agent-server \
-f Containers/Containerfile.agent
# podman images
```

# 컨테이너+폐쇠형 LLM

## LLM 실행

문제가 없으면 CPU기반으로 LLM 동작 및 실행이 가능하다.

```
# podman run --rm -it -p 8082:8082 \
-e MODEL_URL=http://host.containers.internal:8080/v1 \
-e API_KEY=sk-noauth -e WORK_DIR=/work \
--name kiki-agent-server localhost/llm/kiki-agent:v1
```

쉽게 실행하기 위해서 다음과 같이 실행이 가능하다.

```
# podman kube play Containers/pod-llama-ansible.yaml
# podman kube down Containers/pod-llama-ansible.yaml
```

# 컨테이너+폐쇠형 LLM

## LLM 실행

올바르게 연결 및 구성이 되었는지 curl 명령어로 확인한다.

```
# curl -s http://127.0.0.1:8080/v1/chat/completions \
-H "Content-Type: application/json" \
-d '{"model":"local-llama","messages":[{"role":"user","content":"say
ok"}]}'
```

# LLM 활용

## 앤서블 플레이북 생성 및 실행

플레이북 생성 및 실행은 다음과 같다.

```
## 인벤토리 파일 전달이 필요 시
# python3 kiki.py --base-url http://127.0.0.1:8082 \
--message "Nginx 설치" --inventory @./hosts.ini

## 인벤토리 파일 전달이 필요 없을 때
# python3 kiki.py --base-url http://127.0.0.1:8082 \
--message "Nginx 설치" --inventory hosts.ini
```

# LLM 활용

## 오픈스택 자원

플레이북 생성 및 실행은 다음과 같다.

```
# ./kiki.py gen -t openstack --name demo-server \
--image rocky-9-generic --flavor m1.small --network ext-net \
--out heat/demo-stack.yaml
```

자연어 처리는 다음과 같다.

```
# ./kiki.py --base-url http://127.0.0.1:8082 \
--model local-llama --message "ext-net에 rocky-9 이미지를 사용하는
m1.small 서버 2대를 만드는 Heat 템플릿 만들어줘. YAML만 출력." \
--target openstack --name web-stack
```

# LLM 활용

## 쿠버네티스 자원

플레이북 생성 및 실행은 다음과 같다.

```
# ./kiki.py gen -t k8s --name web --image nginx:1.27 \
--port 80 --replicas 3 --ns demo --out k8s/web.yaml \
--validate
```

자연어는 다음처럼 처리가 가능하다.

```
# kiki.py --base-url http://127.0.0.1:8082 \
--model local-llama --message "demo 네임스페이스에 nginx 디플로이먼트 3개와 80 포트 서비스 만들어줘. YAML만 출력." \
--target k8s \  --name web-k8s
```

# LLM 기반으로 VIDC 자원 관리하기

## LLM WITH IaC

강사의 가이드에 따라서 LLM 통해서 플레이북 생성하여 **포드만/쿠버네티스/오픈스택 자원**을 관리한다.

# 수고하셨습니다.