Volume Hot-plugging into Kubernetes Container

박은수
삼성전자

발표자 소개

박은수 **(esevan.park),** SW Engineer
Samsung Research Big Data Team

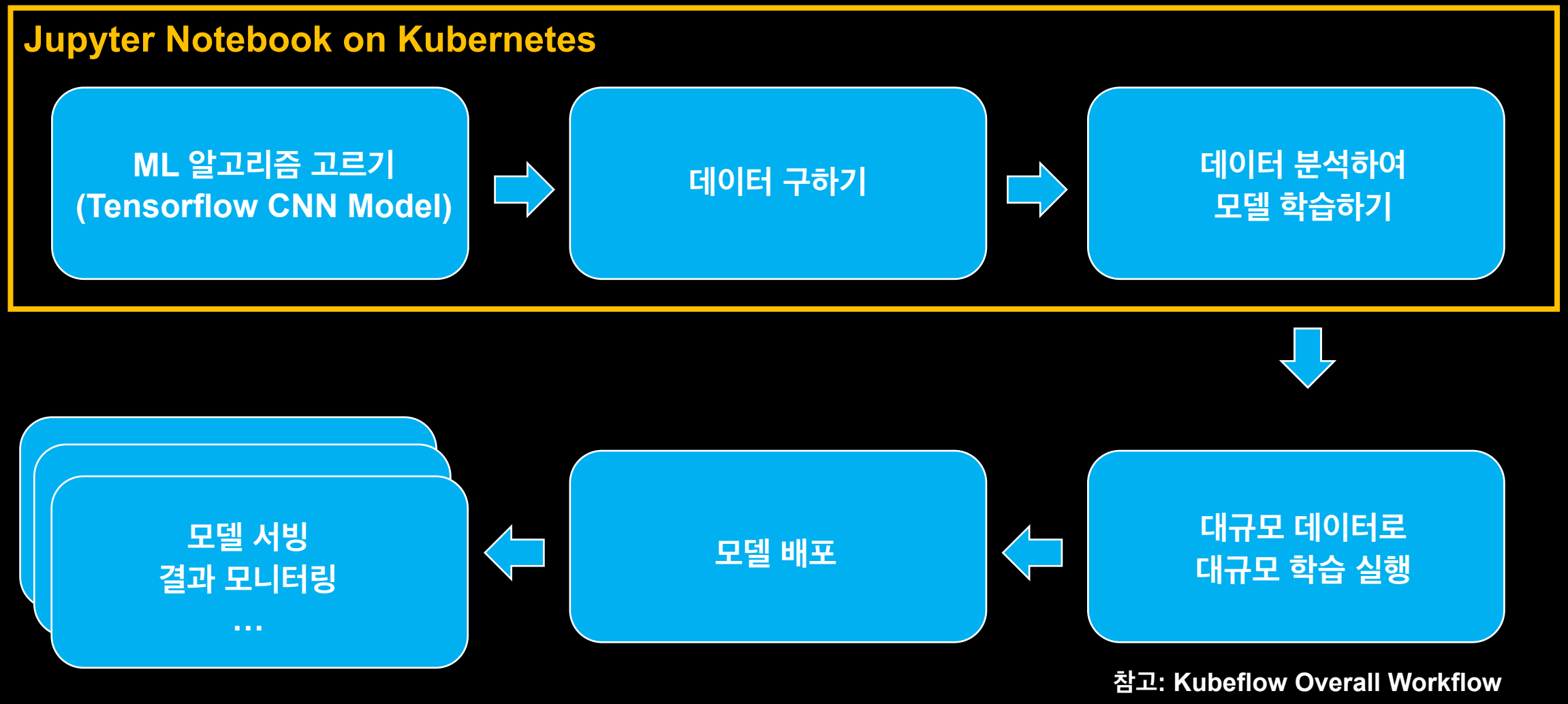
Openstack
Kuryr-kubernetes / Ironic / Nova / Neutron

Kubernetes
CNI / Cloud Provider Openstack

Jupyter
Jupyter Lab / Enterprise Gateway

...

AI Dev Workflow



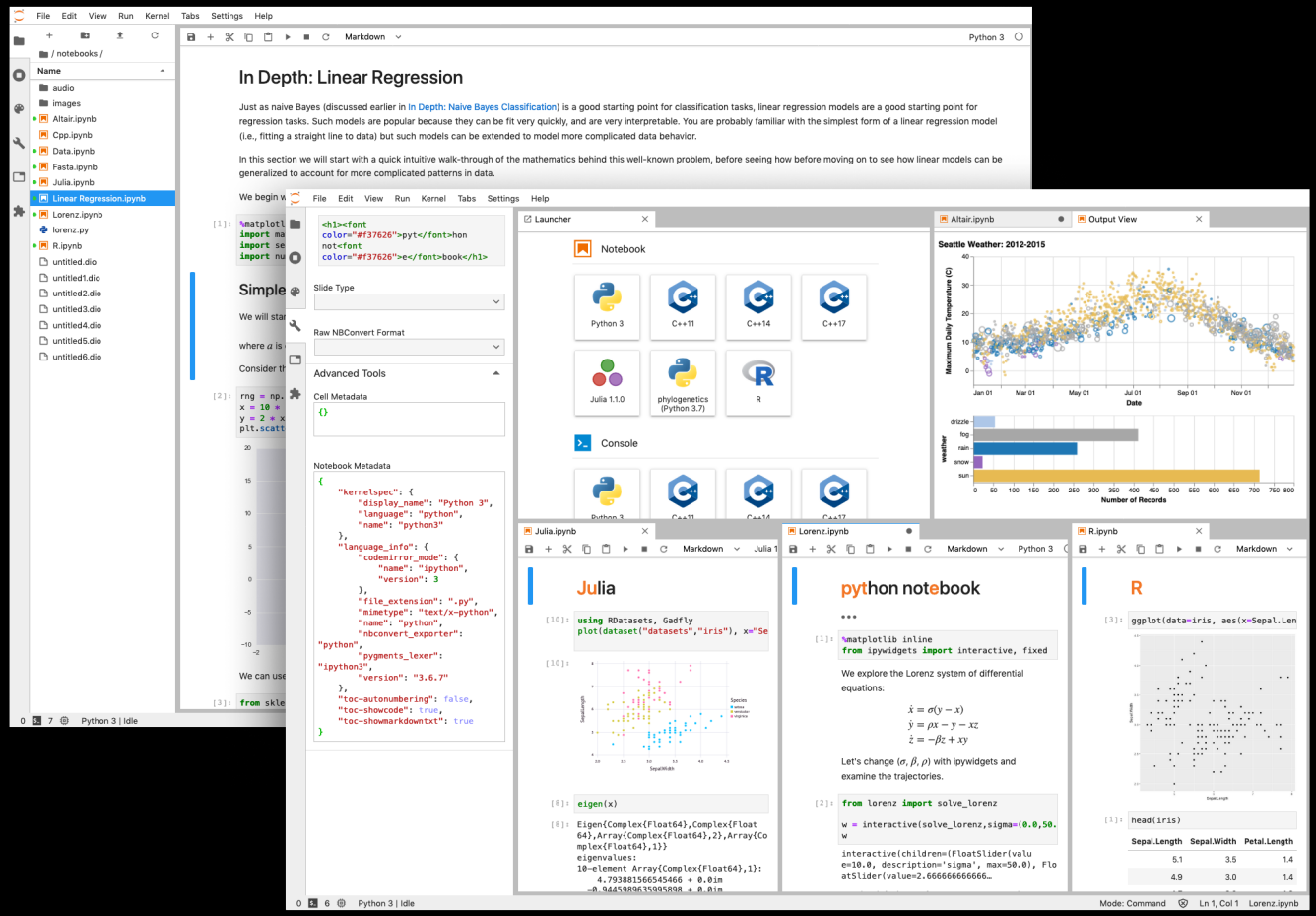
참고: Kubeflow Overall Workflow

What’s Jupyter?

PC 기반의 Web Application

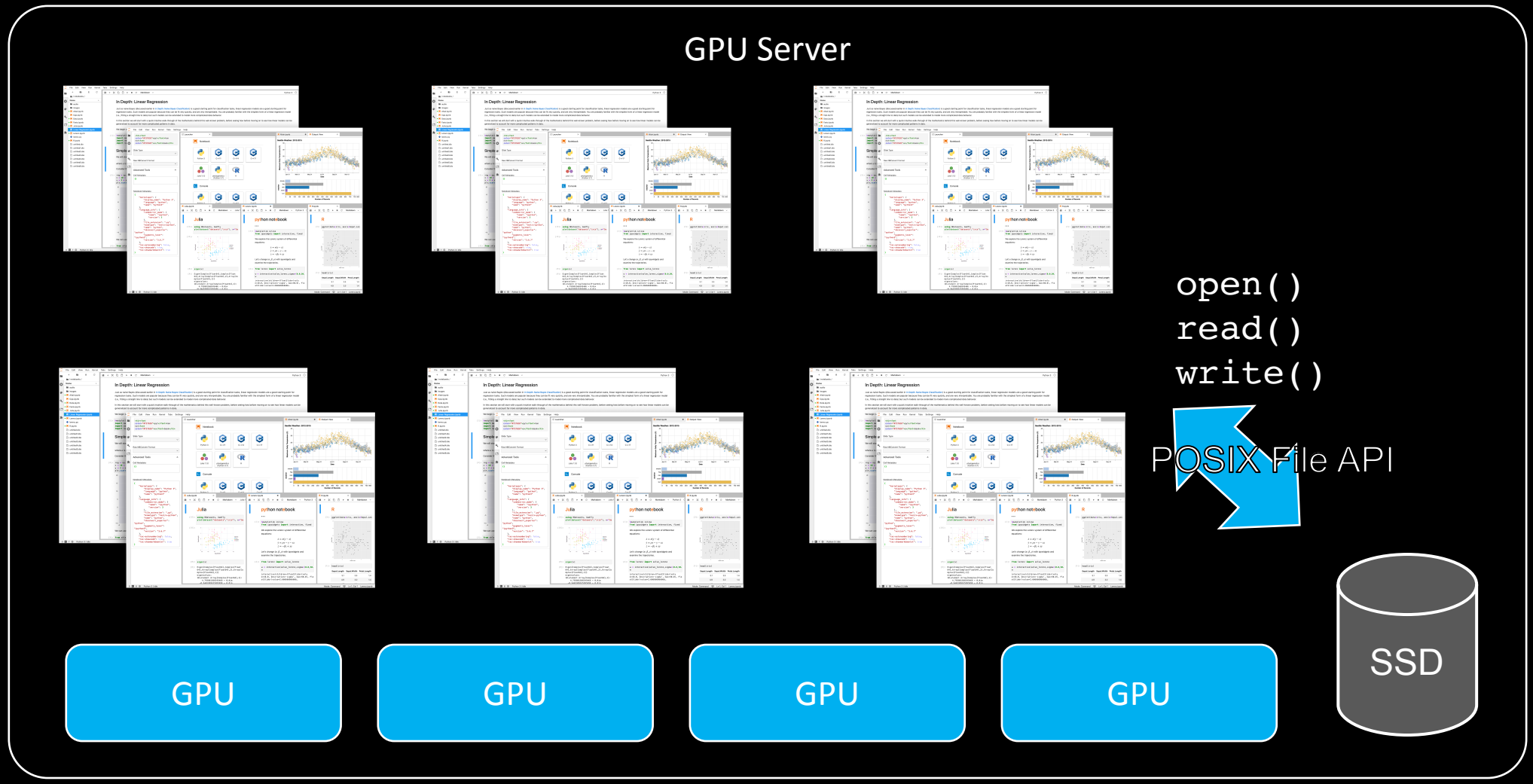
실시간 Interactive 개발 환경

다양한 시각화 툴 및 위젯 제공



출처: https://jupyter.org

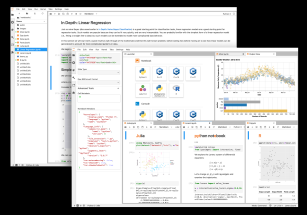
기존 사용자 환경



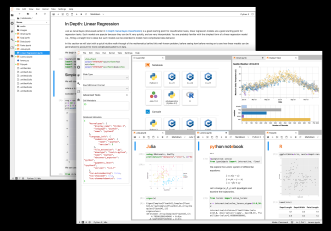
Kubernetes 기반의 사용자 환경

GPU Server #1

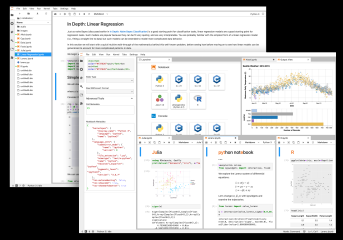
Container



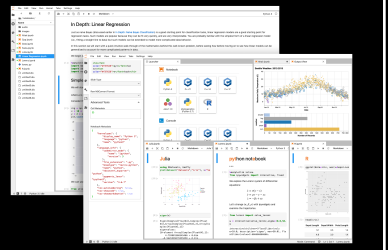
Container



Container



Container



nvidia-docker

GPU

GPU

GPU

GPU

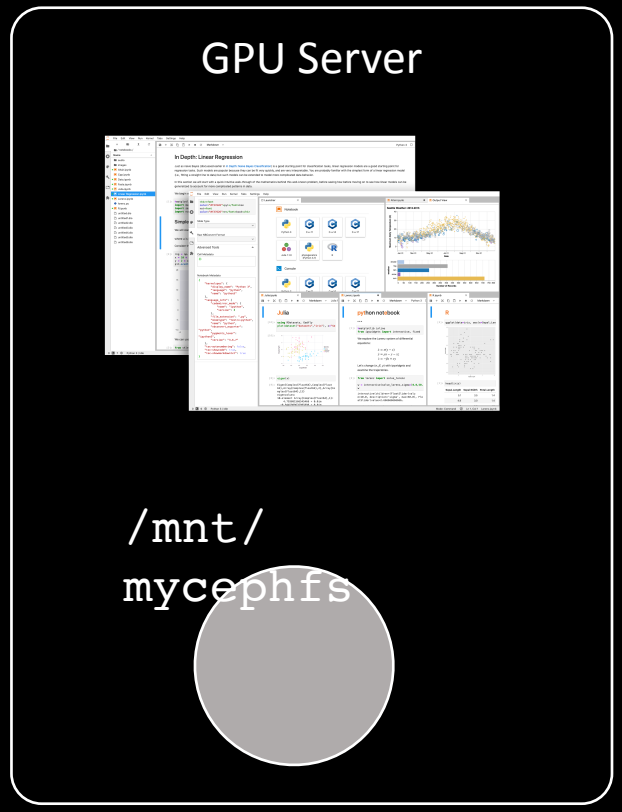
GPU Server #2

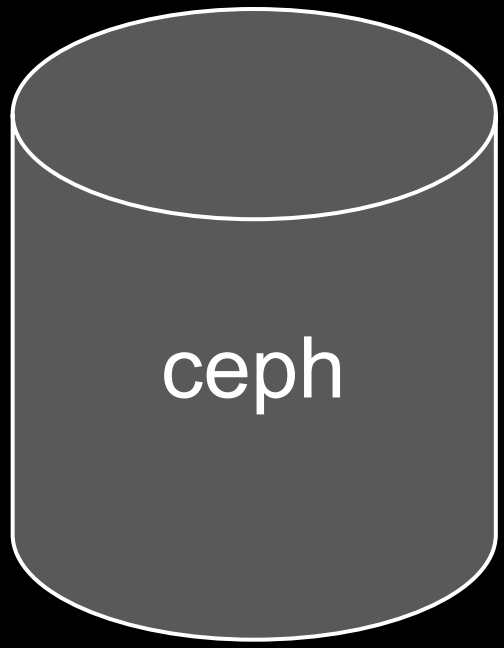
Container

Container

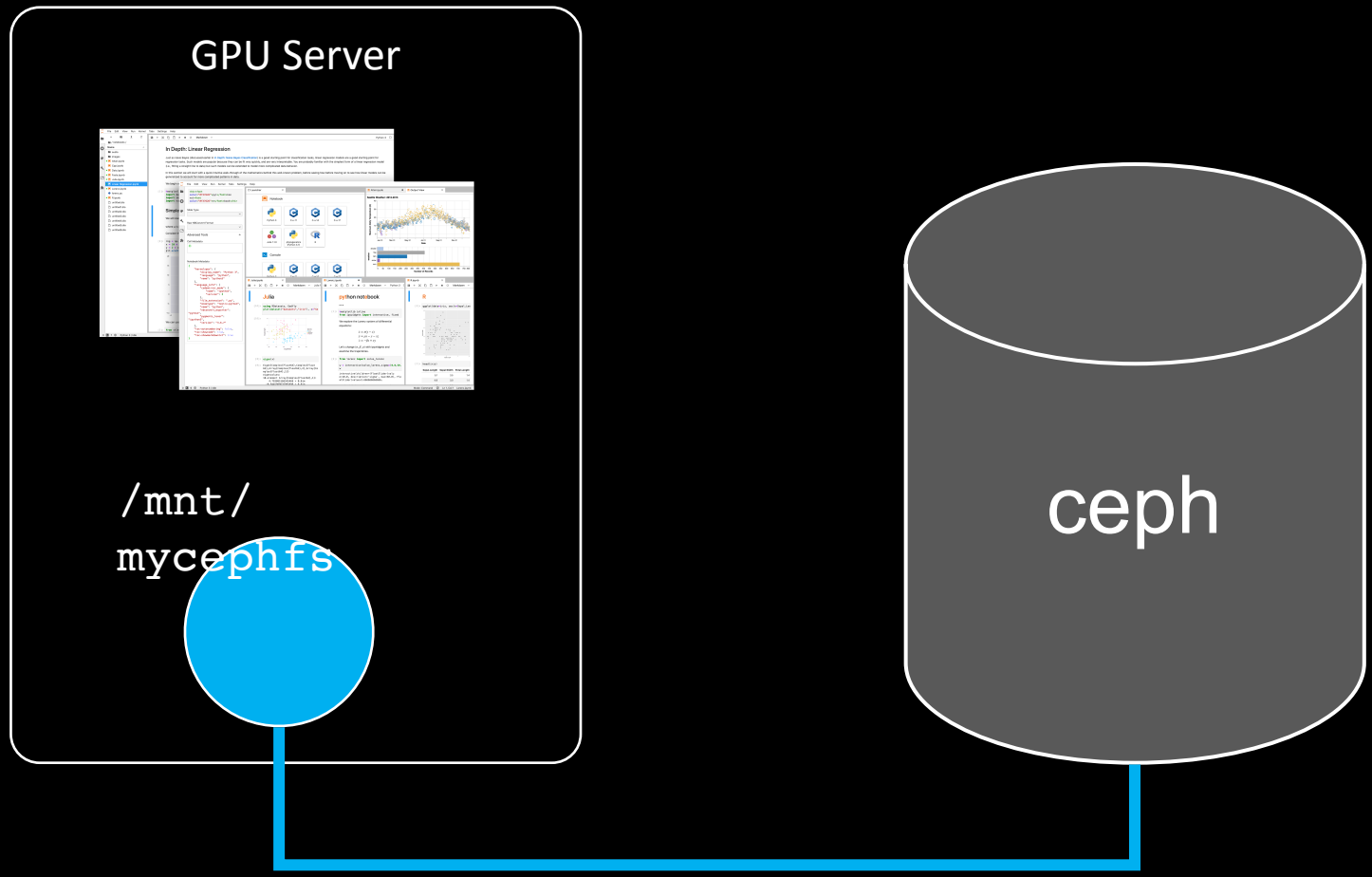
ceph

기존 환경에서의 데이터 연결



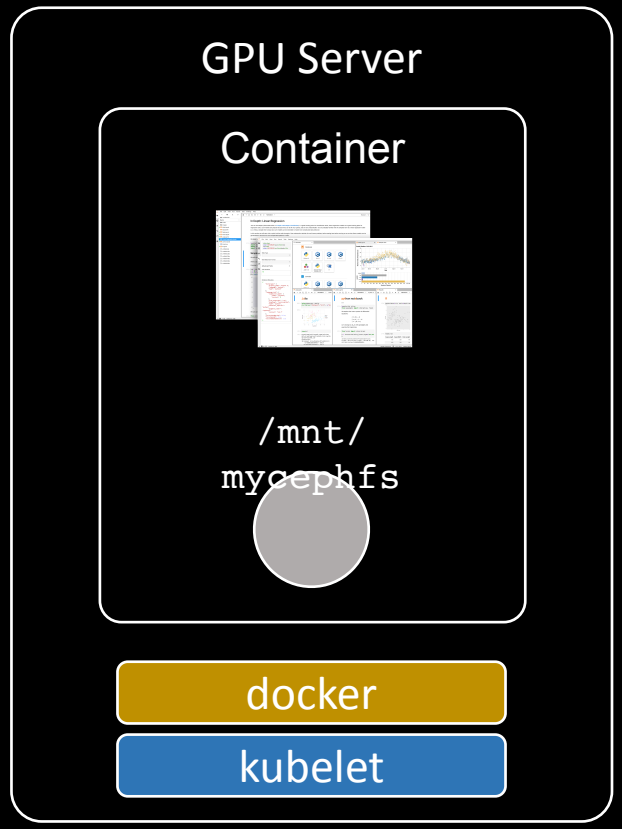


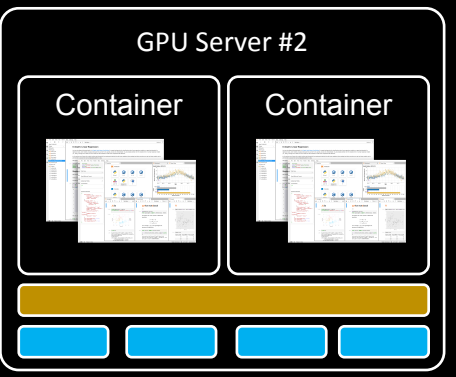
기존 환경에서의 데이터 연결

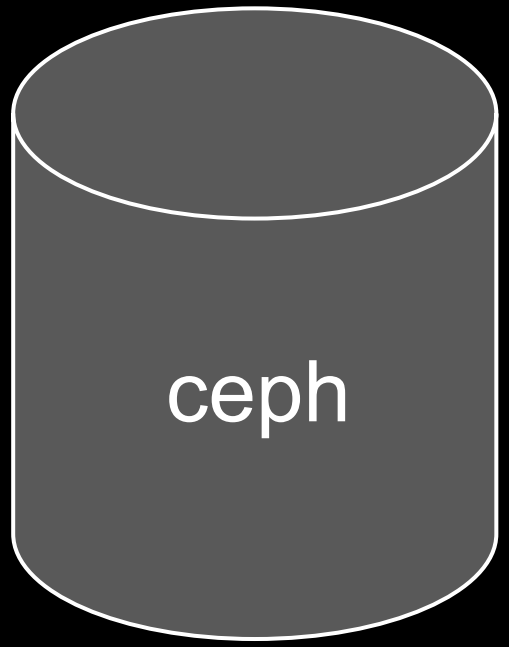



```
# mount.ceph 192.168.0.1,192.168.0.2,192.168.0.3:/ /mnt/mycephfs
```

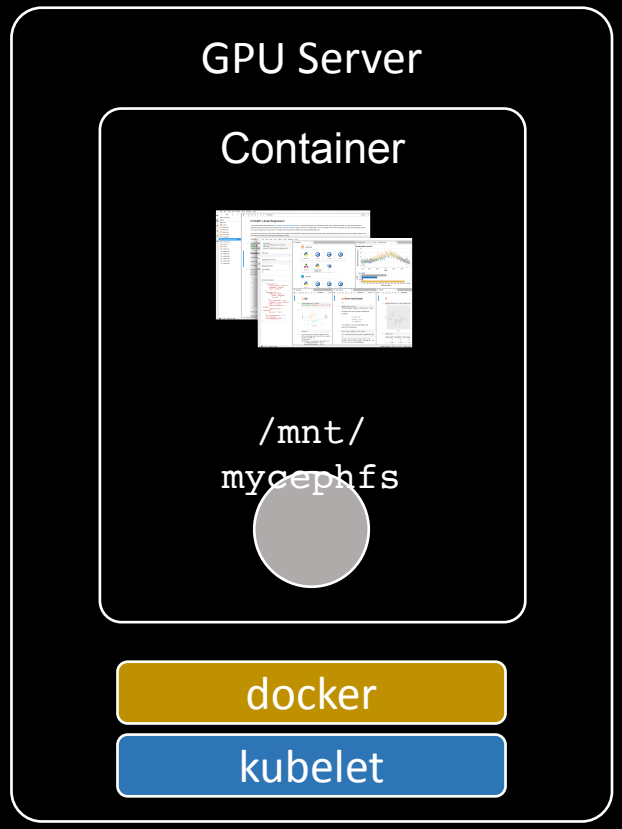
Kubernetes 기반의 사용자 환경

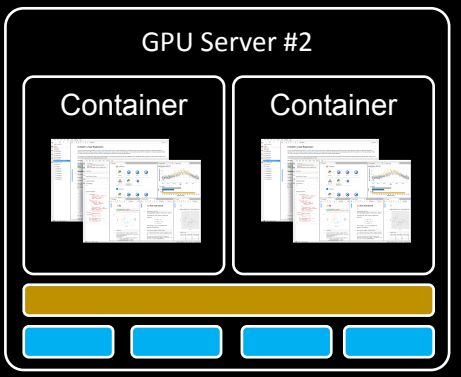


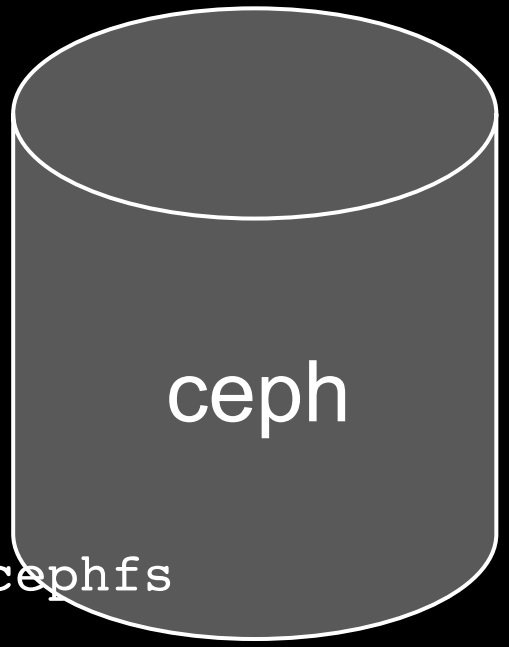




Kubernetes 기반의 사용자 환경

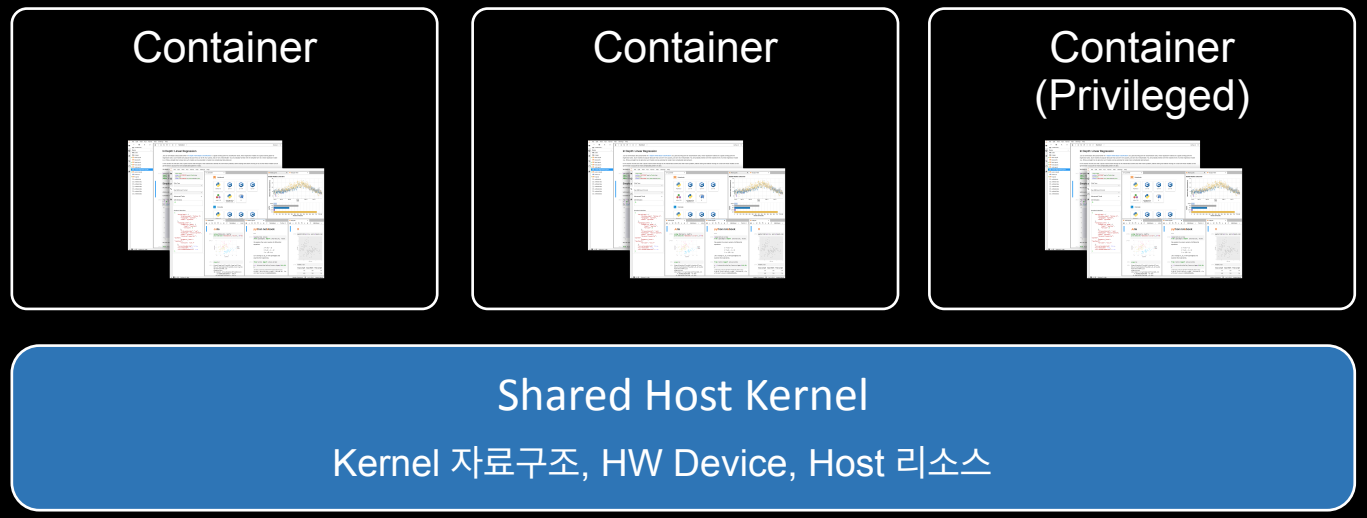




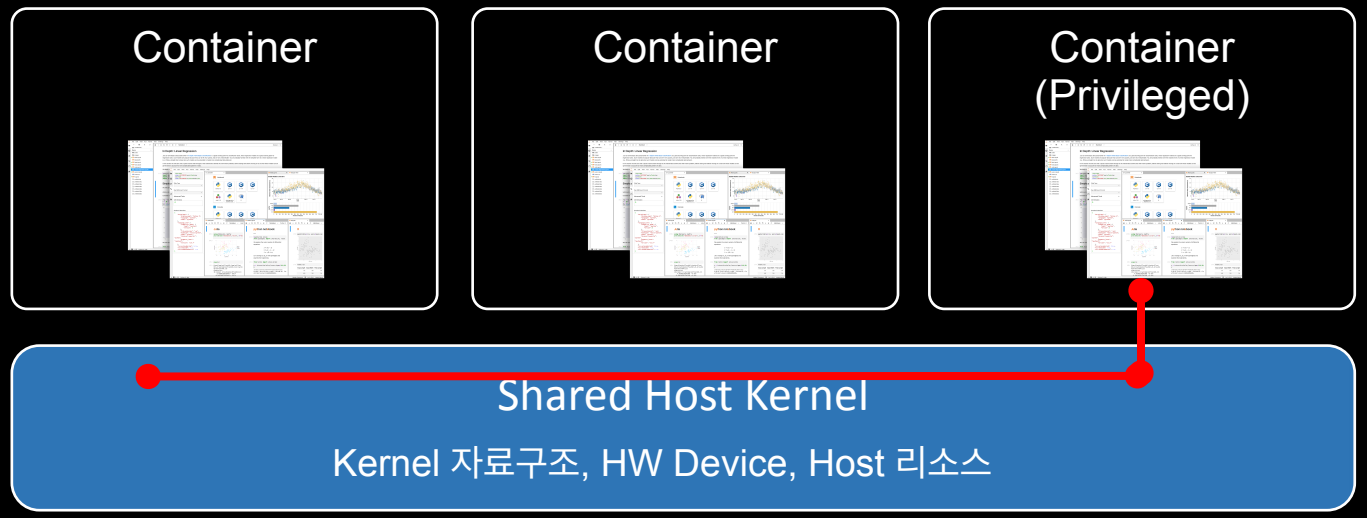



```
$ mount.ceph 192.168.0.1,192.168.0.2,192.168.0.3:/ /mnt/mycephfs
mount error 13 = Permission denied
```

왜 Permission 문제가 발생 했는가?



왜 Permission 문제가 발생 했는가?




```
esevan@DESKTOP-LP3M500:~/test-py$ docker run --rm -it httpd bash
root@7808803cde78:/usr/local/apache2# ls /dev
console  core  fd  full  mqueue  null  ptmx  pts  random  shm  stderr  stdin  st
root@7808803cde78:/usr/local/apache2# exit
esevan@DESKTOP-LP3M500:~/test-py$ docker run --rm -it --privileged httpd bash
root@8559893dc4eb:/usr/local/apache2# ls /dev
autofs           loop3            ptmx   ram7    stdin   tty2   tty33  tty47
bsg              loop4            pts    ram8    stdout  tty20  tty34  tty48
btrfs-control    loop5            ram0   ram9    tty     tty21  tty35  tty49
console          loop6            ram1   random  tty0    tty22  tty36  tty5
core             loop7            ram10  rtc0    tty1    tty23  tty37  tty50
cpu_dma_latency  mapper           ram11  sda     tty10   tty24  tty38  tty51
cuse             mem              ram12  sdb     tty11   tty25  tty39  tty52
fd               memory_bandwidth ram13  sdc     tty12   tty26  tty4   tty53
full             mqueue           ram14  sdd     tty13   tty27  tty40  tty54
fuse             net              ram15  sg0     tty14   tty28  tty41  tty55
kmsg             network_latency  ram2   sg1     tty15   tty29  tty42  tty56
loop-control     network_throughput ram3 sg2     tty16   tty3   tty43  tty57
loop0            null             ram4   sg3     tty17   tty30  tty44  tty58
loop1            nvram            ram5   shm     tty18   tty31  tty45  tty59
loop2            ppp              ram6   stderr  tty19   tty32  tty46  tty6
```

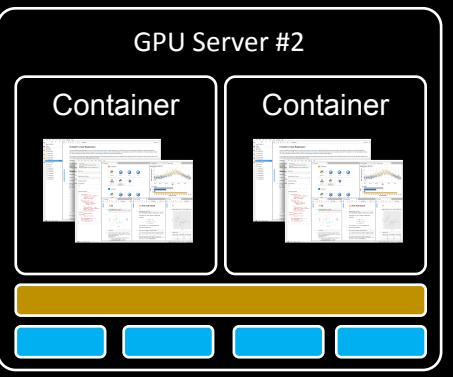
모든 Host 장치가 노출

Kubernetes, by default

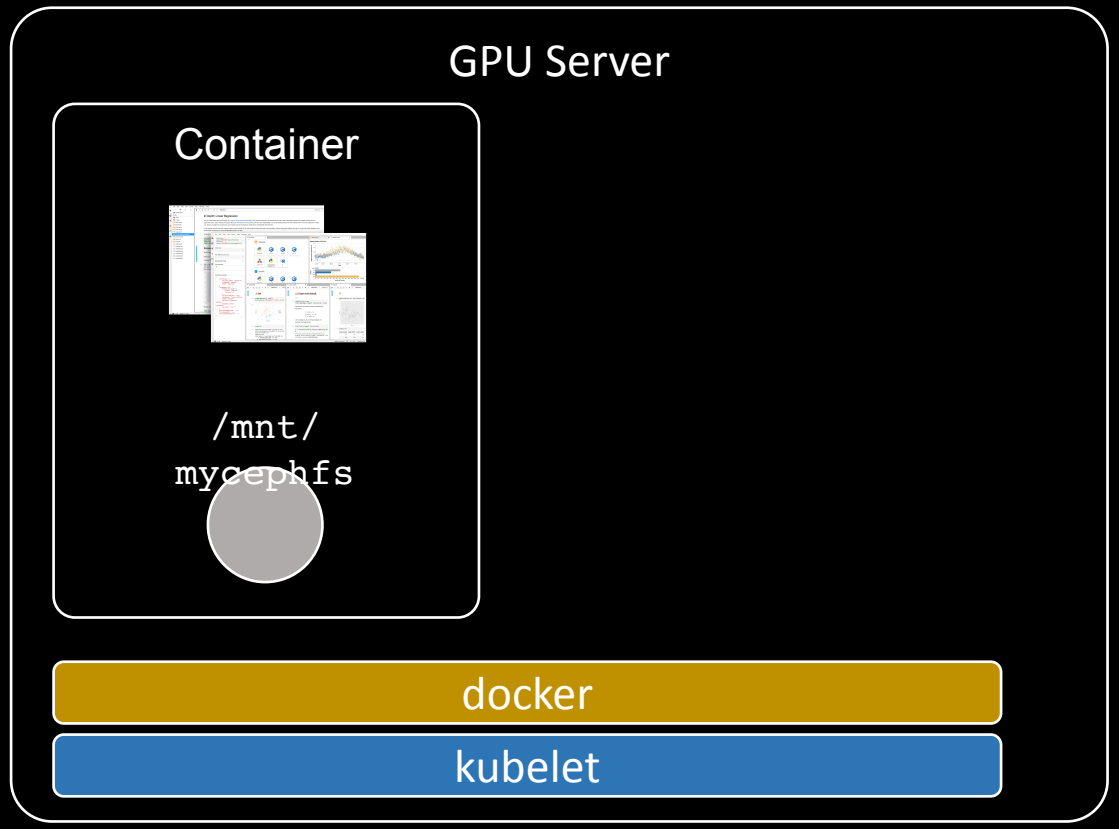
- Root 사용자 제한
- Linux Capability 제한

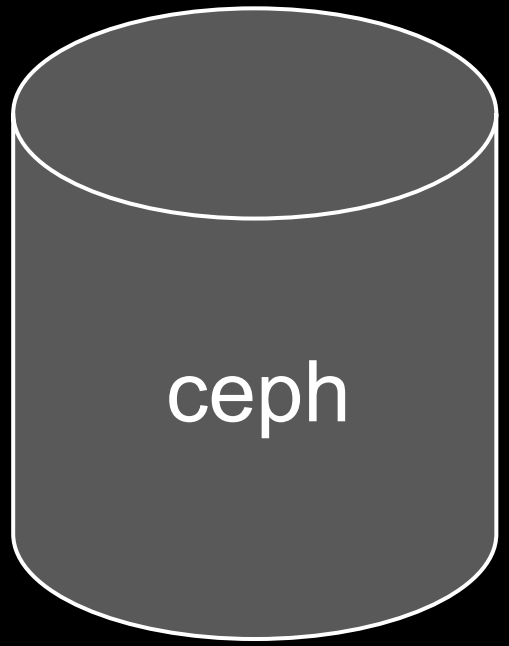
> mount(2) – Linux manual page
>
> Appropriate privilege (Linux: the **CAP_SYS_ADMIN** capability) is required to mount filesystems.

Kubernetes 기반의 사용자 환경



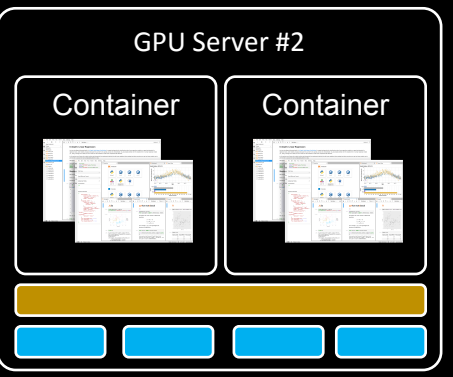

```
spec:
   containers:
   - name: notebook
     image: notebook:latest
     volumeMounts:
     - mountPath: /mnt/mycephfs
       name: ceph-volume
   volumes
   - name: ceph-volume
     cephfs:
       monitors:
       ...
```



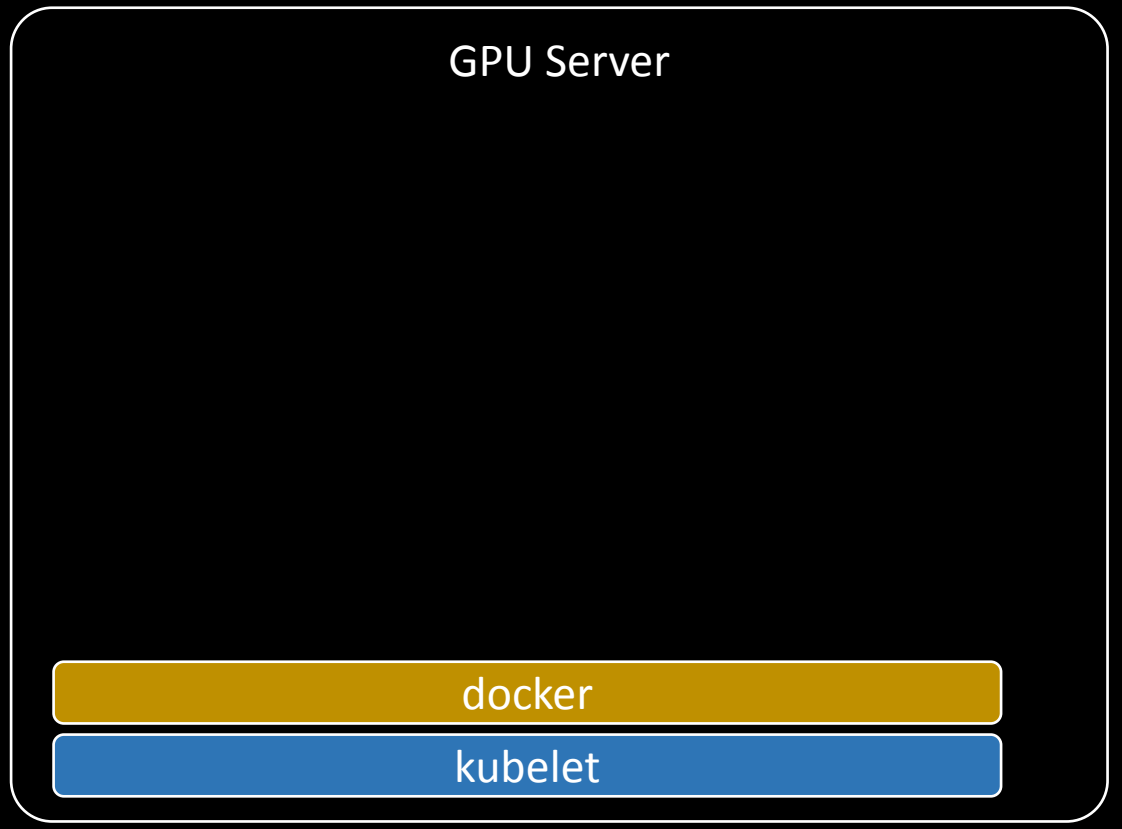


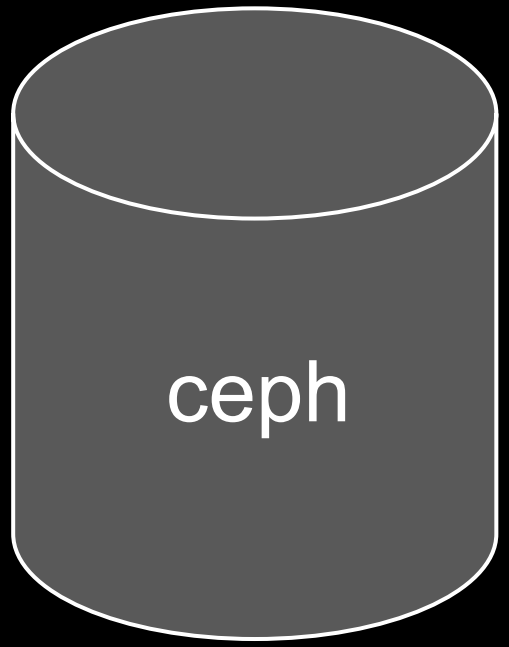

```
$ kubectl edit deploy notebook-user-a
```

Kubernetes 기반의 사용자 환경




```
spec:
   containers:
   - name: notebook
     image: notebook:latest
     volumeMounts:
     - mountPath: /mnt/mycephfs
       name: ceph-volume
   volumes
   - name: ceph-volume
     cephfs:
       monitors:
       …
```

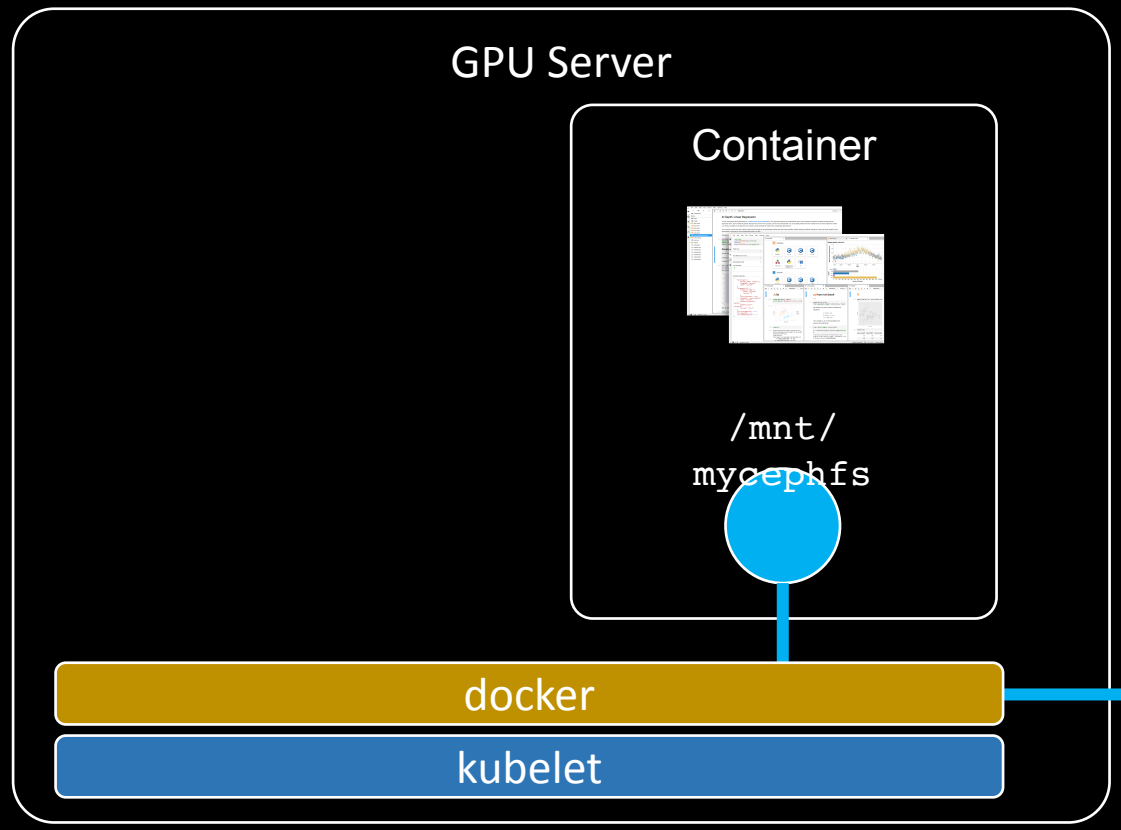


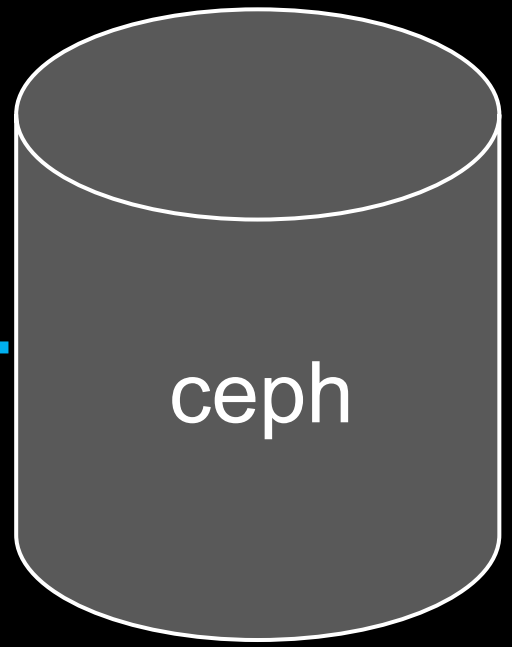


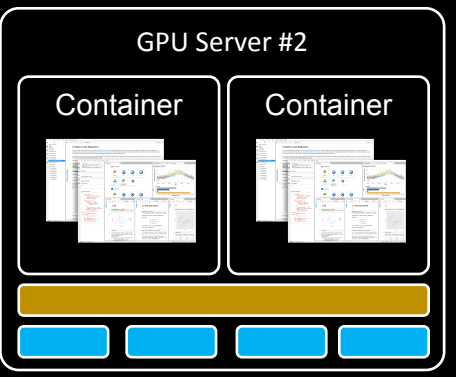

```
$ kubectl edit deploy notebook-user-a
```

Kubernetes 기반의 사용자 환경

```
spec:
  containers:
  - name: notebook
    image: notebook:latest
    volumeMounts:
    - mountPath: /mnt/mycephfs
      name: ceph-volume
  volumes
  - name: ceph-volume
    cephfs:
      monitors:
      ...
```








```
$ kubectl edit deploy notebook-user-a
```

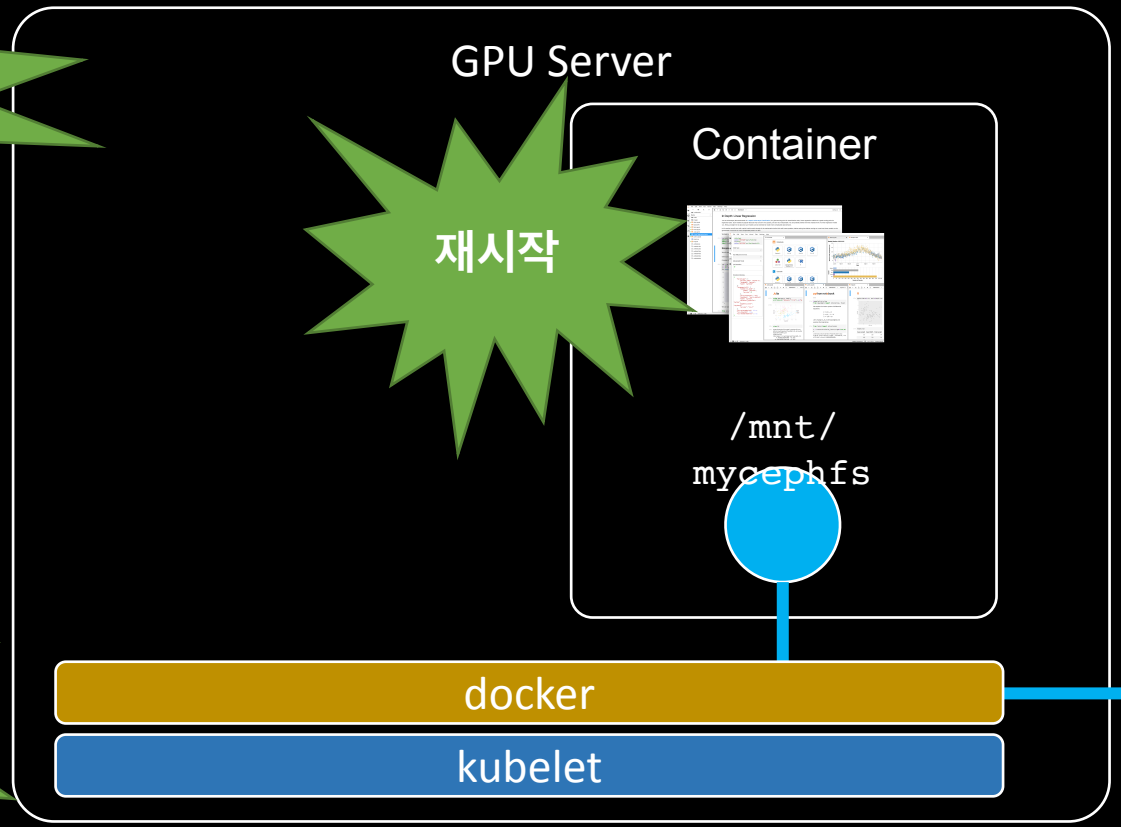
Kubernetes 기반의 사용자 환경

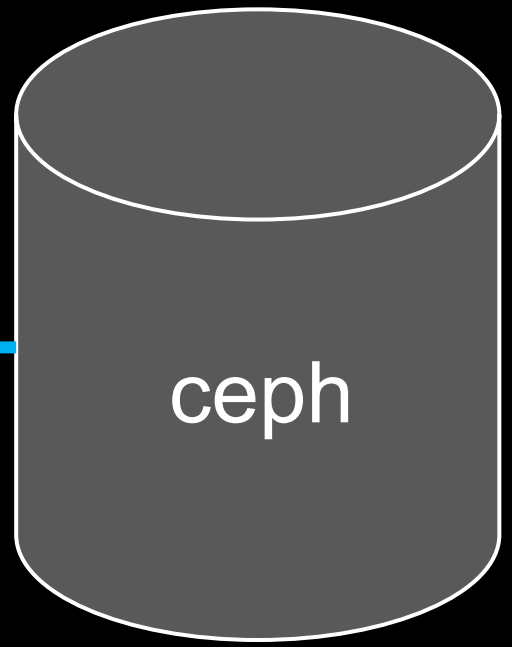
ceph 클러스터 정보 노출

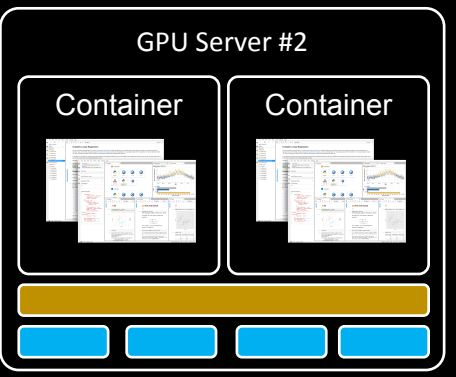
```
spec:
  containers:
  - name: notebook
    image: notebook:latest
    volumeMounts:
    - mountPath: /mnt/mycephfs
      name: ceph-volume
  volumes
    name: ceph-vol
    cephfs:
```

사용자는 kubectl을 직접 사용하지 않음

```
$ kubectl edit deploy notebook-user-a
```







Sidecar Pattern

Design patterns for container-based distributed systems

Brendan Burns *David Oppenheimer*
Google

1 Introduction

In the late 1980s and early 1990s, object-oriented programming revolutionized software development, popularizing the approach of building of applications as collections of modular components. Today we are seeing a similar revolution in distributed system development, with the increasing popularity of microservice architectures built from containerized software components. Containers [15] [22] [1] [2] are particularly well-suited as the fundamental "object" in distributed systems by virtue of the walls they erect at the container boundary. As this architectural style matures, we are seeing the emergence of design patterns, much as we did for object-oriented programs, and for the same reason – thinking in terms of objects (or containers) abstracts away the low-level details of code, eventually revealing higher-level patterns that are common to a variety of applications and algorithms.

This paper describes three types of design patterns that we have observed emerging in container-based distributed systems: single-container patterns for container management, single-node patterns of closely cooperating containers, and multi-node patterns for distributed algorithms. Like object-oriented patterns before them, these patterns for distributed computation encode best practices, simplify development, and make the systems where they are used more reliable.

2 Distributed system design patterns

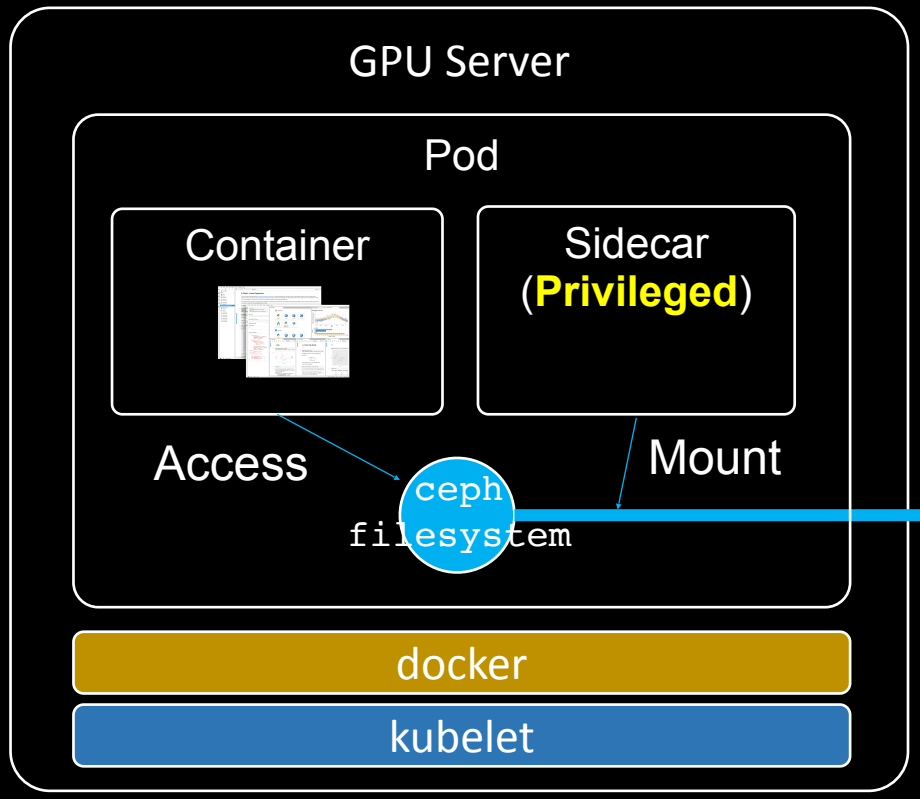
After object-oriented programming had been used for some years, design patterns emerged and were documented [3]. These patterns codified and regularized general approaches to solving particular common programming problems. This codification further improved the general state of the art in programming because it made it easier for less experienced programmers to produce well-engineered code, and led to the development of reusable libraries that made code more reliable and faster to develop.

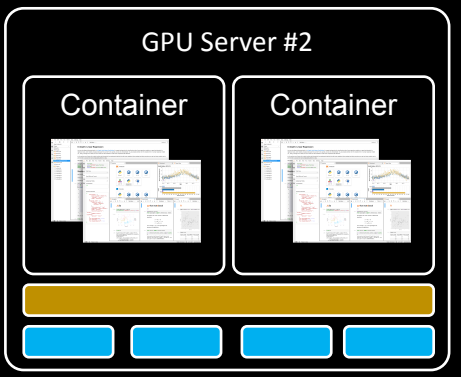
The state-of-the-art in distributed system engineering today looks significantly more like the world of early 1980s programming than it does the world of object-oriented development. Yet it's clear from the success of the MapReduce pattern [4] in bringing the power of "Big Data" programming to a broad set of fields and developers, that putting in place the right set of patterns can dramatically improve the quality, speed, and accessibility of distributed system programming. But even the success of MapReduce is largely limited to a single programming language, insofar as the Apache Hadoop [5] ecosystem is primarily written in and for Java. Developing a truly comprehensive suite of patterns for distributed system design requires a very generic, language-neutral vehicle to represent the atoms of the system.

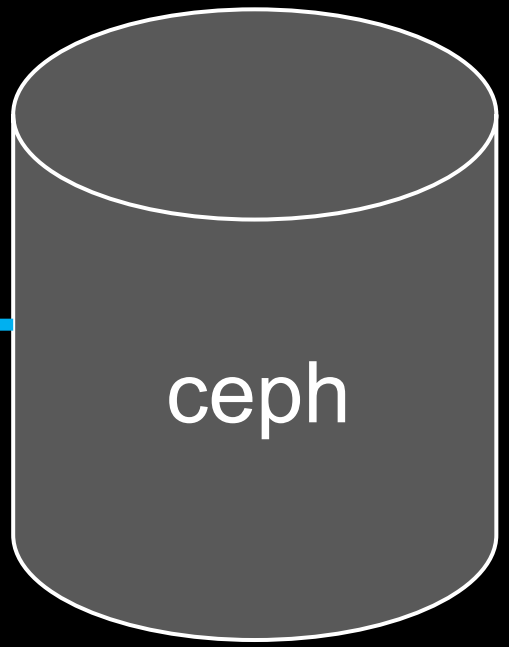
Thus it is fortunate that the last two years have seen a dramatic rise in adoption of Linux container technology. The container and the container image are exactly the abstractions needed for the development of distributed systems patterns. To date, containers and container images have achieved the large measure of their popularity simply by being a better, more reliable method for delivering software from development all the way through production. By being hermetically sealed, carrying their dependencies with them, and providing an atomic deployment signal ("succeeded"/"failed"), they dramatically improve on the previous state of the art in deploying software in the datacenter or cloud. But containers have the potential to be much more than just a better deployment vehicle – we believe they are destined to become analogous to objects in object-oriented software systems, and as such will enable the development of distributed system design patterns. In the following sections we explain why we believe this to be the case, and describe some patterns that we see emerging to regularize and guide the engineering of distributed systems over the coming years.

1

Burns, B., & Oppenheimer, D. (2016).
Design patterns for container-based distributed systems.
(HotCloud 16).

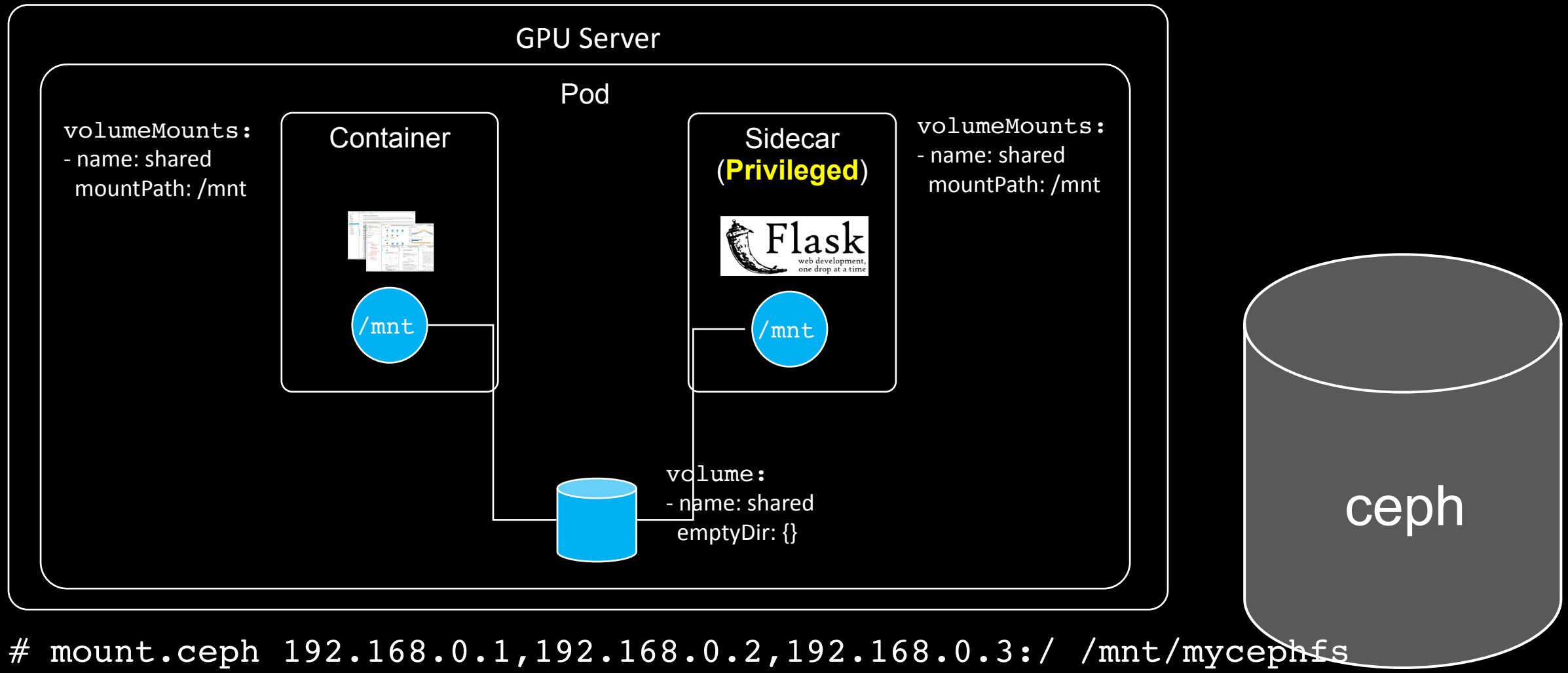






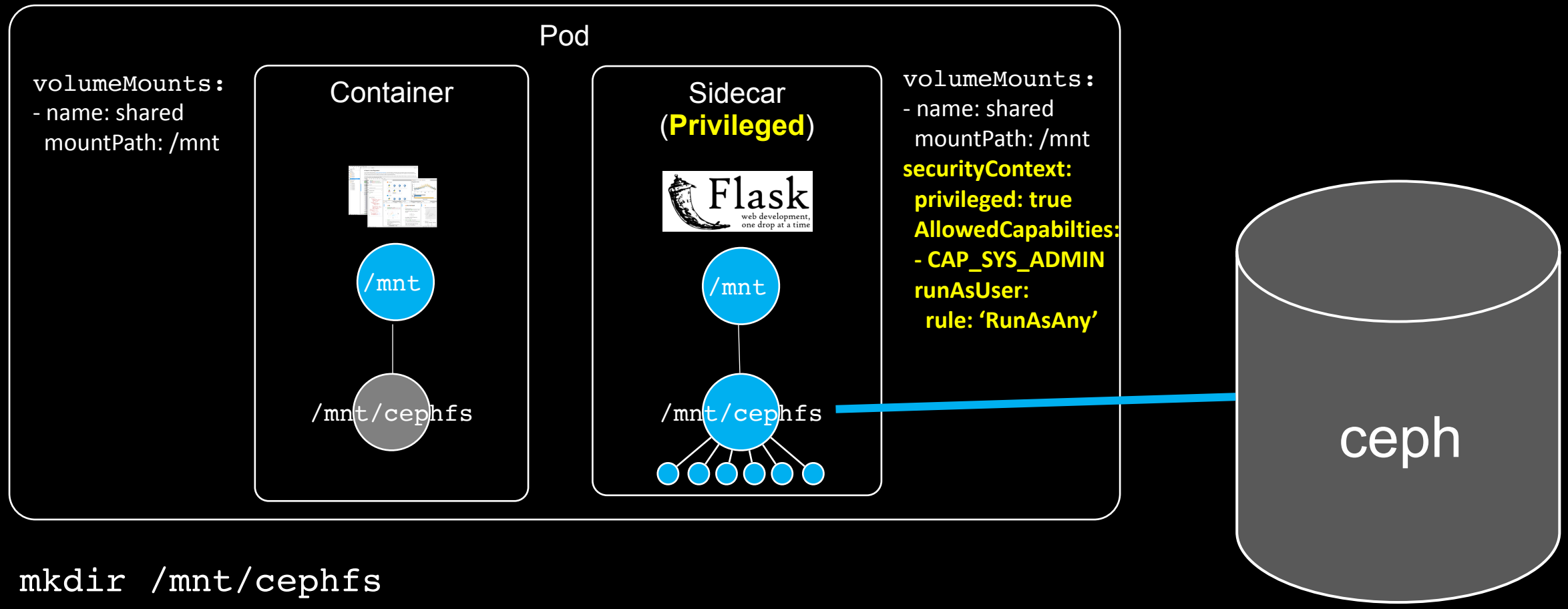

```
# mount.ceph 192.168.0.1,192.168.0.2,192.168.0.3:/ /mnt/mycephfs
```

Sidecar Pattern



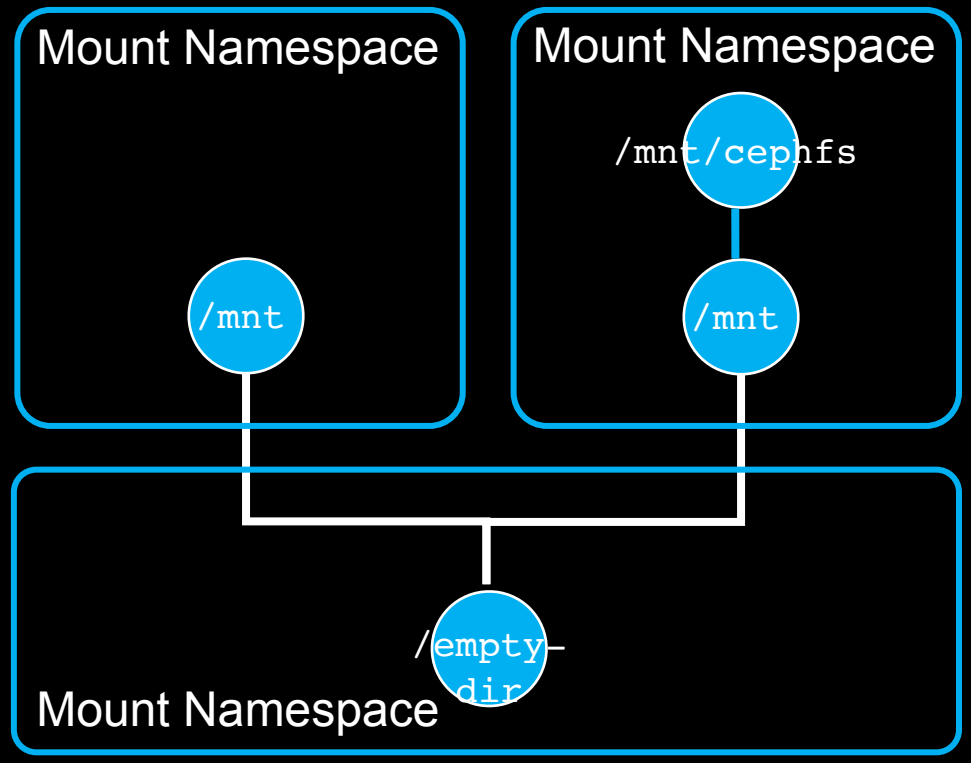

```
# mount.ceph 192.168.0.1,192.168.0.2,192.168.0.3:/ /mnt/mycephfs
mount error 13 = Permission denied
```

Sidecar Security Context 설정




```
# mkdir /mnt/cephfs
# mount.ceph 192.168.0.1,192.168.0.2,192.168.0.3:/ /mnt/mycephfs
```

MountPropagation 설정



Mount Propagation: None // *Private* in Linux Kernel
Isolated Namespace

- **다른 Mount Namespace에 영향을 주지 않음**

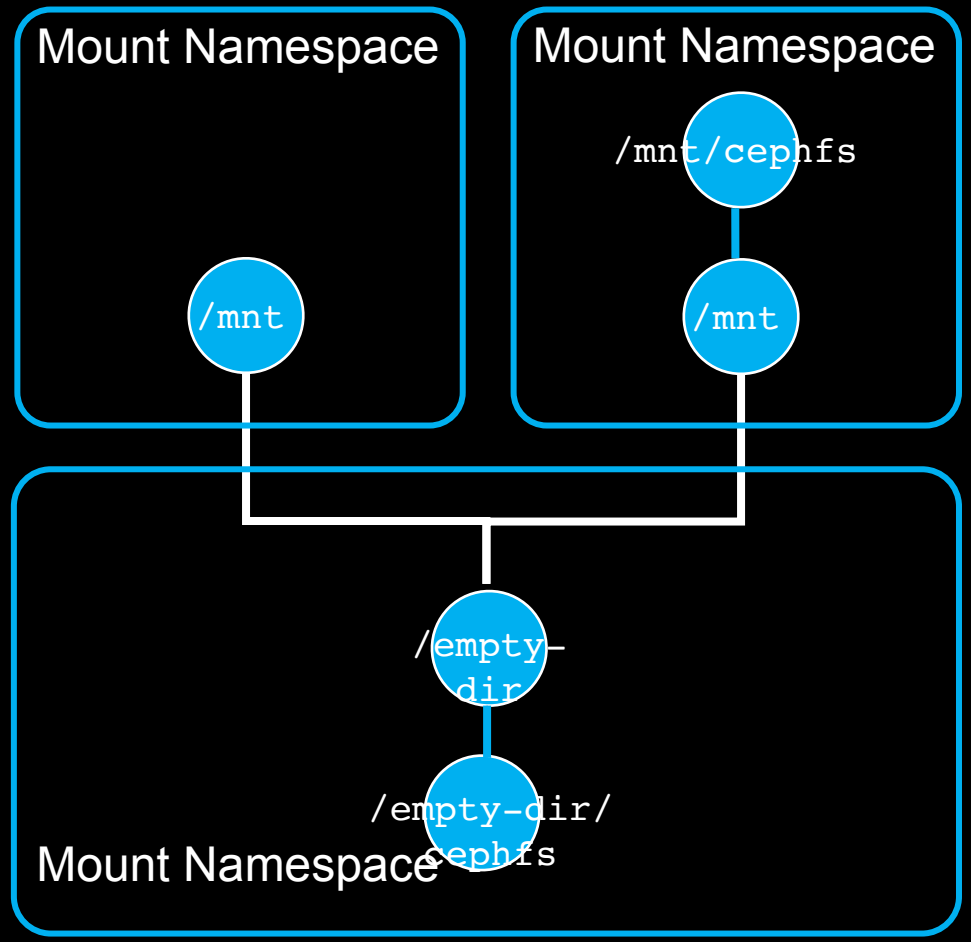
Mount Propagation: BiDirectional // *rshared* in Linux Kernel
연결된 Namespace

- Host의 Namespace와 동기화

Mount Propagation: HostToContainer // *rslave* in Linux Kernel
단방향 연결된 Namespace

- Host의 Namespace 변경 → Container Namespace 변경
- Container Namespace 변경 → Host Namespace는 그대로

MountPropagation 설정



Mount Propagation: None // *Private* in Linux Kernel
Isolated Namespace

- 다른 Mount Namespace에 영향을 주지 않음

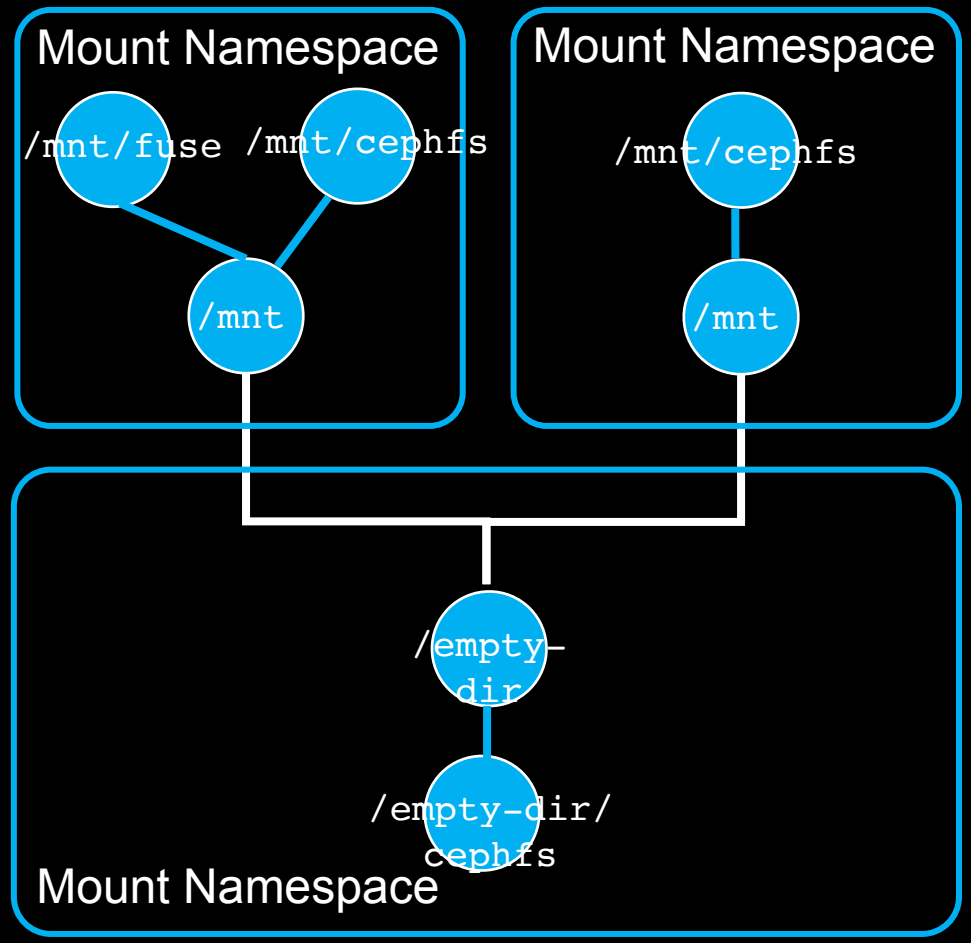
Mount Propagation: BiDirectional // *rshared* in Linux Kernel
연결된 Namespace

- **Host의 Namespace와 동기화**

Mount Propagation: HostToContainer // *rslave* in Linux Kernel
단방향 연결된 Namespace

- Host의 Namespace 변경 → Container Namespace 변경
- Container Namespace 변경 → Host Namespace는 그대로

MountPropagation 설정



Mount Propagation: None // *Private* in Linux Kernel
Isolated Namespace

- 다른 Mount Namespace에 영향을 주지 않음

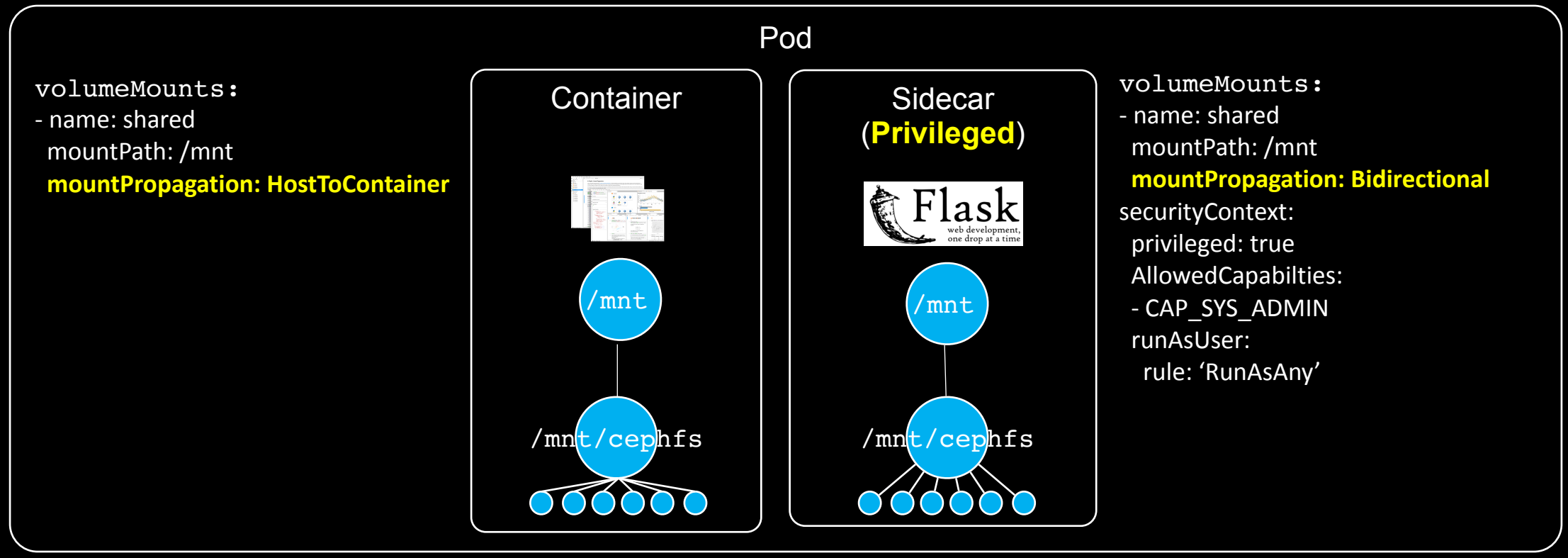
Mount Propagation: BiDirectional // *rshared* in Linux Kernel
연결된 Namespace

- Host의 Namespace와 동기화

Mount Propagation: HostToContainer // *rslave* in Linux Kernel
단방향 연결된 Namespace

- **Host의 Namespace 변경 → Container Namespace 변경**
- **Container Namespace 변경 → Host Namespace는 그대로**

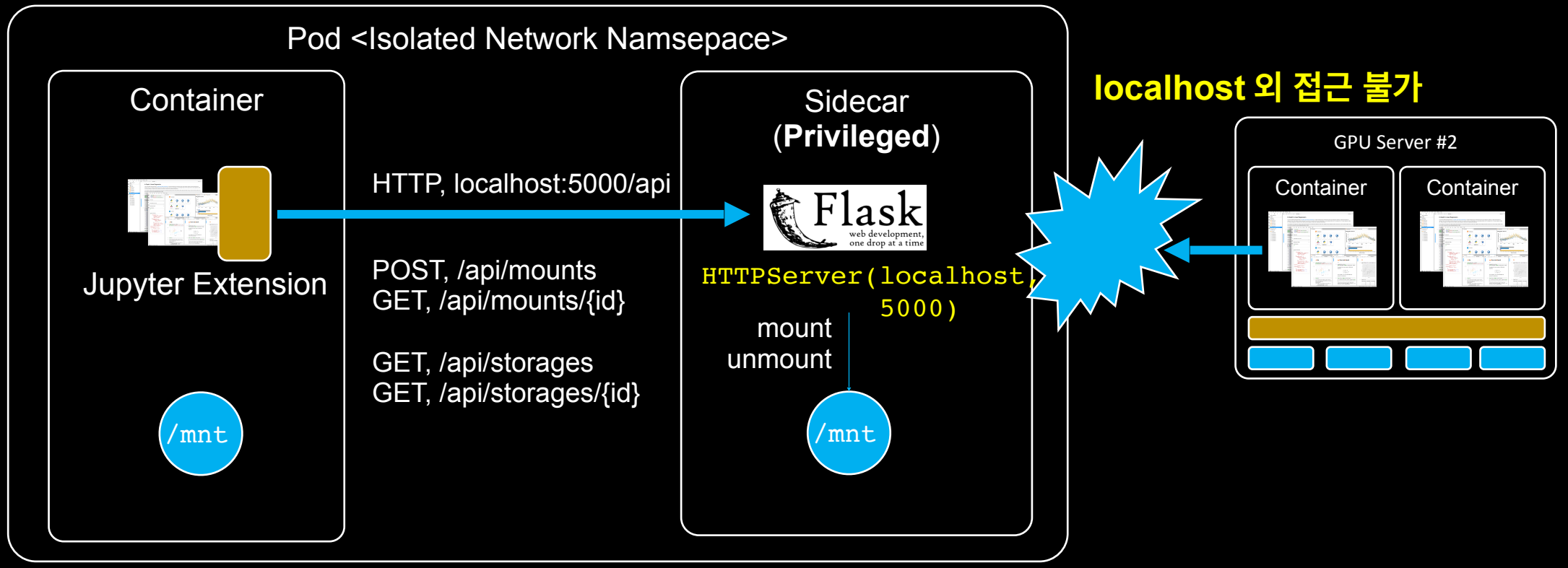
MountPropagation 설정




```
# mkdir /mnt/cephfs
# mount.ceph 192.168.0.1,192.168.0.2,192.168.0.3:/ /mnt/mycephfs
```

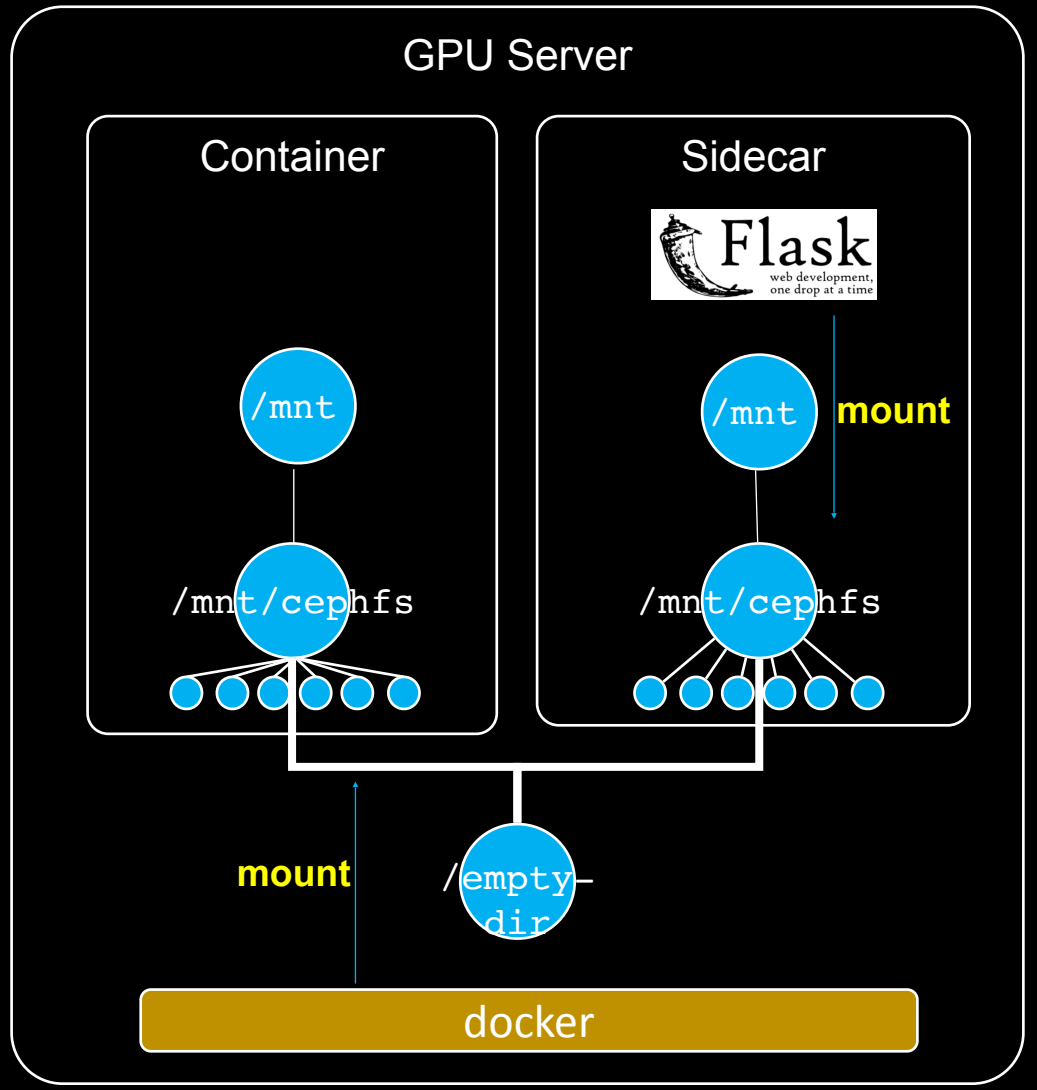
Restful Mount Interface

- 사용자에게는 어떻게 Mount 인터페이스를 제공할까?
- 다른 사용자의 Mount 인터페이스엔 접근하면 안되는데...



UnGraceful Termination

- Docker는 Pod 내 Container 삭제 시도
- 도커가 모르는 자원을 누군가 사용중

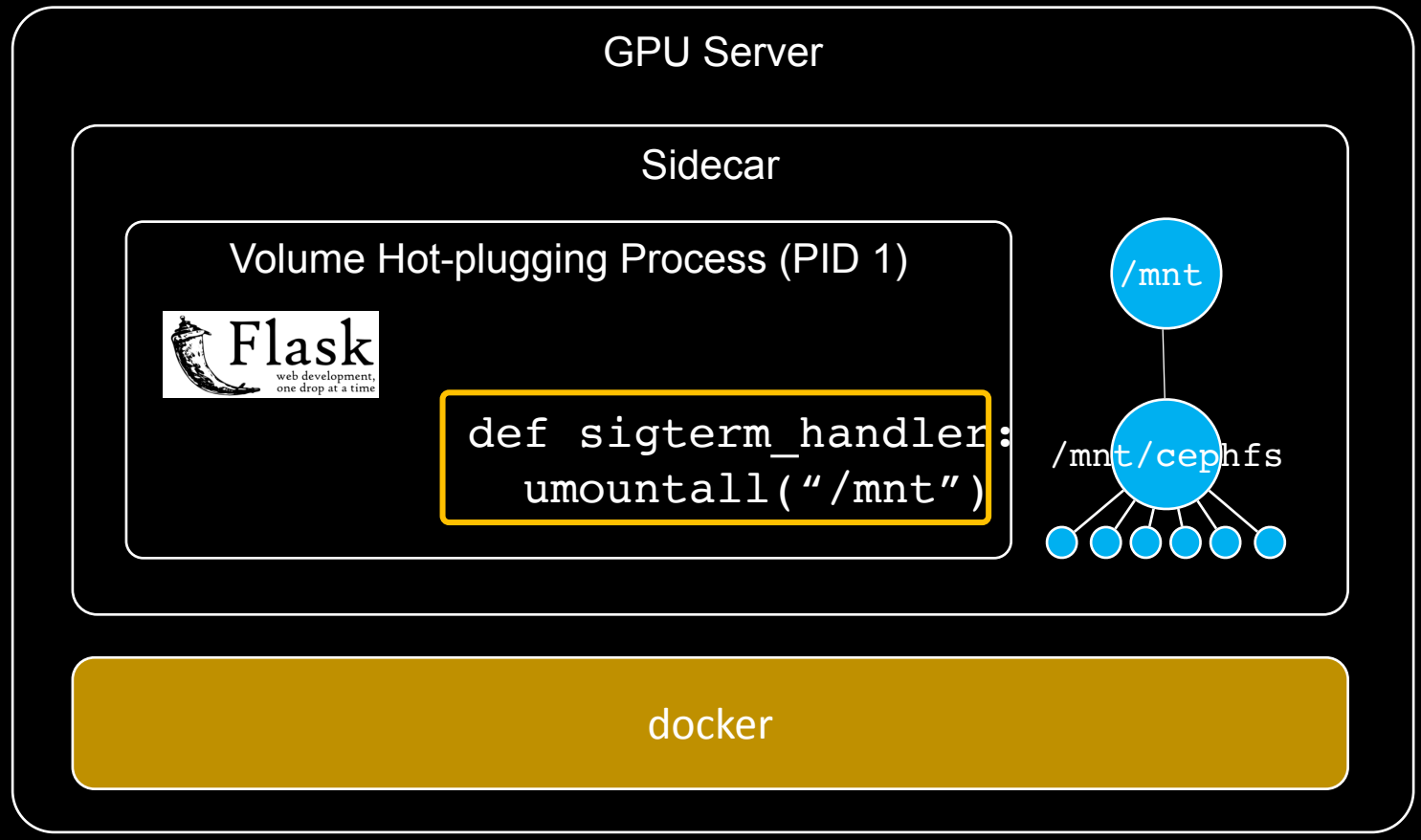


Graceful Termination

1. **VHP 프로세스 실행 시 SIGTERM Handler 등록**

2. **Pod 삭제 시 Container 내 PID 1 프로세스에 SIGTERM 시그널 발생**

2. **시그널 트랩하여 /mnt 내 모든 디렉토리 Unmount**

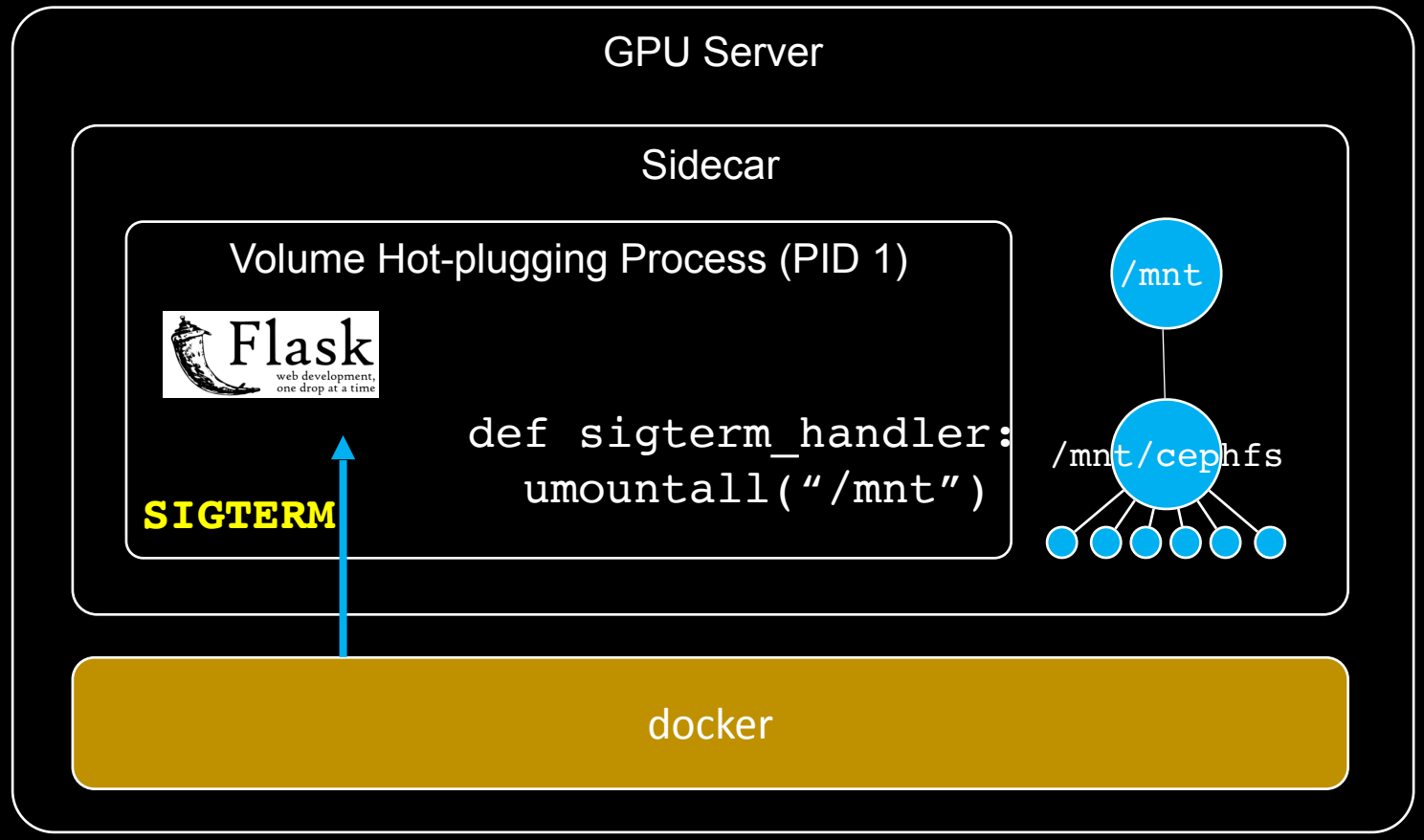


Graceful Termination

1. VHP 프로세스 실행 시 SIGTERM Handler 등록

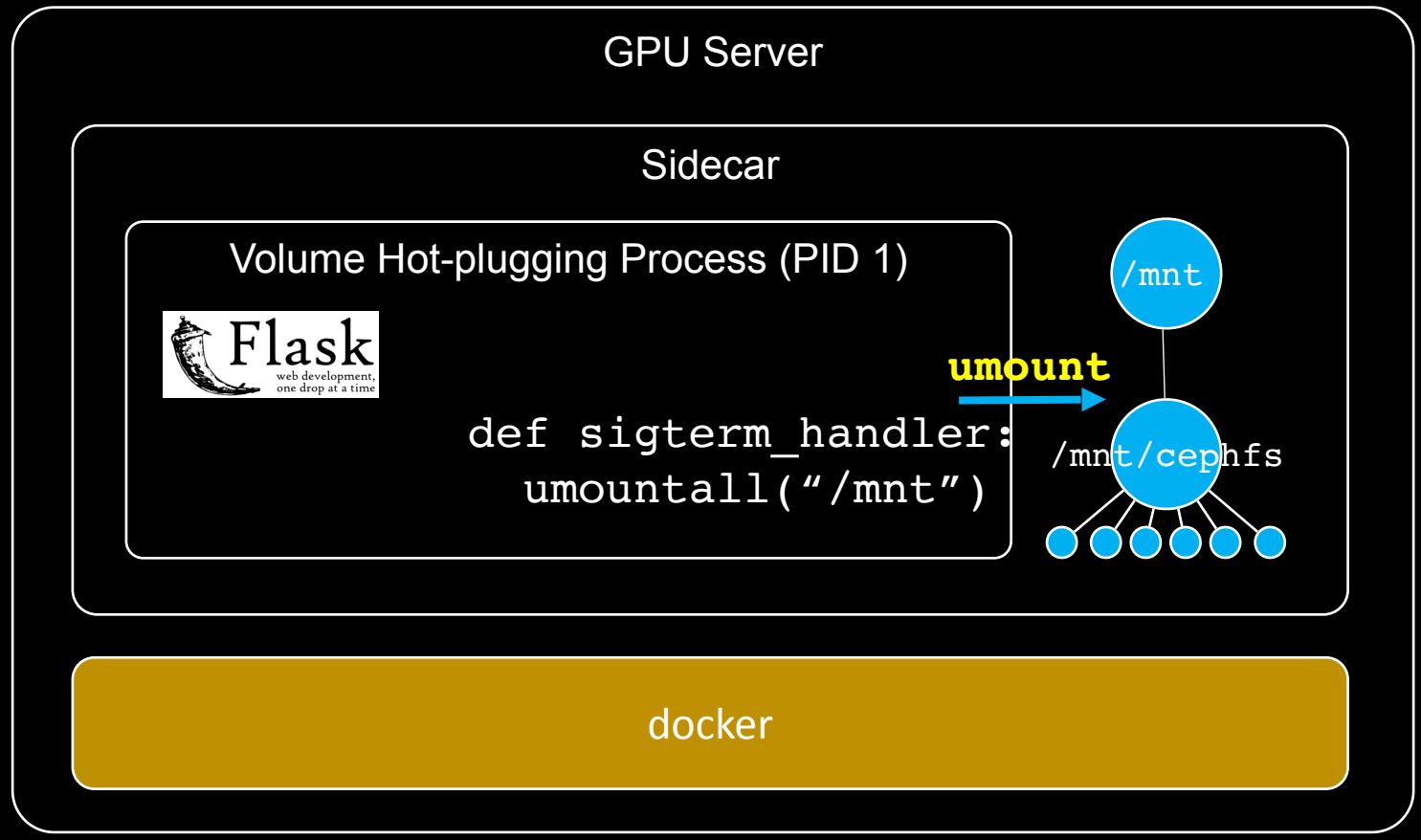
2. Pod 삭제 시 Container 내 PID 1 프로세스에 SIGTERM 시그널 발생

3. 시그널 트랩하여 /mnt 내 모든 디렉토리 Unmount



Graceful Termination

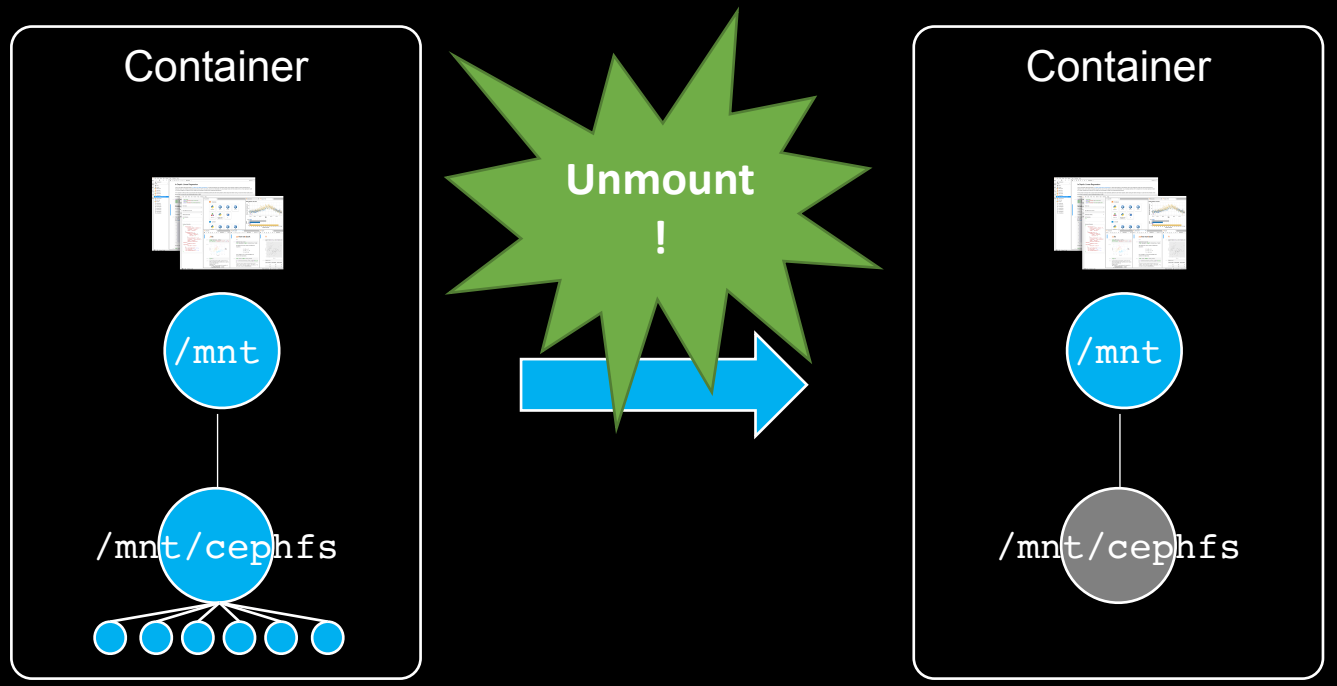
1. VHP 프로세스 실행 시 SIGTERM Handler 등록
2. Pod 삭제 시 Container 내 PID 1 프로세스에 SIGTERM 시그널 발생
3. 시그널 트랩하여 /mnt 내 모든 디렉토리 Unmount



Intermittent Unmount

한 번에 잘 될 리가 없잖아…

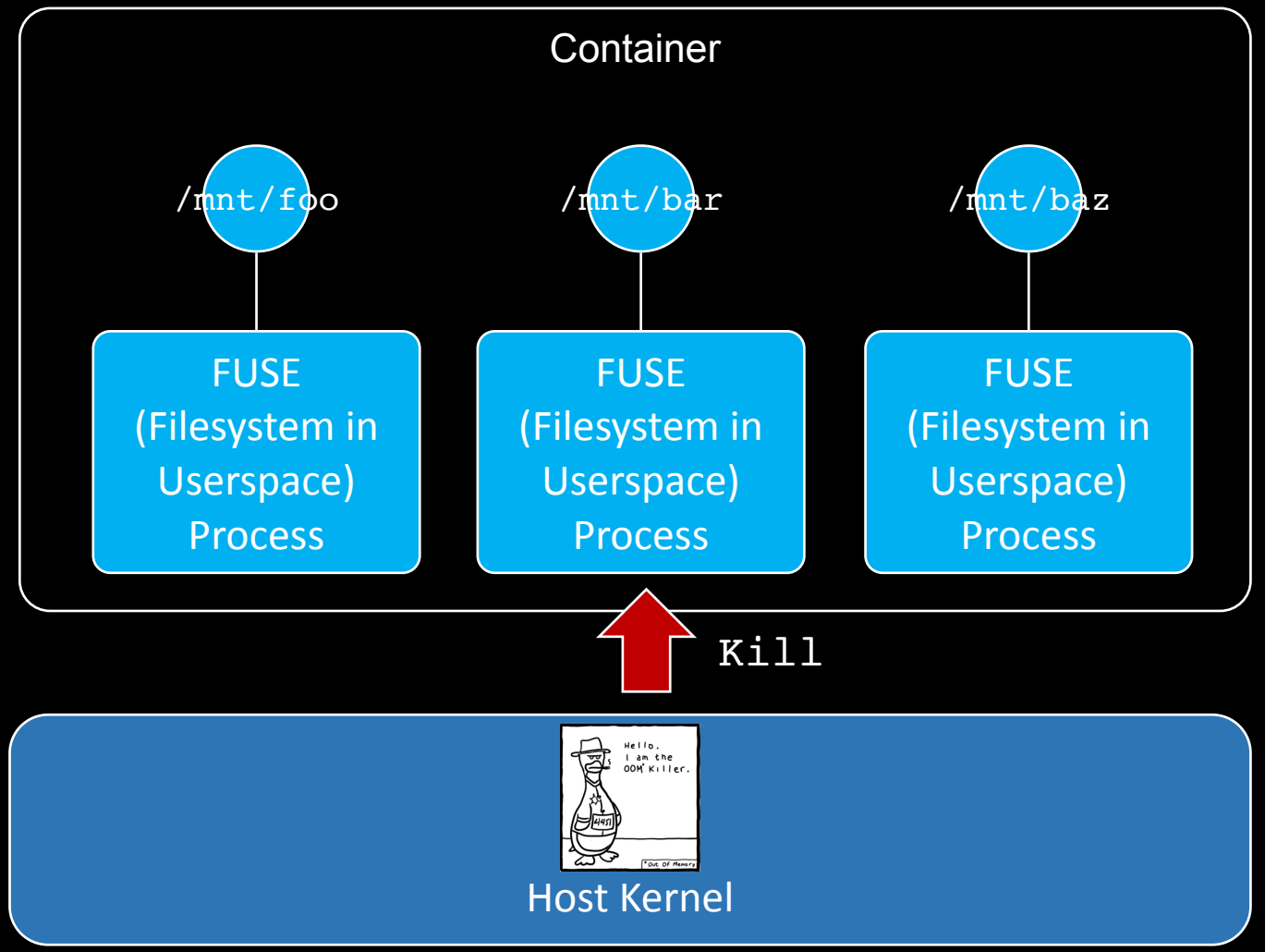
- 사용 중 갑자기 디렉토리내 파일이 사라지고 read가 실패한다…



Intermittent Unmount

범인은 OOM Killer

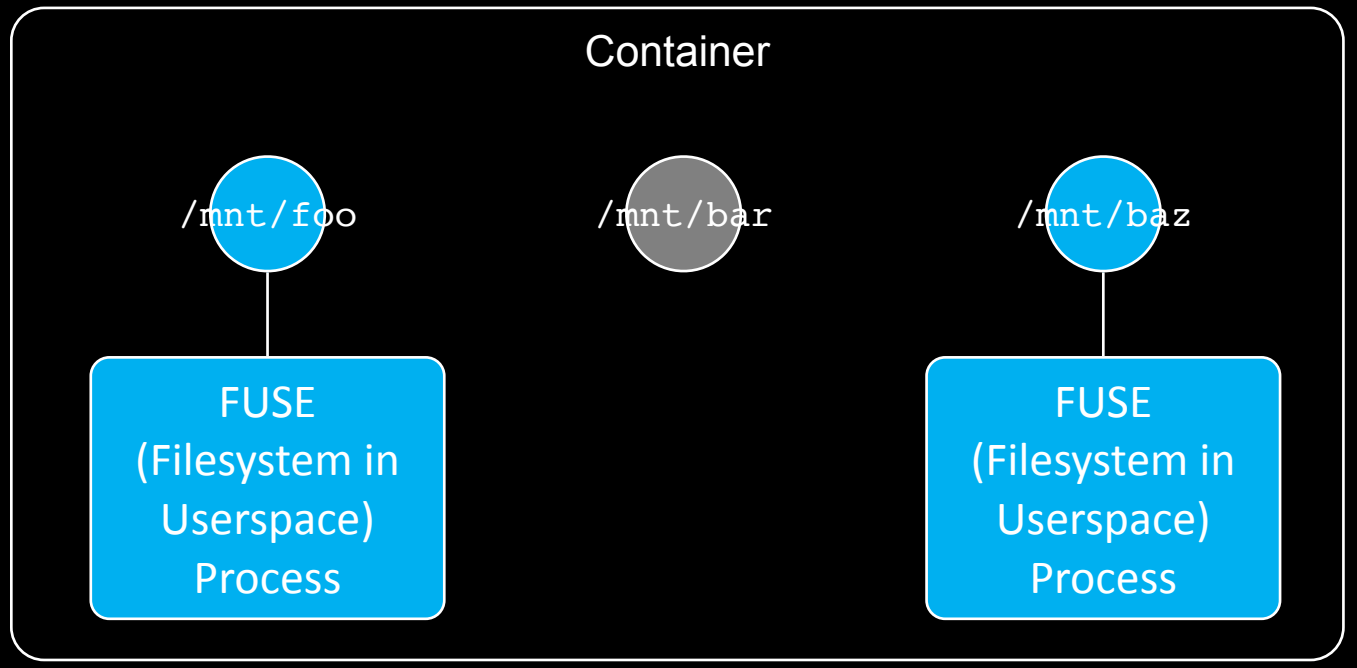
출처: turnoff.com

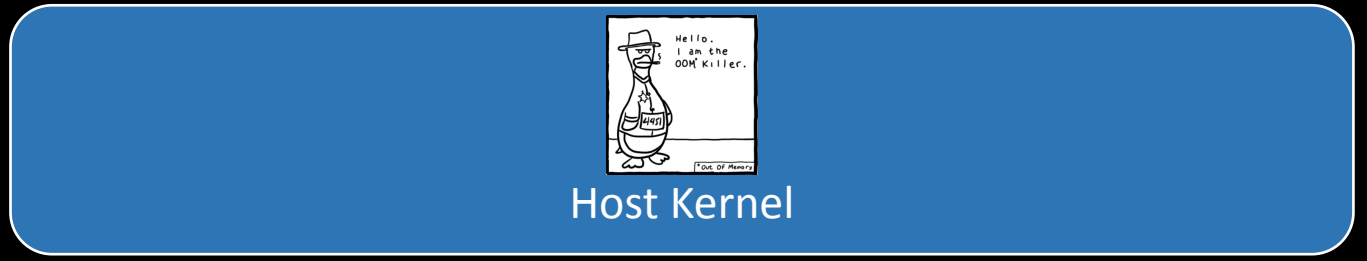


Intermittent Unmount

범인은 OOM Killer

출처: turnoff.com





OOM Score 조절하기

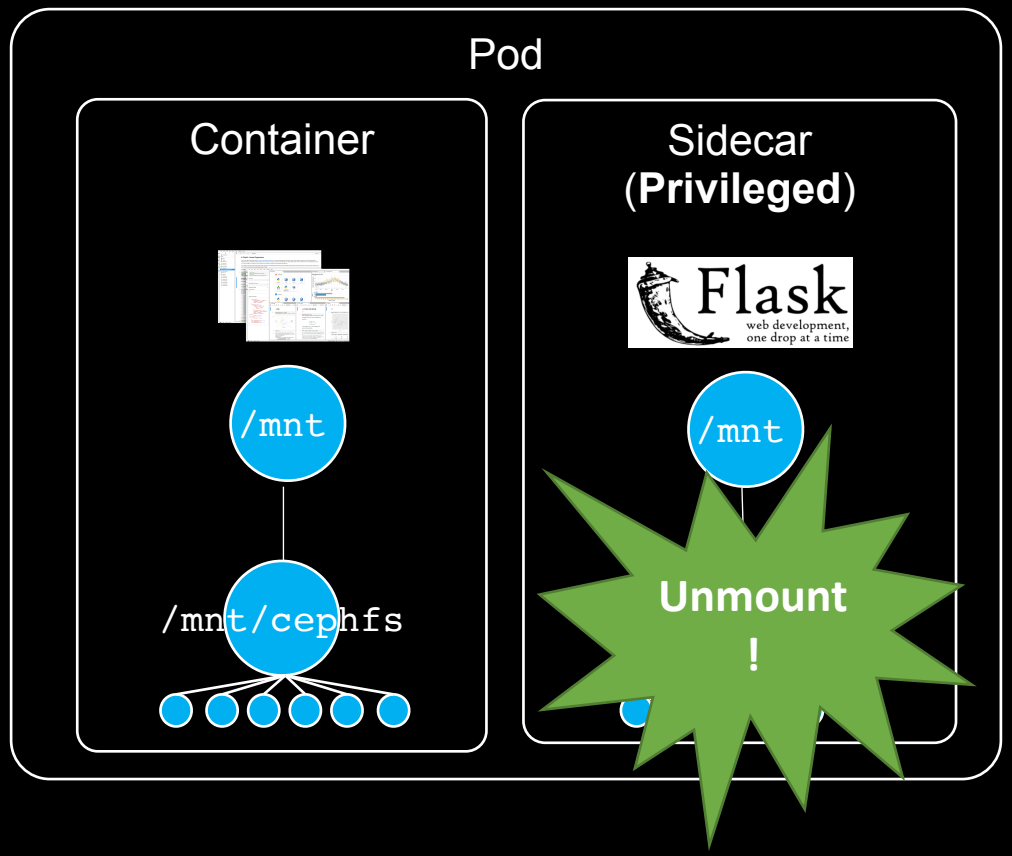
OOM Score = Badness Score = 실제 메모리 사용량 + oom_score_adj

높은 Process가 먼저 죽는다.

Best-effort QoS Class

- 모든 컨테이너의 Resource Request 가 없음

  → oom_score_adj = 1000



OOM Score 조절하기

OOM Score = Badness Score = 실제 메모리 사용량 + oom_score_adj

높은 Process가 먼저 죽는다.

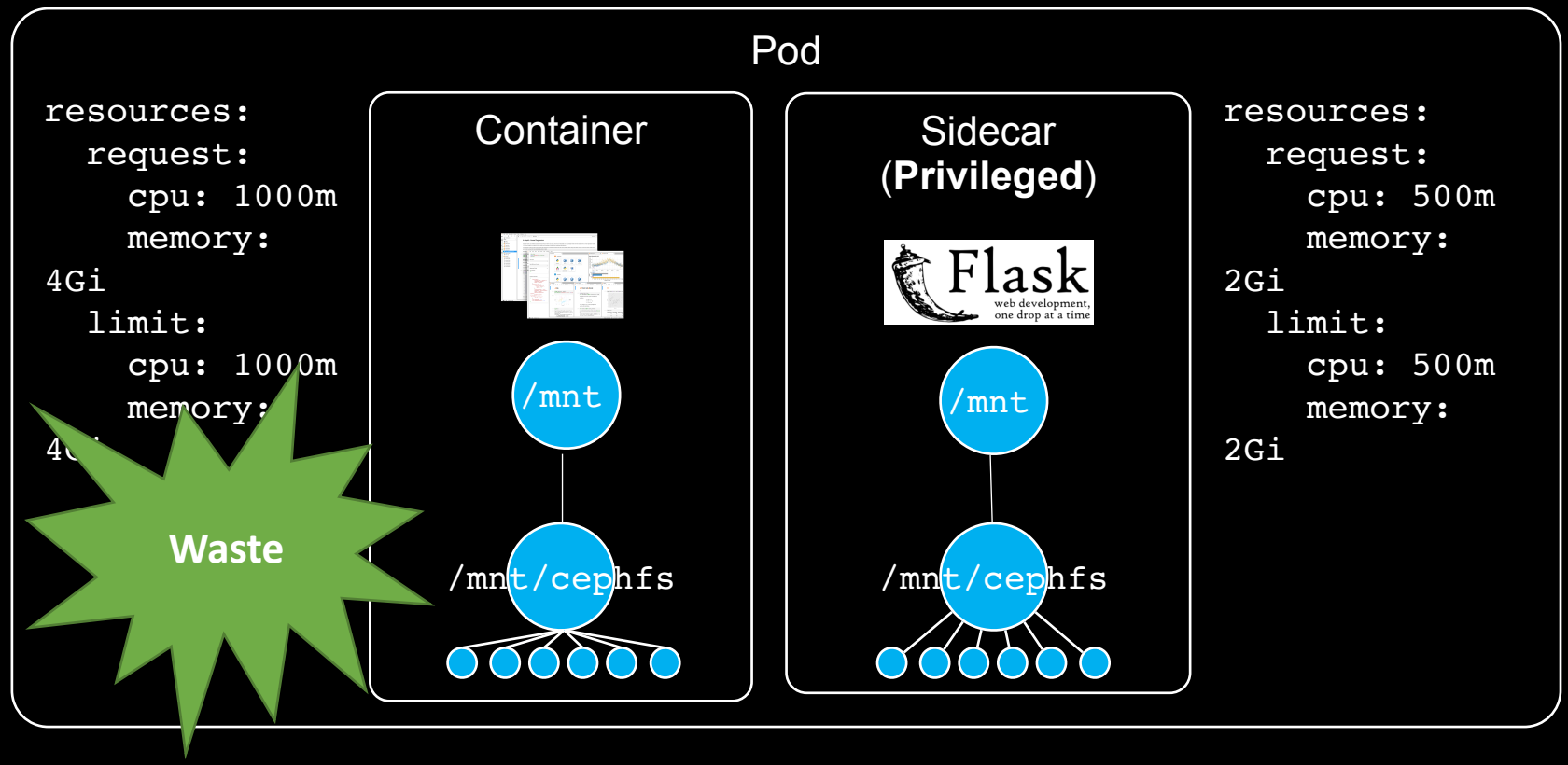
Best-effort QoS Class

- 모든 컨테이너의 Resource Request가 없음

Guaranteed QoS Class

- 모든 컨테이너의 Request, Limit이 동일

  → oom_score_adj = -998

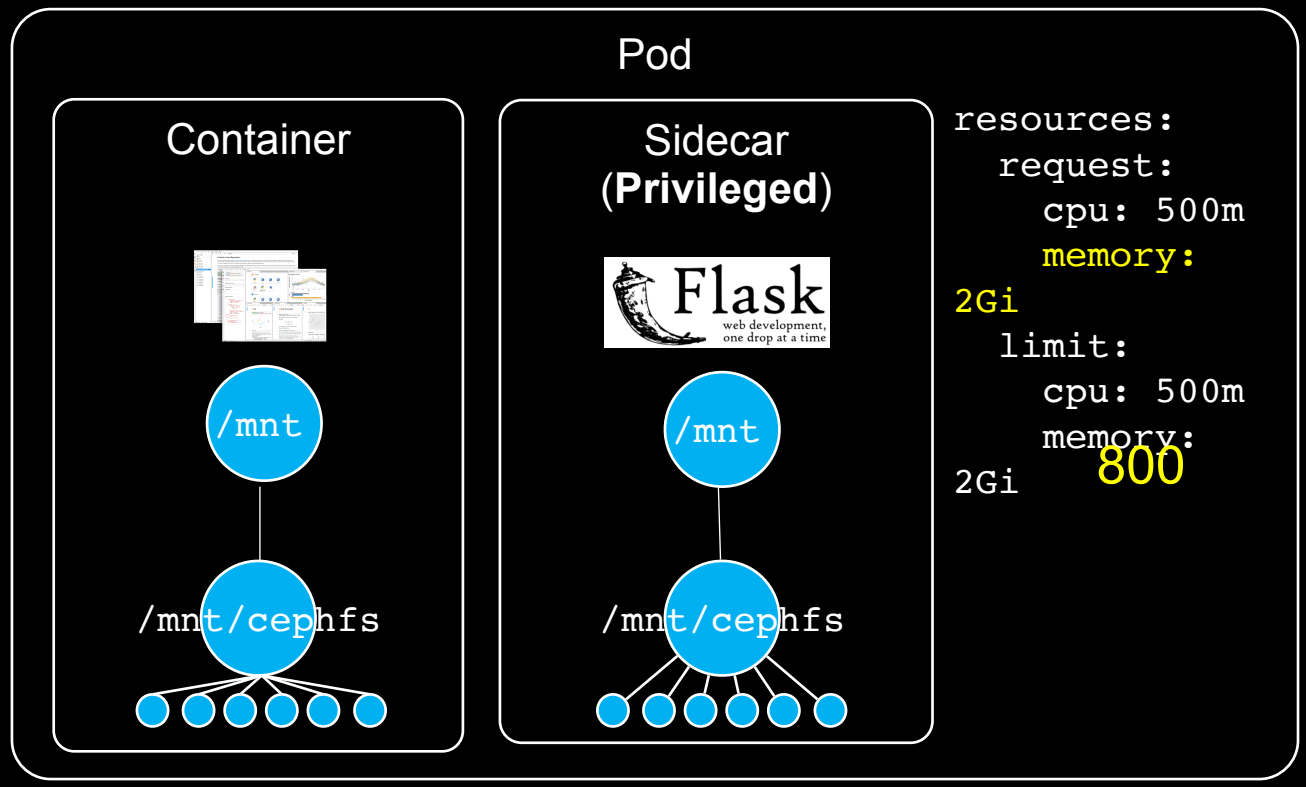


OOM Score 조절하기

OOM Score = Badness Score = 실제 메모리 사용량 % + oom_score_adj

높은 Process가 먼저 죽는다.

Server Memory Capacity: 10Gi



Burstable QoS Class

- 위 두 경우를 제외한 경우

  → Container 별 계산

  : oom_score_adj = [2 , 999]

- Request보다 덜 사용하면 다른 Pod에 의해 Eviction 되지 않음

  → Sidecar는 항상 최대 사용량을 Request 로 지정

  (Limit을 초과 방지 위해 Mount 개수 제한)

Quick Demo in 30 seconds

https://user-images.githubusercontent.com/8223765/97962846-52507980-1df9-11eb-93ce-30572c1bc836.gif