We Do Technology | SK hynix

초고성능 S3 클러스터 구축을 위한 Kubernetes 최적화 달성하기

2025. 09. 16

SK Hynix 신호승

Make IT Intelligent

호기심 용기 시련

그리고

가능한 모든 것을 의심

1. 목표

S3 Performance	**21.8 TiB/s** Throughput at 1.0 EiB, via 484 node cluster with 400Gbps NICs
Network	**Network Overhead 최소화** eBPF, Native Routing, BGP, Host-routing
Kubernetes	**Kubernetes의 병목 구간 제거** **SNAT/DNAT, Cluster Mesh, ECMP, ExternalTrafficPolicy**
AI Ready for Lakehouse	**국내 최대 규모 데이터 플랫폼 서비스 대응과 S3 통합** **Ad-hoc 쿼리 80만건/day, ETL Table 1만개, 실시간 데이터 처리** **30PB 데이터 통합 S3, 수 십개의 AI 서비스 연동**

Cloud Native의 장점은 Legacy를 압도

구분	Bare Metal Linux	Kubernetes 배포
설치/운영 난이도	설치는 비교적 쉽지만 전체 클러스터 관리를 위한 자동화 도구 수준 하락	**Cloud Native 기반 GitOps 활용**
성능 (Throughput/Latency/CPU)	컨테이너 오버헤드는 없으나 eBPF Native 네트워크 구성 어려움	**eBPF + Native Routing을 사용할 경우 최고의 성능 확보 가능**
스토리지/디스크	JBOD/NVMe 직접 연결하여 운영 비용 절감	DirectPV 기반의 CSI 관리 역량 필요
네트워킹/로드밸런싱	F5 VIP+BGP/ECMP/DSR로 고성능 아키텍처 구현 가능하나 HW에 의존도 높음	Software Define Network 구성 가능, eBPF로 최고의 LB 환경 구성 가능
확장성/가용성	서비스 구성에 따라 100% 가용성 확보 가능	K8S의 가용성이라는 또 다른 난이도 해결 필요
보안/멀티테넌시	MinIO 자체 정책 중심. 워크로드 레벨 격리는 별도 설계	네임스페이스, RBAC, NetworkPolicy 활용해 테넌트 격리
관측성/모니터링	Legacy 형태의 Observability 스택의 운영	**Cloud Native Observability와 관련된 표준 스택 활용**
업그레이드/장애조치	수동적인 프로세스가 다수 발생	**Cloud Native가 제공하는 다양한 서비스 활용 가능**
비용/TCO	오케스트레이션 레이어 비용 없음. 자동화 인력 리소스 필요	K8S 클러스터 운영 비용 있지만 GitOps/자동화로 장기 TCO 절감 가능

대규모/고성능 S3가 필요한 경우 비교 대상이 없음

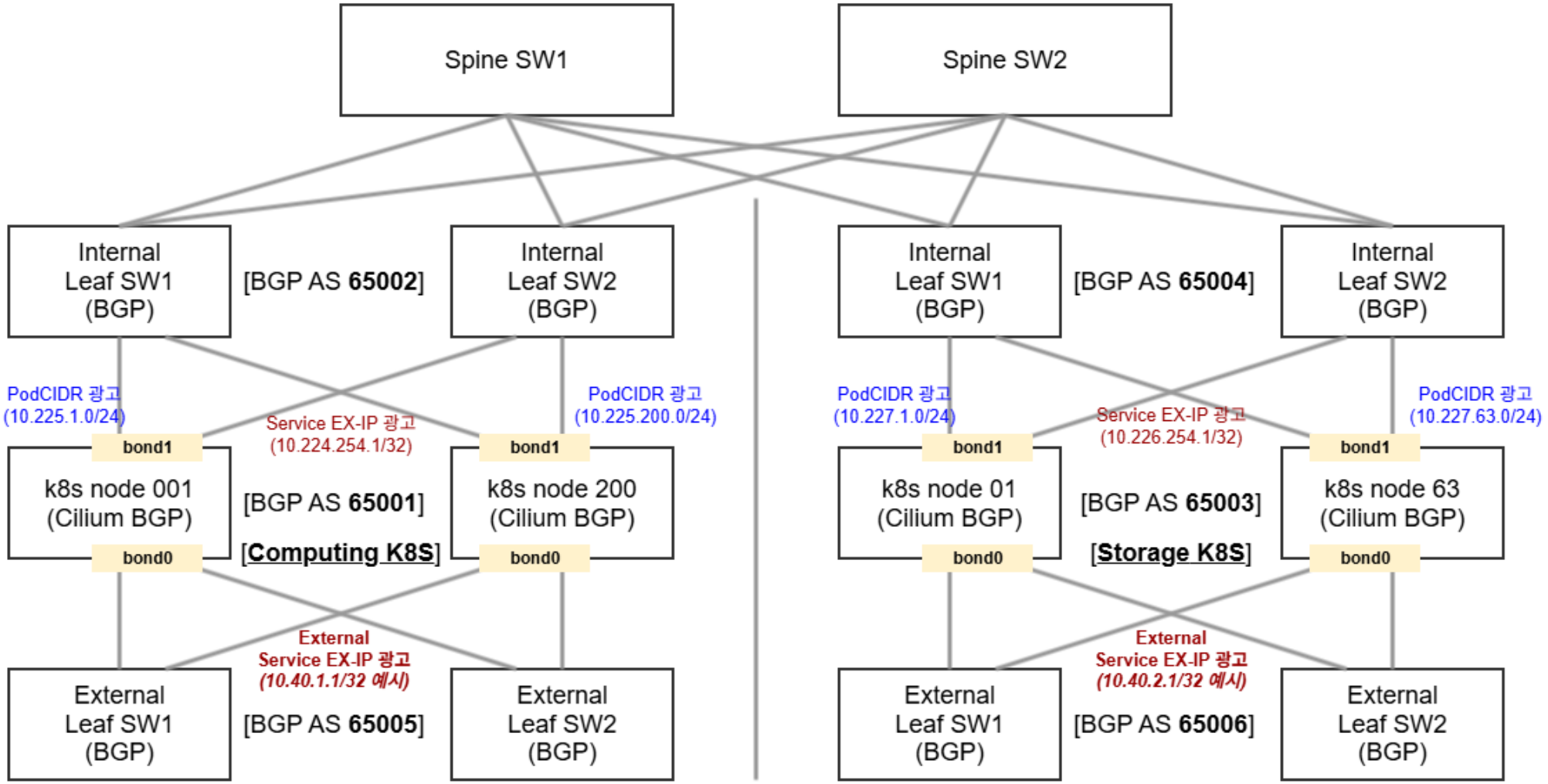
구분	MinIO	Ceph
설계 철학	분산 오브젝트 스토리지 전용 **POSIX FS 같은 블록/파일 계층 없음** -> 오직 S3 API에 최적화 Erasure Coding 기반 단순 구조 -> 디스크를 직접 붙이고, 바로 데이터/파리티 계산 후 저장 → 중간 레이어 없음 Cloud Native 기반 설계, 단일 바이너리, K8S 발전 방향과 align	레거시 설계 - 2006년 개발(HPC/대형 스토리지 환경을 위해 만들어진 범용 분산 스토리지) **다중 인터페이스 지원 - RADOS(내부 API), RBD(Block), CephFS(POSIX FS), RGW(S3 Gateway)** 다양한 유형의 스토리지 프로토콜 지원 ←→ Layer가 많아 I/O 경로 복잡 Rook-Ceph Operator 덕에 K8s에서 돌아가긴 하지만, 원래 설계는 VM/물리 서버 중심 -> CN 설계 라기보단 CN 환경에서 동작하도록 포장
데이터 경로	**Client App → MinIO (S3 API) → Erasure Coding → Disk(XFS/NVMe)** **오버헤드가 극도로 적음** 네트워크, CPU, **디스크 성능을 100% 가까이 활용** 디스크 집합을 EC+SHA256 checksum으로 감싼 구조라서 튜닝 포인트 명확	Client App → RGW (RadosGW) → Librados → RADOS → OSD Daemon → BlueStore → Disk 오브젝트를 S3로 수신 → RGW가 RADOS 계층 전달 → OSD 데몬이 블록 단위로 분해 후 저장 BlueStore가 디스크 위에 동작하면서 또다시 자체 저널링/메타데이터 관리 CPU/메모리/네트워크 hop이 많음 → **디스크 IOPS/Throughput의 50~70% 수준이 한계**
성능 차이	NVMe + 25/40/100G NIC 환경에서 물리 디스크 성능의 95% 이상 도출 오브젝트 워크로드(S3 put/get)의 성능 선형 확장 가능	**MinIO와 동일 HW에서 60~70% 수준** → OSD 프로세스, BlueStore 메타데이터, CRUSH 맵 계산, **내부 복제 오버헤드 발생** 특히 **작은 오브젝트에서 성능이 급격히 떨어짐** (메타데이터 병목) -> RocksDB의 잦은 compaction과 small file merge에 의한 부하 심각 -> 설계 한계 Throughput 중심 워크로드(대형 sequential write)는 괜찮지만, 레이턴시 민감한 환경에서는 불리
스토리지 공간 효율	Erasure Coding으로 평균적으로 물리 공간의 70%를 Usable 영역으로 사용	3-way replication(Erasure Coding Pool로 변경시 성능 저하 발생)
활용 영역	**고성능 S3 전용 스토리지 (AI 학습 데이터, HPC, 빅데이터, 로그 저장)** Small & Large Object 혼합 환경 **Lakehouse 데이터 저장소** 실시간 데이터 저장용 DR/멀티 클러스터 복제, 버전 관리 중심 서비스	**통합 스토리지 플랫폼 필요할 때 (VM 디스크 + 파일 공유 + S3)** Large Object 위주, 성능보다 유연성·기능이 중요한 환경 Legacy 스토리지 대체 및 OpenStack, VM 기반 인프라 통합 데이터 복제·자체 복원력·멀티 테넌시가 더 중요한 경우

4.1 Network - 설계 고려 사항(필수)

- ✓Network Topology – Spine/Leaf, East-West Traffic/North-South Traffic 분리
- ✓Switch – Bandwidth, Data Center Location, Stretch Fabric, ToR/Spine/NIC
- ✓L2 네트워크간 통신 지원 방안 – BGP, BFD
- ✓Native Routing 지원, NIC, Driver
- ✓MTU
- ✓Offloading
- ✓Switch RSS Queue & CPU 성능
- ✓LB with DSR
- ✓Deep Buffer & QDrop

4.1 Network - Topology

Private Network : East-West Internal 트래픽 처리
Public Network : North-South External 트래픽 처리
Stretch Fabric : East-West 트래픽을 spine 없이 우회



4.2 – HW & OS

[HW]

- Linux Kernel version 6.8 이상 -> RHEL 10 or Ubuntu 24.04.02
  - 핵심 이유 **fstrim**
  - Kernel 6.8 이전 버전의 fstrim은 block device에 대한 순차적인 free
  - 작은 free 영역 바로 discard 실행 -> 불필요한 IO 다량 발생
  - MQ 기반으로 discard 요청을 CPU vCore로 병렬 처리
  - discard를 큰 chunk 단위로 묶어서 처리 -> 컨트롤러 내부 병합 효율 향상
  - deallocate 명령어 최적화로 대규모 FS 작업시 4배 이상 빠른 속도 보장
  - SSD는 기본적으로 erase-before-write 구조 -> free block가 부족하면 write 시점에 GC 발생 -> Latency 급증 -> fstrim으로 free block pool 확보 -> Latency 스파이크 감소
  - Free Block이 부족하면 write amplification 발생 -> 디스크 수명 감소
  - S3와 같은 write/delete가 반복많은 workload에서는 필수 -> 물리적인 fresh 상태 유지 핵심

- XFS 파일 시스템
  - 병렬 IO와 metadata 동시 처리에 우수, inode64 기반 대용량 파일시스템을 위한 directory/metadata 관리 강점
  - 대용량 sequential write, multi thread s3 workload에 적합하여 안정적인 throughput 보장 가능
  - noatime, inode64, allocsize, logbufs, logbsize에 대한 고성능 옵션 최적화 가능
  - NVMe의 IO 스케줄러와(none or mq-deadline) 조합 검증으로 latency 확보에 유리

- NVMe의 PCI x 4 인터페이스 검토(CPU Core, PCI Express Gen/IO, NVMe SSD IO)
  - PCIe x 4/5 NVMe SSD는 4개의 lanes 필요
  - Intel Xeon Sapphire Rapids(Gen4)는 CPU당 64 lanes
  - AMD EPYC(Rome/Mialn/Genoa)는 CPU당 128~160 lanes
  - 노드당 20개의 NVMe를 사용할 경우 Intel의 경우 2 Socket 필수 -> NUMA 매핑과 로컬화 필수

- MinIO의 DirectPV/LocalPV + CPU NUMA Aware Scheduling 적용 필수

[OS]

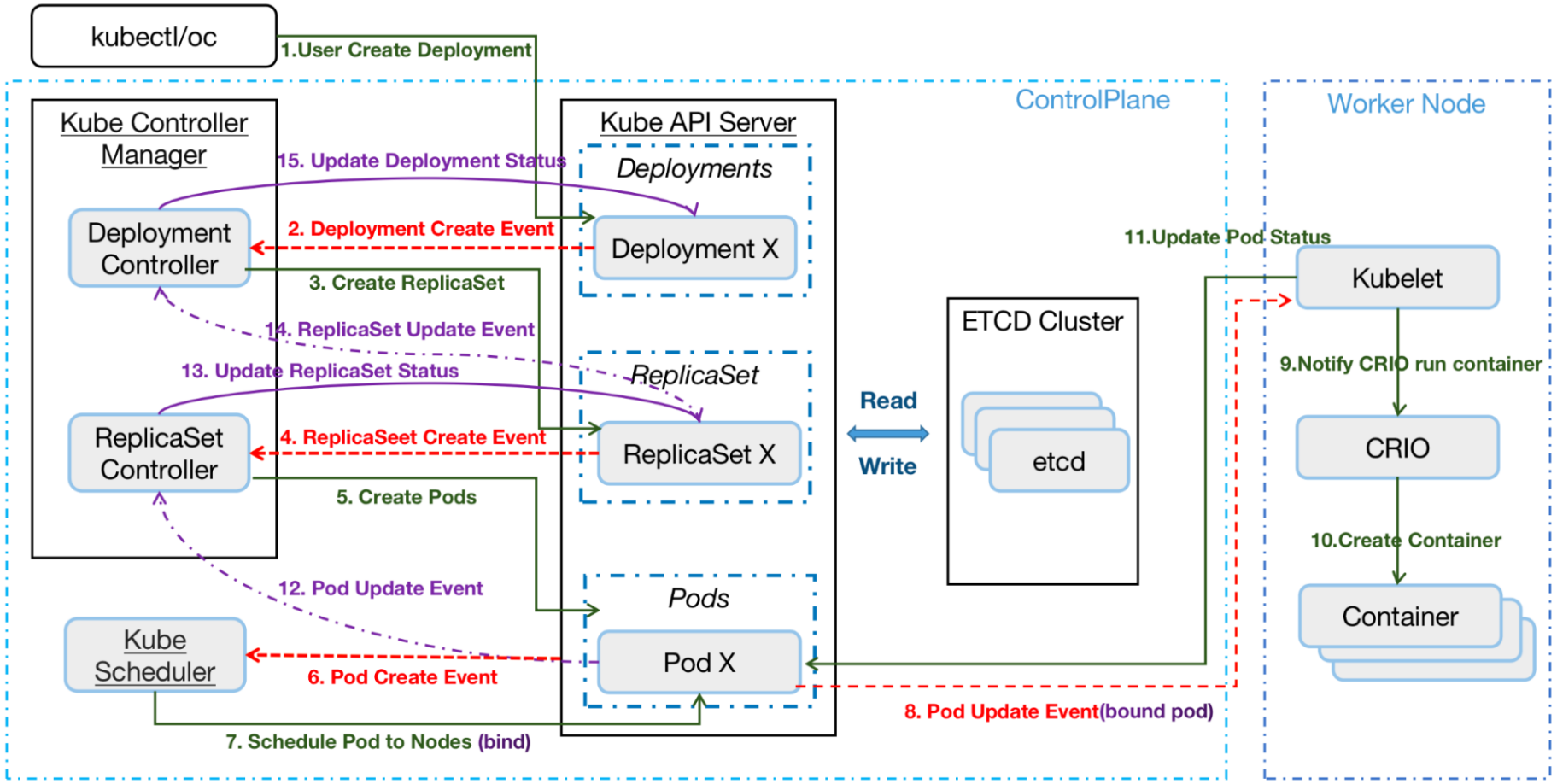
항목	권장값	기본값 (RHEL9.4)	항목 설명	튜닝 목적
fs.xfs.xfssyncd_centisecs	72000	3000 (30초)	XFS 파일 시스템에서 메타데이터를 디스크에 동기화하는 주기를 설정	• 동기화 주기를 늘려 디스크 I/O를 줄이고 성능을 향상시킬 수 있음 • 시스템 장애 시 데이터 손실 위험이 증가할 수 있음
net.core.busy_read net.core.busy_poll	50	0 (비활성화)	네트워크 인터페이스에서의 busy polling 동작을 제어	• busy polling을 활성화하여 지연 시간을 줄일 수 있음 • CPU 사용량이 증가할 수 있음
kernel.numa_balancing	1	1 (활성화)	NUMA 시스템에서 메모리 접근 최적화를 위한 자동 밸런싱 기능을 제어(메모리 접근 지연을 줄일 수 있음)	권장값 기 적용됨
vm.swappiness	0	60	시스템이 스왑 공간을 사용하는 정도를 제어	스왑 사용을 최소화하여 메모리 내에서 작업을 유지하려는 설정
vm.vfs_cache_pressure	50	100	VFS 캐시의 유지 정도를 제어	값을 낮추어 inode 및 dentry 캐시가 더 오래 유지되어 파일 시스템 성능이 향상될 수 있음
vm.dirty_background_ratio	3	10	페이지 캐시에서 'dirty' 페이지의 비율을 제어	값을 낮추어 데이터가 더 자주 디스크에 기록되어 데이터 손실 위험이 줄어들 수 있음
vm.dirty_ratio	10	20	페이지 캐시에서 'dirty' 페이지의 비율을 제어	값을 낮추어 데이터가 더 자주 디스크에 기록되어 데이터 손실 위험이 줄어들 수 있음
vm.max_map_count	524288	65530	프로세스당 메모리 매핑 가능한 영역의 최대 수를 설정	대규모 애플리케이션에서 필요한 메모리 매핑 수를 지원하기 위함
kernel.sched_migration_cost_ns ??	5000000	500000 (0.5ms)	스레드가 다른 CPU로 이동할 때의 비용을 설정	값을 높여 스레드가 현재 CPU에 더 오래 머무르게 되어 캐시 효율성이 향상될 수 있음
kernel.hung_task_timeout_secs	85	120 (sec)	중단된 태스크를 감지하는 시간 제한을 설정	값을 낮추어 중단된 태스크를 더 빨리 감지할 수 있음
net.core.netdev_max_backlog	250000	1000	NIC에서 수신한 패킷을 커널이 처리하기 전에 대기할 수 있는 최대 패킷 수를 설정	값을 높여서 네트워크 트래픽이 급증할 때 패킷 손실을 줄일 수 있음
net.core.somaxconn	16384	4096	서버 소켓이 수신 대기할 수 있는 최대 연결 요청 수를 설정	값을 높여서 많은 동시 연결을 처리하는 애플리케이션에서 연결 요청을 더 많이 수용할 수 있음
net.ipv4.tcp_syncookies	0	1 (활성화)	SYN 플러딩 공격에 대한 방어를 위해 SYN 쿠키를 사용	SYN 쿠키를 비활성화하면 DoS 공격에 취약해질 수 있음
net.ipv4.tcp_max_syn_backlog	16384	4096	SYN 큐의 최대 대기 연결 수를 설정	값을 높여 동시에 많은 연결 요청을 처리할 수

Kernel profile 최적화

https://github.com/minio/minio/tree/master/docs/tuning
https://raw.githubusercontent.com/minio/minio/master/docs/tuning/tuned.conf
https://overcast.blog/kernel-tuning-and-optimization-for-kubernetes-a-guide-a3bdc8f7d255

운영 환경에 대한 정확한 자기 객관화를 위한 질문

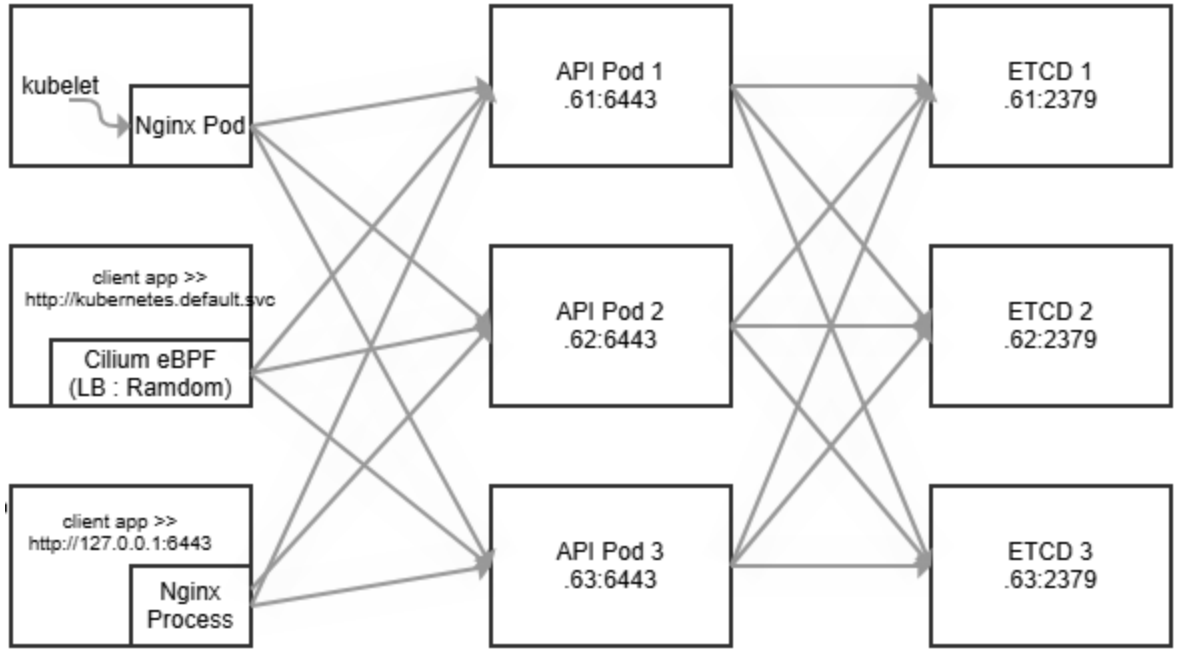
- Fallacies of distributed computing(https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing)
- 진리를 탐구하는 자에게는 인생에서 단 한 번이라도, 가능한 한 모든 것을 의심해야 한다.

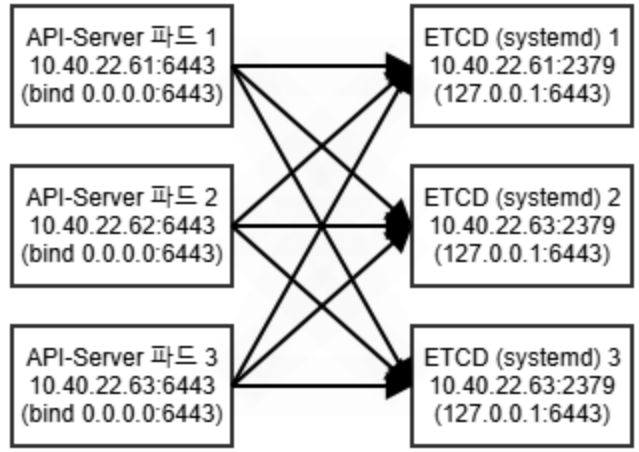


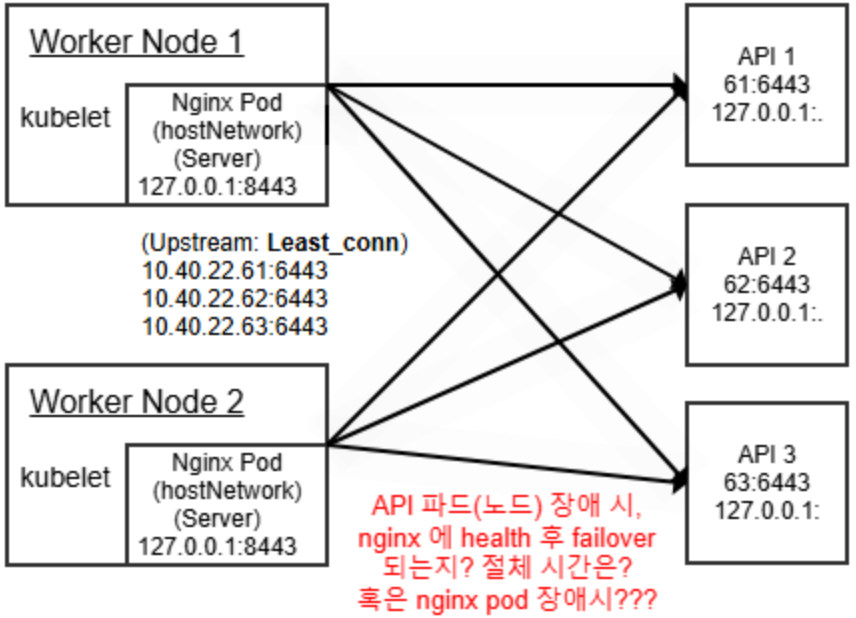
https://www.redhat.com/rhdc/managed-files/kubernetes-img1_1.png

K8S 기본 서비스 구성요소의 통신, 흐름, 장애 상태 판단을 위한 구간별 정보 파악

- App/Kubelet/Cilium eBPF/Nginx –> API Server로 이어지는 분산 과정의 LB 알고리즘
- API Server –> ETCD
- Worker Node(Kubelet) –> API Server
- 각 구간별 Health Check에서 장애 발생시 Failover의 설정과 과정

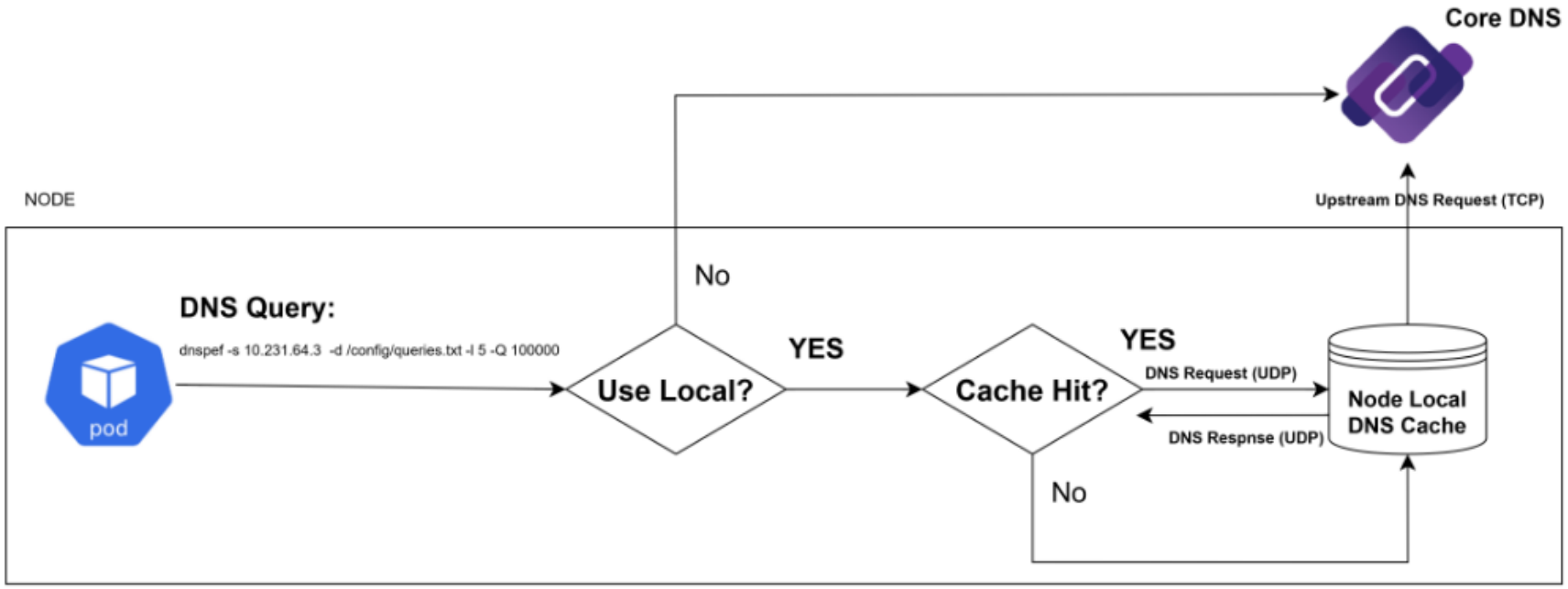






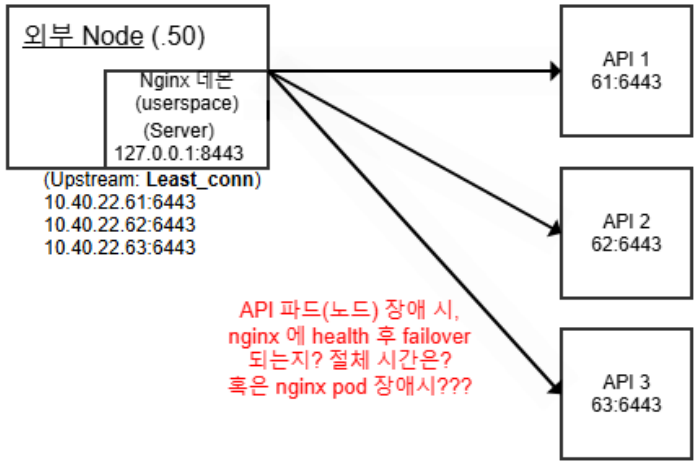
CoreDNS 상태에 대한 투명화

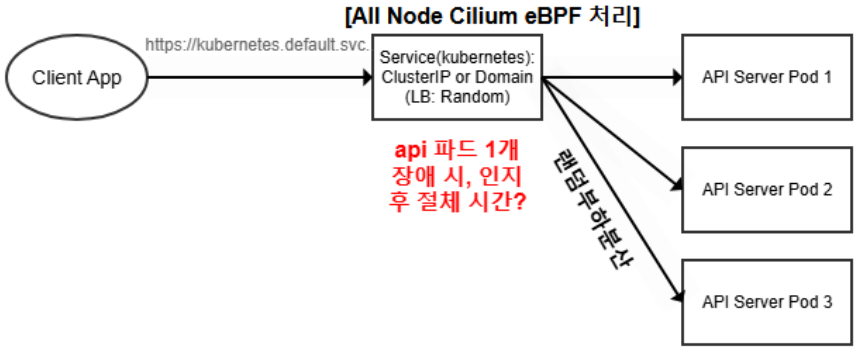
- Pod의 질의 -> NodeLocal DNScache -> (Cache Miss) → CoreDNS -> DNS Server 전체 구조에 대한 이해
- CoreDNS HPA + Resource Limit
- Corefile 캐시 최적화 (cache plugin) - success/denial 용량 확대, prefetch, serve_stale -> cache hit/eviction 지표 검증
- Autopath로 검색 경로(ndots)로 인한 추가 쿼리를 감소로 지연 방지
- QPS, 응답 코드, Cache Hit, Plug-in Metric 지표 추적
- Endpoints 대신 EndpointSlice 기반 운영 – watch 이벤트 크기 감소, diff 효율 증가, 레코드 최신화 지연과 SERVAIL 확률 감소

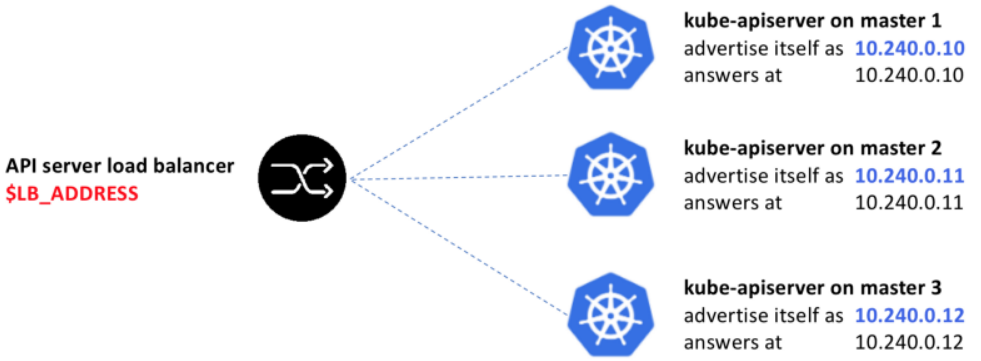


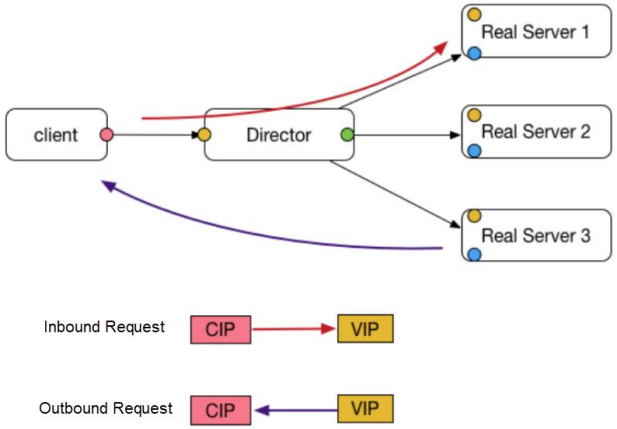
K8S 외부 클러스터에서 발생하는 트래픽 분산 관리 방안

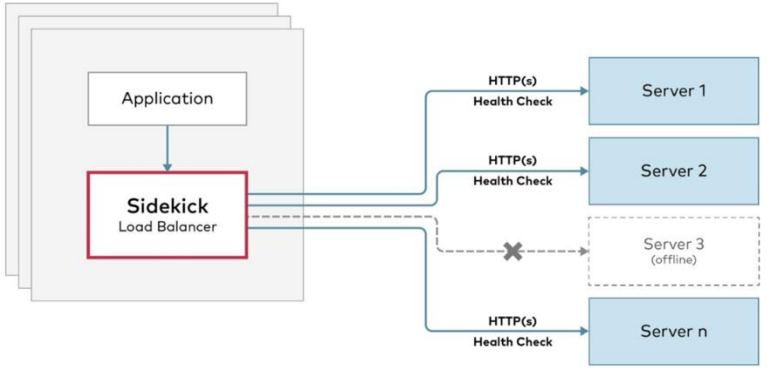
- 외부 App 자체의 Client LB
- K8S API Server의 물리 LB
- Cilium LB IPAM
- 외부 App Linux의 LVS 기반 Client IPVS LB
- 외부 K8S의 Sidecar Proxy 기반 Client LB





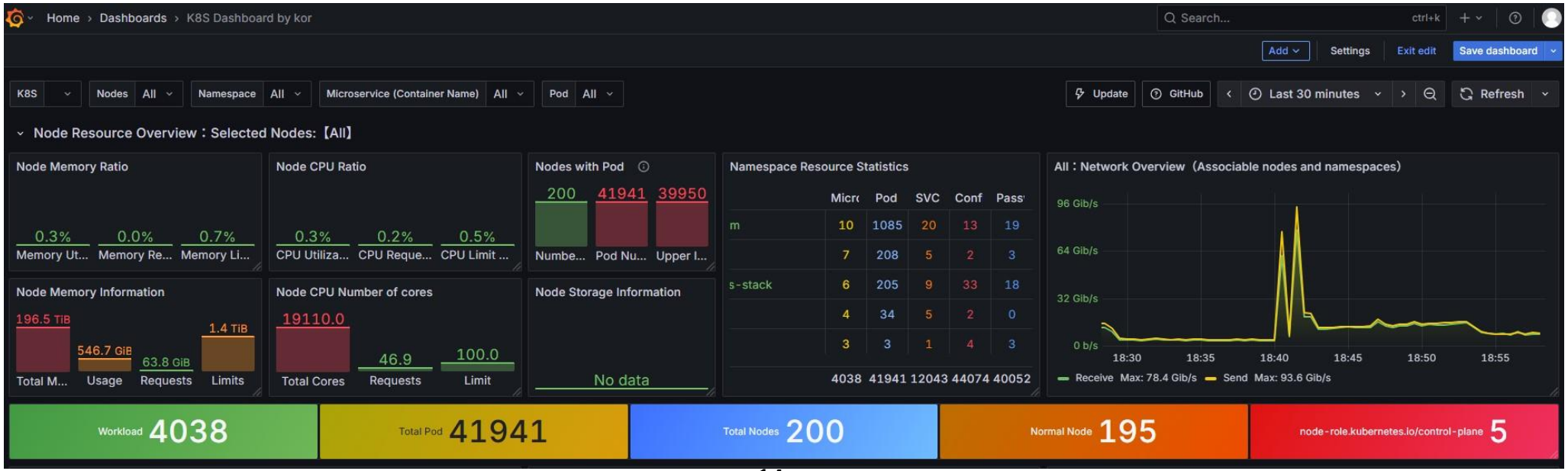






운영 환경을 위한 최적의 환경 검증 – Kube-burner Test 결과를 분석하여 반복 검증

Test 리소스	Max Pod(Per Node)	API Server QPS	ETCD	Cilium	Coredns (nodelocaldns)
▪ CPU 200,000 ▪ Mem 200TiB ▪ NIC(per node) - 25G * 2 Bond(내부) - 25G * 2 Bond(외부)	200 Pod * 200 Nodes = 40,000개 Pod 실행 + 그외 리소스 configmap 40,000개 secret 40,000개 service 12,000개	총 QPS: 5380 총 Request수: 322823 (캐싱 포함) 최대 API 서버 동시 처리수: 60(mutating) CPU: 6.57core memory: 거의 모든 api 서버가 동일하게 12Gib 이상 사용	최대 부하 기준 (4000pod 기동) 상태 결과 fsync latency (disk) : Max 15ms → 안정 기동 기준 DB 사용량 : 2.5GB → 안정 메모리 사용수준 : 4GB → 안정 cpu 사용 수준 : 4 core → 안정 peer RTT (etcd 동기화) : 6.35 ms → 안정 client 요청 QPS : 2400c/s peer propposal 지연 건수 : 3 peerpropposal 실패 건수 : 0	API Call latency : 1초 이내 완료 BPF map pressure : 10% 이내 처리 Node 별 IPAM : 200개 할당 정상	Coredns QPS: 1.11K(TCP) Nodelocal QPS: 15M(UDP)



4.3 Kubernetes(6) – 현시점의 최적화 값

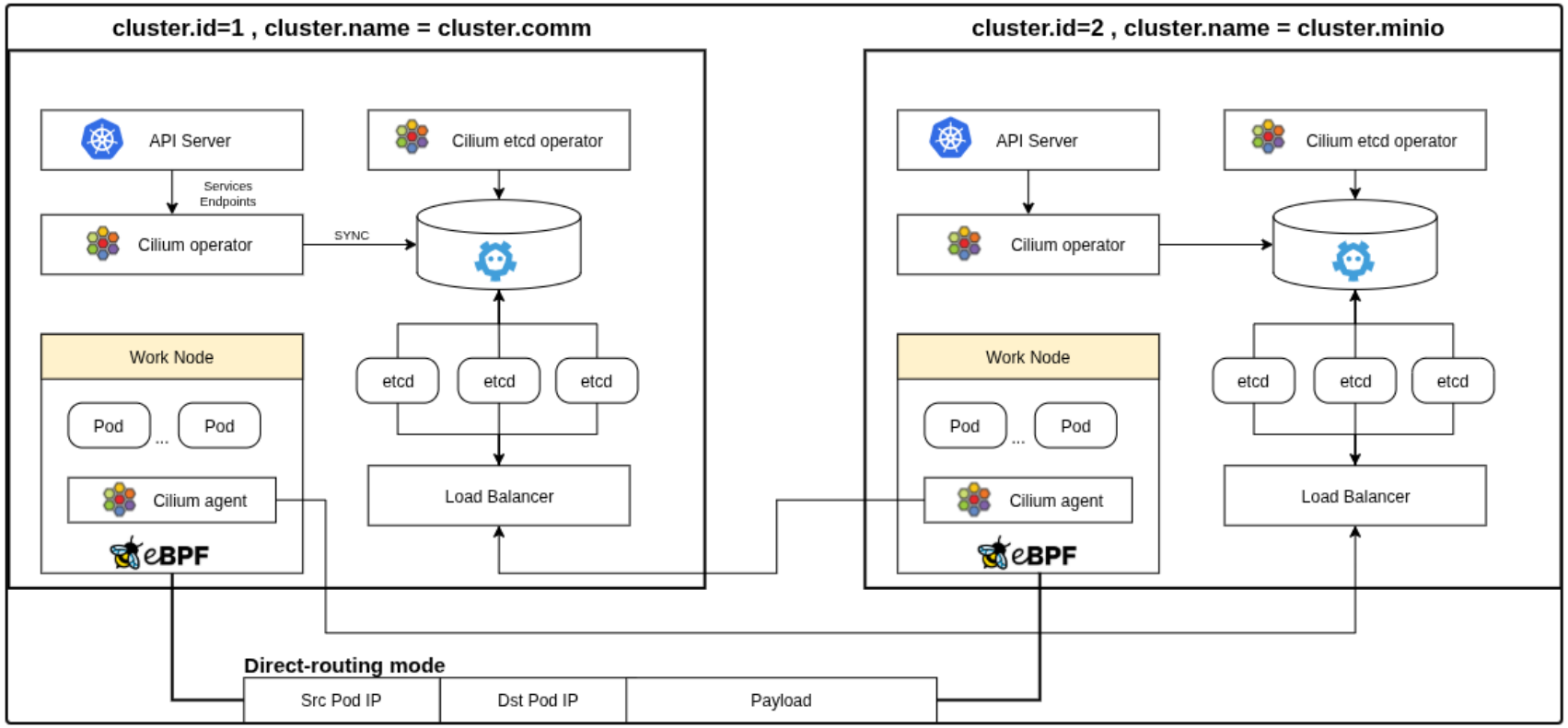
운영 환경에 대한 정확한 자기 객관화 -> 병목 구간 파악 후 최적의 설정 값 도출

Resource	최적화 Value	Default	비고
API	max-mutating-requests-inflight: 400 max-requests-inflight= 800 delete-collection-workers=200 CPU: 16core Memory: 24Gi	max-mutating-requests-inflight: 200 max-requests-inflight= 400	DeleteCollection Kubernetes API 서버에 요청(예: 레이블 선택기를 기반으로 여러 리소스 삭제)이 발생 하면 Kubernetes API 서버는 kube-apiserver이러한 워커를 사용하여 해당 컬렉션 내 개별 객체의 삭제를 처리, Delete 작업 시 많은 부하가 걸리기 때문에 delete-collection-workers 200 으로 상향, inflight 수치는 5000개의 Pod Spike 테스트 시 최초로 429 code가 발견되었을 만큼 필요성이 크지 않지만 delete-collection-worker의 증가로 인한 inflight reuqest의 처리량이 높아질 가능성, APF의 우선순위에 따른 API 의 부하로 인한 Delay의 가능성, Controller-Manager의 WorkQueue 지연, 최대 CPU 9,Memory 13Gi 정도의 낮은 리소스 사용량을 바탕으로 inflight의 수치를 2배씩 상향 delete-collection-worker 의 경우 delete 작업 시 controller-manager 의 부하와 controller-manager의 concurrent 수치 상향으로 default=1 에서 controller-manager의 gc concurrent와 비슷한 수치인 200으로 상향. DeleteCollectionKubernetes API 서버에 요청(예: 레이블 선택기를 기반으로 여러 리소스 삭제)이 발생 하면 Kubernetes API 서버는 kube-apiserver이러한 워커를 사용하여 해당 컬렉션 내 개별 객체의 삭제를 처리. CPU와 Memory는 테스트시 1 : 1.5 비율 정도로 증가, 최대 CPU, Memory 기반으로 limit 설정
Controller-manager	concurrent-gc-syncs: 200 concurrent-deployment-syncs: 50 concurrent-endpoint-syncs : 50 concurrent-replicaset-syncs : 50 concurrent-rc-sync: 50 concurrent-replicaset-syncs: 50 concurrent-namespace-syncs: 50 concurrent-service-syncs=50 concurrent-job-syncs s=50 qps: 150 burst: 300	concurrent-gc-syncs: 20 concurrent-deployment-syncs: 5 concurrent-endpoint-syncs : 5 concurrent-replicaset-syncs : 5 concurrent-rc-sync: 5 concurrent-replicaset-syncs: 5 concurrent-namespace-syncs: 5 concurrent-service-syncs=5 concurrent-job-syncs s=5 qps: 20 burst: 30	concurrent 값의 변경이 workqueue latency metric에 영향도 체크 우선, 실제로 workqueue latency의 P99가 최대 10s로 느려지는 케이스가 존재 하지만, concurrent 값 default 5 에서 20배 올린 100이 이 케이스를 해결해주지는 못함. 하지만 CPU 사용량과 Memory 사용량이 크지 않고 ETCD 부하가 관측되지 않아 10배로 상향 후 관찰
ETCD	ETCD_ELECTION_TIMEOUT: 2700 ETCD_HEARTBEAT_INTERVAL: 270 ETCD_QUOTA_BACKEND_BYTES=8589934592	ETCD_ELECTION_TIMEOUT: 1000 ETCD_HEARTBEAT_INTERVAL:100 ETCD_MAX_REQUEST_BYTES: 1572864	테스트 과정에서 최대 peer 통신 Latency인 180ms 를 기준으로 하여 1.5 배로 heartbeat_interval 설정, ElectionTimeout은 heartbeat_interval의 10배로 설정. ETCD_QUOTA_BACKEND_BYTES 는 공식문서의 최대 권장값으로 설정
Kubelet	qps: 150 burst: 300	qps: 50 burst: 100	EventPLEG 설정이 Pod 시작 시간을 오히려 늦추는 현상 발생, 기존 Polling 방식의 PLEG가 kubelet 의 Overhead가능성이 있지만 Kubelet의 CPU,Memory 사용이 많지 않아 Polling 방식 유지
Scheduler	qps: 150 burst: 300	qps: 50 burst: 100	2차 테스트 기준 qps:150,burst: 300 으로 ClientThrottling 발생 안함. 추후 scheduling 에 문제가 있을 경우 상향

4.4 CNI(1) – 최적의 아키텍처 도출을 위한 가설 검증

수 많은 테스트, 시행착오, 가설 검증과 관련된 시련의 시간 필요

- Cilium CNI의 eBPF 기반 네트워크 구성 필수
- CNI 메뉴얼 전체 Review 필수



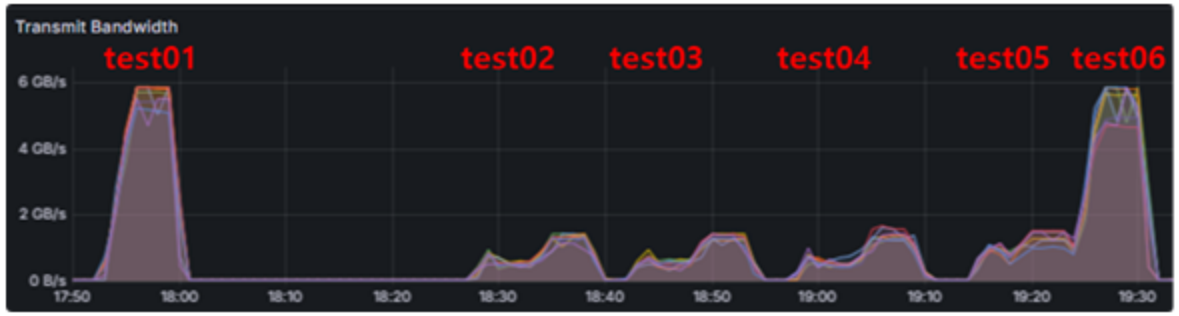
https://docs.cilium.io/en/stable/operations/performance/
https://github.com/redhat-performance/tuned
https://docs.cilium.io/en/stable/operations/performance/tuning/#tuned-network-profiles

Cilium CNI가 제공하는 라우팅 구조의 이해와 성능 검증

- Native Routing vs Overlay Mode 성능 비교
- Iperf, Hperf, Warp 등으로 특성별 성능 검증
- Test 조건
  - object size: 64MiB
  - concurrent: 140
  - warp client# 7ea (각 20개씩 동시 부하)
  - 접근 URL
  - clusterip(local): minio.test-objectstore.svc.cluster.local:9000
  - clusterip(remote): minio-global.test-objectstore.svc.cluster.local:9000
  - nodeport: 10.xx.xx.62:32001

GET : overlay mode는 native routing 대비 2.9% throughtput 하락 (latency는 3.6% 증가)
PUT : overlay mode는 native routing 대비 19.7% throughtput 하락 (latency는 25.7% 증가)

	Mode	Service	Get Throughput(MiB/s)	Put Throughput(MiB/s)	Get IOPS (MiB/s)	Put IOPS (MiB/s)	Get Latency (ms, TTFB)	Put Latency (ms)
test01	native routing	clusterip	22953.7	15415.15	358.65	240.86	15	495.9
test02	overlay	clusterip	22287.1	12521.61	348.24	195.65	15	624.5
test03	overlay	clusterip	22290.32	12454.69	348.29	194.6	14	621.8
test04	native routing	clusterip	22954.01	15670.11	358.66	244.85	13	495.5



https://github.com/minio/warp https://iperf.fr/ https://github.com/minio/hperf

Multi K8S 클러스터 환경의 L2 네트워크 구성 최적화

- S3 K8S 클러스터와 Computing K8S 클러스터의 고성능 통신 구조 달성 필수
- 성능
  - BGP는 Cluster Node 간 Overlay 대비 네트워크 **대역폭 성능 3% 이상 증가**
  - 데이터 인입 시 **요청 노드 분산**
- Overlay + Cluster Mesh
  - K8S 클러스터간 서비스 호출에 ClusterIP 사용
- Overlay + L2 Announcement (+ DSR)
  - 서비스 별 특정 노드 1대만(병목 가능) 인입 요청을 받고, 대상 파드가 배포된 노드들에 전달 후 DSR 리턴
- Client Side LoadBalancer
  - Client 구성 설정(LVS, IPVS, Sidecar)의 복잡성만 극복할 수 있으면 클러스터간 분산처리 가능
- BGP 사용
  - 네트워크 장비가 특정 Service(LB EX-IP)에 대해서 서빙하는 노드들로 분산 전달(ECMP)
  - 클라이언트 측 자체 LB 구현 불필요
  - S3 클러스터 L2 네트워크와 연결하는 가장 개선된 방안

Plan A : Bond0(NodePort) , Bond1(Native Routing + BGP + ClusterMesh)

Plan B : Overlay(L2 + DSR) + ClusterMesh (+ Client-Side LB)

Plan C : Native Routing + BGP + ClusterMesh

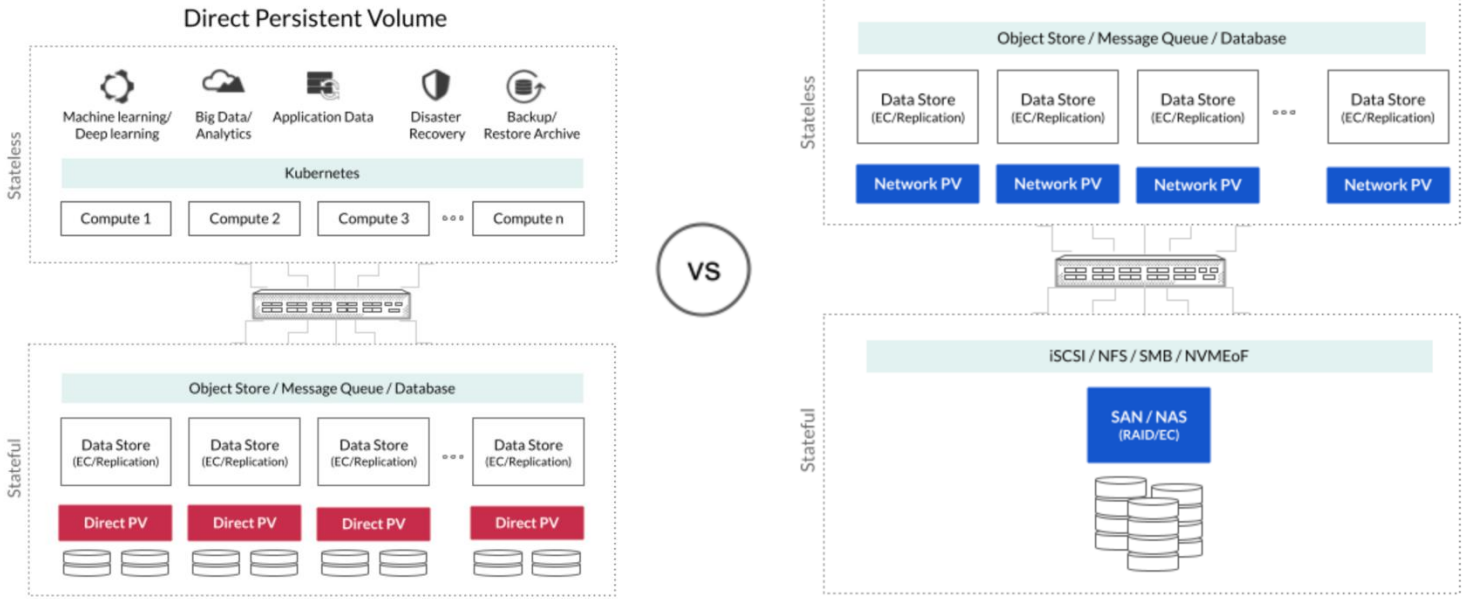
K8S 클러스터의 객관화 테스트에 했던 내용을 Cilium 설정을 변경하여 동일하게 적용

- **10,000개 Job * 11 Pod**(10GB) = ETL Job
- **555.5개 Ad-Hoc Query * 10 Pod**(10GB) * 60분 * 24시간 = 80만개
- **10,000개 Spark Job * 10 Executors** = 100,000개 Spark Pod 동시 실행
- Istio의 Performance and Scalability Test 참고(https://istio.io/latest/docs/ops/deployment/performance-and-scalability)
  - Service(1,000개), Pod(2,000개), Request Rate(초당 7,000건), Throughput, tail latency, CPU/메모리 소비량, idle 시 리소스 소비량 등

항목	설명	적용값	근거
routingMode	네이티브 라우팅 or 오버레이 라우팅 모드 지정	native	네트워크 통신 구간 최적화를 위함
bpf.datapathMode	노드에서 파드로 진입 모드 지정	netkit	홉 줄이고, 파드 진입 시 최적화를 위함
bpf.masquerade	노드에서 외부로 통신 시, 매스커레이딩 처리 방식	true	기존 IPtables 대신 eBPF로 동작 처리
enableIPv4BIGTCP	Large size MTU 설정 없이도, 송수신시 최대 단위로 전달	true	테스트 검증 필요.
endpointRoutes.enabled	노드에 배치된 파드별 개별 라우팅 주입 여부	false	굳이 필요하지 않음
endpointHealthChecking.enabled	엔드포인트 헬스체크 기능	false	대규모에서는 굳이 중복 헬스체크 불필요.
healthChecking	노드 상태, cilium-agent 상태 체크 기능	false	대규모에서는 굳이 중복 헬스체크 불필요.
kubeProxyReplacement	legacy kube-proxy 를 대체하는 (종합적인) 기능 집합	true	핵심 설정으로 활성화.
loadBalancer.acceleration	Cilium 이 LB 처리 시 동작 모드	best-effort	NIC이 XDP 지원 시, XDP 고속 처리.
loadBalancer.mode	Cilium 이 LB 처리 시 백엔드 대상 연결 시 모드	snat	BGP ECMP, ExternalTrafficLocal(Local) 사용 시 최적값
localRedirectPolicy	Cilium eBPF로 특정 트래픽을 처리할 수 있는 기능	true	nodelocal dns 트래픽 가로채기를 eBPF로 처리하기 위함
…	…	…	…

DirectPV - 물리 Disk를 중간 Layer 없이 PV로 바인딩하는 Cloud Native CSI 드라이버

- Ceph와 같은 중간 경로(App->RGW->RADOS->OSD->BlueStore->RocksDB->Disk)가 없는 단순한 경로 구조
- **매우 경량화된 metadata 정보**(XFS 파일시스템의 File name과 Object name만 매핑 -> Disk 자체의 inode/dir tree가 metadata 자체
- 운영 복잡성 감소
  - Disk == PV == MinIO이기 때문에 복잡한 rebalance/placement 불필요
- Cloud Native 아키텍처 CSI
  - 보통의 CSI Driver는 LVM, Ceph RBD, iSCSI와 같은 중간 계층을 붙여 PV 생성
  - **DirectPV는 node의 raw 블록 디바이스를 그대로 PV로 노출하여 오버헤드 최소화**
- StorageClass 단순화
  - Disk 단위로 PV 생성하여 MinIO Pod에 직접 마운트
  - Erasure Set 구조가 K8S 스케줄링과 일치
  - StorageClass를 tier별 정책 적용(NVMe, SSD, HDD)
- Pod 스케줄링 연계
  - **topologySpreadConstraints 혹은 anti-affinity와 조합해 erasure set을 rack/zone별로 분산 배치**
  - 다른 CSI Driver보다 고가용성 모델 대비 직접적인 우위 아키텍처
- GitOps 친화
  - GitOps와 연결하여 PV/PVC 리소스 관리



4.5 MinIO(2) – 기본 설정 최적화

K8S/Cilium과 유사한 최적화 과정 필요

- 기본적인 OS 튜닝 파라미터는 Tuned 활용
- https://github.com/minio/minio/tree/master/docs/tuning
- Erasure set 최적화 – 데이터의 종류, 운영 상황에 맞는 set 설계
- topologySpreadConstraints - 동일한 erasure set이 동일 노드/랙에 저장되지 않도록 최적화

[vm] 메모리 관리
transparent_hugepage = madvise
THP를 앱 요청 시에만 사용하여 예기치 못한 지연을 줄이고 메모리 효율을 확보

[sysfs] THP 세부 설정
/sys/kernel/mm/transparent_hugepage/defrag = defer+madvise
메모리 결합(defrag)을 지연시키고 앱이 요청할 경우에만 수행
/sys/kernel/mm/transparent_hugepage/khugepaged/max_ptes_none = 0
빈 PTE가 포함된 페이지는 결합하지 않음 → 불필요한 stall 방지

[cpu] CPU 주파수/전원 정책
force_latency = 1
레이턴시 최적화 모드 활성화
governor = performance
CPU 클럭을 성능 위주로 고정
energy_perf_bias = performance
EPP를 성능 우선으로 설정
min_perf_pct = 100
최소 성능 비율 100% 유지

[네트워크 성능 튜닝]
net.core.netdev_max_backlog = 250000
NIC 백로그 큐 확장.
net.ipv4.tcp_syncookies = 0
SYN cookies 비활성화(성능↑, 보안↓).
net.ipv4.tcp_max_syn_backlog = 65535
SYN 큐 확장.
net.core.wmem_max / rmem_max = 4194304
소켓 버퍼 최대값 4MiB.
net.ipv4.tcp_rmem = 4096 87380 4194304, tcp_wmem = 4096 65536 4194304
TCP 버퍼 최소/기본/최대값 설정.
net.ipv4.tcp_timestamps = 0
타임스탬프 비활성화.
net.ipv4.tcp_sack = 1
선택적 확인응답(SACK) 활성화.
net.ipv4.tcp_low_latency = 1
TCP 저지연 모드.
net.ipv4.tcp_adv_win_scale = 1
TCP 윈도/앱 버퍼 비율 조정.
net.ipv4.tcp_slow_start_after_idle = 0
유휴 후 슬로 스타트 비활성화.
net.ipv4.tcp_mtu_probing = 1, tcp_base_mss = 1280
MTU 문제 시 MSS 조정.
net.ipv6.conf.*.disable_ipv6 = 1
IPv6 비활성화.
net.ipv4.tcp_fin_timeout = 20
FIN_WAIT2 타임아웃 단축.
net.ipv4.ip_local_port_range = 1024 65535
에페메럴 포트 범위 최대화.

[sysctl] 파일시스템/NUMA/네트워크/VM 설정
fs.xfs.xfssyncd_centisecs = 72000
XFS 동기화 주기를 12분으로 늘려 백그라운드 부하 완화.
net.core.busy_read = 50, net.core.busy_poll = 50
NAPI busy-polling으로 지연 감소.
kernel.numa_balancing = 1
NUMA 메모리 자동 밸런싱 활성화.
vm.swappiness = 0
스왑 사용 금지.
vm.vfs_cache_pressure = 50
캐시를 오래 유지.
vm.dirty_background_ratio = 3, vm.dirty_ratio = 10
쓰기 스톰 방지.
vm.max_map_count = 524288
메모리 매핑 상한 확대.
kernel.sched_migration_cost_ns = 5000000
코어 간 태스크 이동 억제.
kernel.hung_task_timeout_secs = 85
hung task 탐지 타임아웃 연장.

[conntrack]
nf_conntrack_max = 800000
최대 연결 추적 수.
nf_conntrack_buckets = 100000
해시 버킷 수.
*nf_conntrack_tcp_timeout_ / udp_timeout / icmp_timeout**
연결 추적 타임아웃 단축으로 자원 회수 가속.
[bootloader] 부트로더 옵션
cmdline = skew_tick=1
CPU 타이머 틱 오프셋을 적용해 인터럽트 피크 분산

최대 부하와 최대 분산 환경 검증

- Computer Cluster(195 nodes) * S3 Storage Cluster(56 nodes)
- 다양한 가설을 정의하여 Test Case 정의

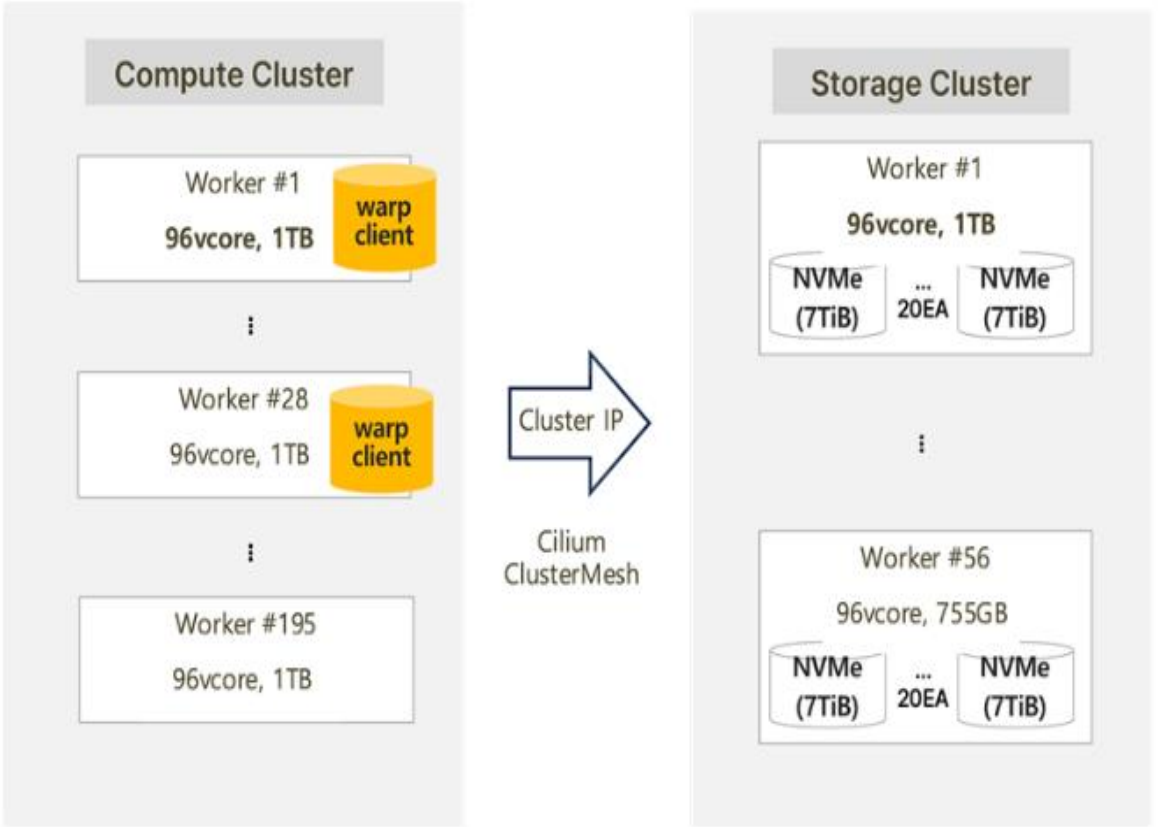
[테스트 시나리오]
개요:
- block size, concurrent 및 warp client 개수를 조정하며 saturation point 및 cpu, memory, network, disk 사용율 확인 (saturation point 확인 시 throughput, latency 등 기준)

[조합]
Block size:
- 큰파일: 4MiB, 24MiB (비교용 hadoop cluster 평균 size), 128MiB(hadoop block size)
- 작은파일: 4KiB, 64KiB, 512KiB → 기존 S3 상위 Top 버킷 평균 사이즈 최소/median/최대값

Warp client 개수:
- 7개 -> 14개 -> 28개 → 100개 -> 195개 -> linear하게 증가하는 수치 확인 및 병목점 분석
- client 7개 및 14개는 큰 파일 한개(24MiB), 작은 파일 한개(64KiB) 만으로 saturation point 확인(28개), 이후 28개 이상으로 고정하여 다른 block size들 확인

Concurrency:
object size 및 warp client 개수를 참고하여 낮은 수치에서 높은 수치로 2배씩 증가 (낮은 수치는 개발환경에서의 데이터를 참고하여 3~4번째 정도에 saturation point가 나올 정도로 정함)

[비고]
- Minio 권고 OS tuned 적용 상태
- TLS 구성 상태임
- warp client마다 2500 object upload
- object storage 사용량: 거의 없음(2TB 이하, 50000 objects 이하)

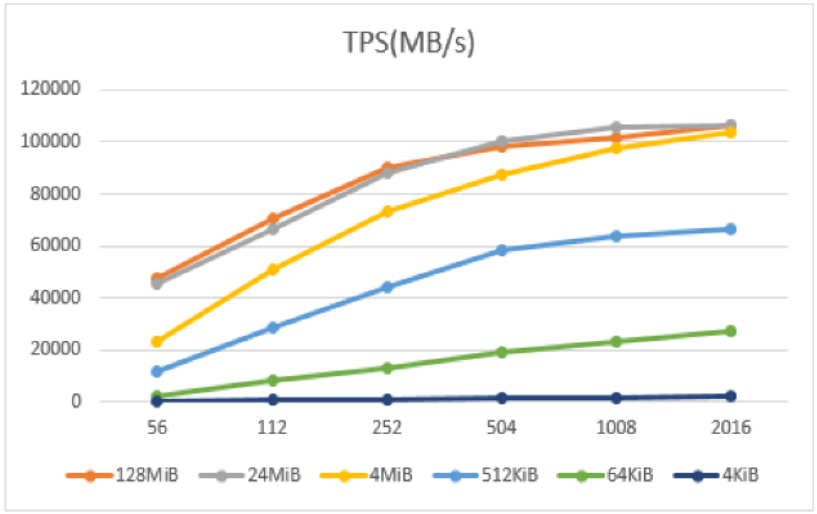


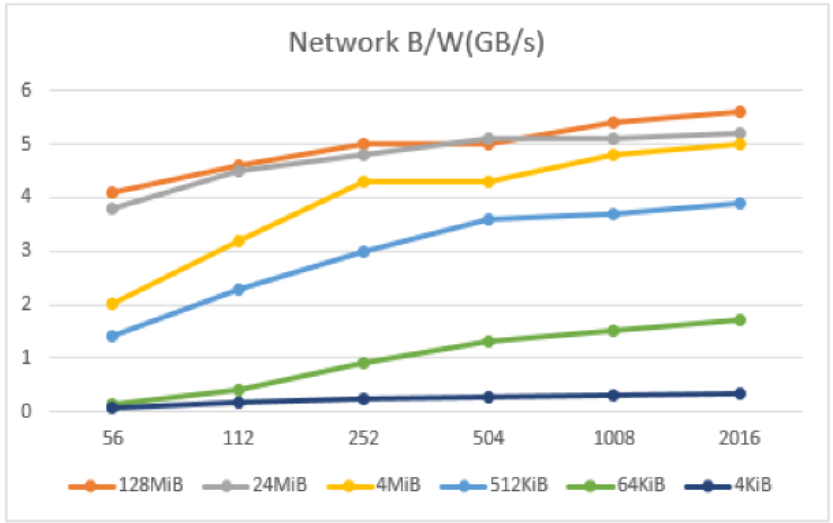
https://github.com/minio/warp

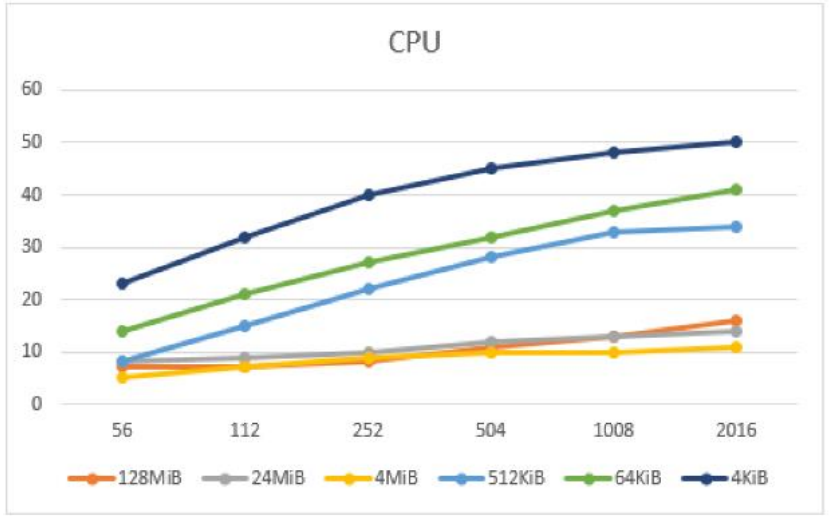
4.5 MinIO(4) – 성능 검증 결과

성능 가설과 예상 결과에 맞는지 분석

- 노드 수가 증가할 경우 Linear하게 성능이 증가하는 것이 MinIO 아키텍처의 기본 가설
- 실제 테스트에서는 일정 구간 이후에 성능 그래프가 증가하지 않음
- 어느 구간에서 문제가 발생하는지 Low Level 영역부터 다시 분석 필요







	THROUGHPUT	IOPS
PUT	81 GiB/s	1,293 objs/s
GET	152 GiB/s	2,424 objs/s

End of Document