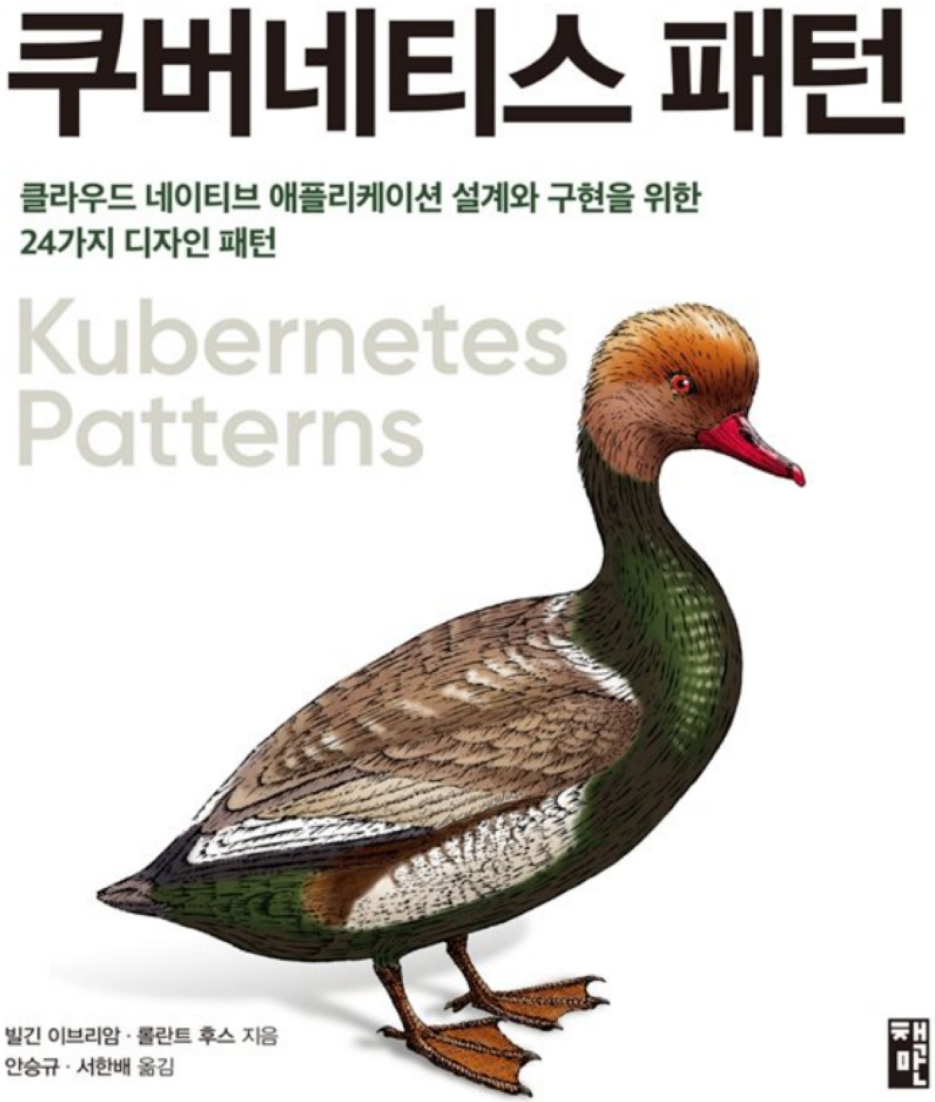
Kubernetes Pattern & Istio

2021. 3

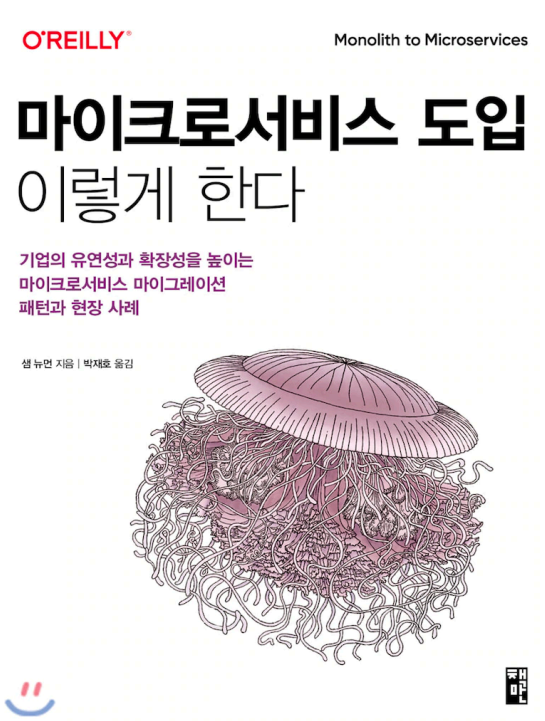
안승규 (seungkyua@gmail.com)

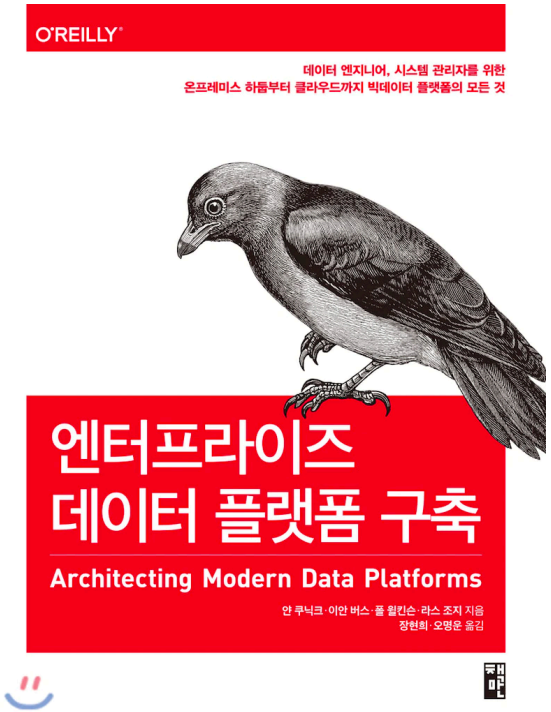
(책 협찬) SKT T-Hub(thub@sk.com), 책만 출판사(support@onlybook.co.kr)



15분 선정하여 DM 으로 연락

thub@sk.com 이름/배송지주소/전화번호





로컬과 분산 기본 요소

개념	로컬 기본 요소	분산 기본 요소
캡슐화 동작	클래스	컨테이너 이미지
인스턴스화 동작	객체	컨테이너
재사용 단위	.jar 파일	컨테이너 이미지
컴포지션(Composition)	클래스 A가 클래스 B를 포함	사이드카 패턴
상속	클래스 A가 클래스 B를 확장	'FROM 부모 이미지'로 만든 컨테이너 이미지
배포 단위	.jar/.war/.ear	파드
빌드타임/런타임 격리	모듈, 패키지, 클래스	네임스페이스, 파드, 컨테이너
초기화 필요 조건	생성자	초기화 컨테이너
초기화 직후 트리거	Init 메소드	`postStart`
삭제 직전 트리거	Destroy 메소드	`preStop`
클린업(Cleanup) 절차	`finalize()`, 셧다운 훅	Defer 컨테이너*
비동기 & 병렬 실행	`ThreadPoolExecutor`, `ForkJoinPool`	잡
주기적 작업	`Timer`, `ScheduledExecutorSerivice`	크론잡
백그라운드 작업	데몬 스레드	데몬세트
설정 관리	`System.getenv()`, `Properties`	컨피그맵, 시크릿

Container

- ✓ 제어 역전 (Inversion of Control, IoC)
- ✓ 컨테이너 이미지는 하나의 문제를 해결하는 기능 단위다.
- ✓ 컨테이너 이미지는 하나의 팀에 의해 소유되며, 릴리스 주기가 있다.
- ✓ 컨테이너 이미지는 자기 완비적이며, 런타임 의존성을 정의하고 수행한다.
- ✓ 컨테이너 이미지는 불변적이며, 한번 만들어지면 변경되지 않는다. 즉 이미 설정 값이 정해져 있다.
- ✓ 컨테이너 이미지는 런타임 의존성과 자원 요구사항이 정의되어 있다.
- ✓ 컨테이너 이미지에는 기능을 노출시키기 위해 잘 정의된 API가 있다.
- ✓ 컨테이너는 전형적으로 하나의 유닉스 프로세스로 실행된다.
- ✓ 컨테이너는 일회용이며 언제든지 스케일 업scale up과 스케일 다운scale down을 안전하게 수 행할 수있다.

- 예측 범위 내의 요구사항 (Predictable Demands)
- 선언적 배포 (Declarative Deployment)
- 정상상태 점검 (Health Probe)
- 수명주기 관리 (Managed Lifecycle)
- 자동 배치 (Automated Placement)

예측 범위 내의 요구 사항

❖ 문 제

의존성 (영구 스토리지 사용, 특정 포트 사용)

자원 요구 사항 (spike 를 치는 서비스)

❖ 해결책

1. 런타임 의존성
   - 스토리지 (emptyDir, PVC), hostPort, ConfigMap, Secret
   - 요청한 모든 컨피그맵이 생성되지 않으면, 컨테이너는 노드에 스케줄링될 수 있지만 시작되지는 않는다.
2. 자원 프로파일
   - 압축 가능 자원 (compressible resource: CPU, Network bandwidth)
   - 압축 불가능 자원 (incompressible resource: 메모리)
   - requests / limits == -Xms -Xmx (Java 언어 실행 시 힙 메모리 사이즈 설정)

예측 범위 내의 요구 사항

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
    resources:
      requests:
        cpu: 100m
        memory: 100Mi
      limits:
        cpu: 200m
        memory: 200Mi
```

예측 범위 내의 요구 사항

➢ 파드 서비스 품질 (Pod QoS)

- ✓ 최선적 (Best-Effort) 파드
  - 요청(requests)와 제한(limits)을 갖고 있지 않음
  - 노드의 압축 불가능 자원이 전부 사용되면 가장 먼저 죽음
- ✓ 확장 가능 (Burstable) 파드
  - 요청과 제한을 갖고 있지만 값이 다름
  - 노드의 압축 불가능 자원이 압박을 받을 때 최선적 파드가 없다면 죽을 확률이 높다
- ✓ 보장 (Guaranteed) 파드
  - 요청과 제한이 동일한 값을 가짐
  - 가장 우선순위가 높음

예측 범위 내의 요구 사항

3. 파드 우선순위

- 파드 우선순위(Priority), 파드 선점(Preemption)
- PriorityClass
  - value 값이 높을 수록 우선순위가 높음
  - priority admission controller 동작
  - 스케줄링에 활용
  - system-cluster-critical, system-node-critical

* 노드에서 파드 축출: 서비스 품질, 파드 우선순위
* 스케줄러가 파드 배치: 파드 우선순위

* PodDisruptionBudget

```
apiVersion: scheduling.k8s.io/v1beta1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000
globalDefault: false
description: This is a very high priority Pod class

---
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
  labels:
    env: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
  priorityClassName: high-priority
```

예측 범위 내의 요구 사항

4. 프로젝트 자원

- 리소스쿼터 (ResourceQuota)
  - CPU, 메모리, 스토리지 전체 사용량
  - 컨피그맵, 시크릿, 파드, 서비스 등의 총 개수
- 리미트레인지 (LimitRange)
  - 각 자원의 종류마다 기본값, 최대 자원량 설정
  - Container, PersistentVolumeClaim 등
- LimitRequestRatio
  - 요청과 제한 사이의 차이 값을 제어

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-min-max-demo-lr
spec:
  limits:
    - max:
        memory: 1Gi
      min:
        memory: 500Mi
      type: Container

---
apiVersion: v1
kind: LimitRange
metadata:
  name: storagelimits
spec:
  limits:
    - type: PersistentVolumeClaim
      max:
        storage: 2Gi
      min:
        storage: 1Gi
```

선언적 배포

❖ 문 제

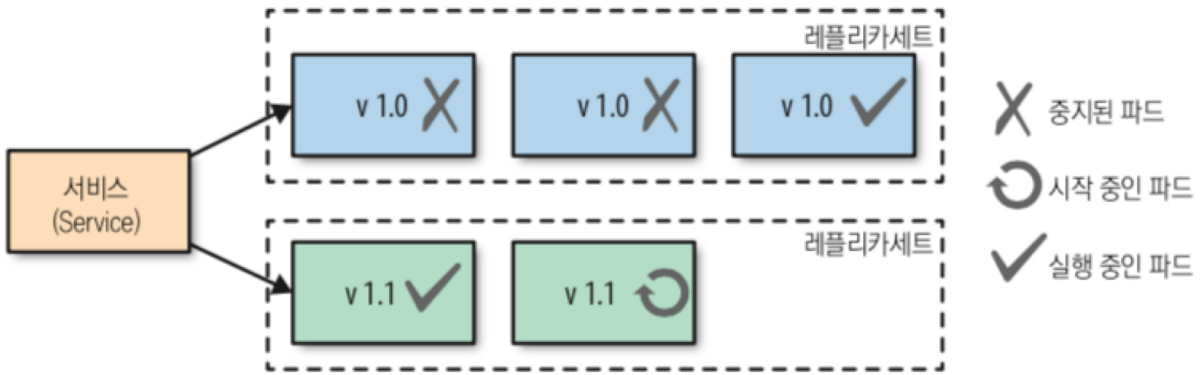
서비스 업그레이드의 복잡함, 다운타임 발생

업그레이드 실패시 롤백

❖ 해결책

1. 롤링 배포

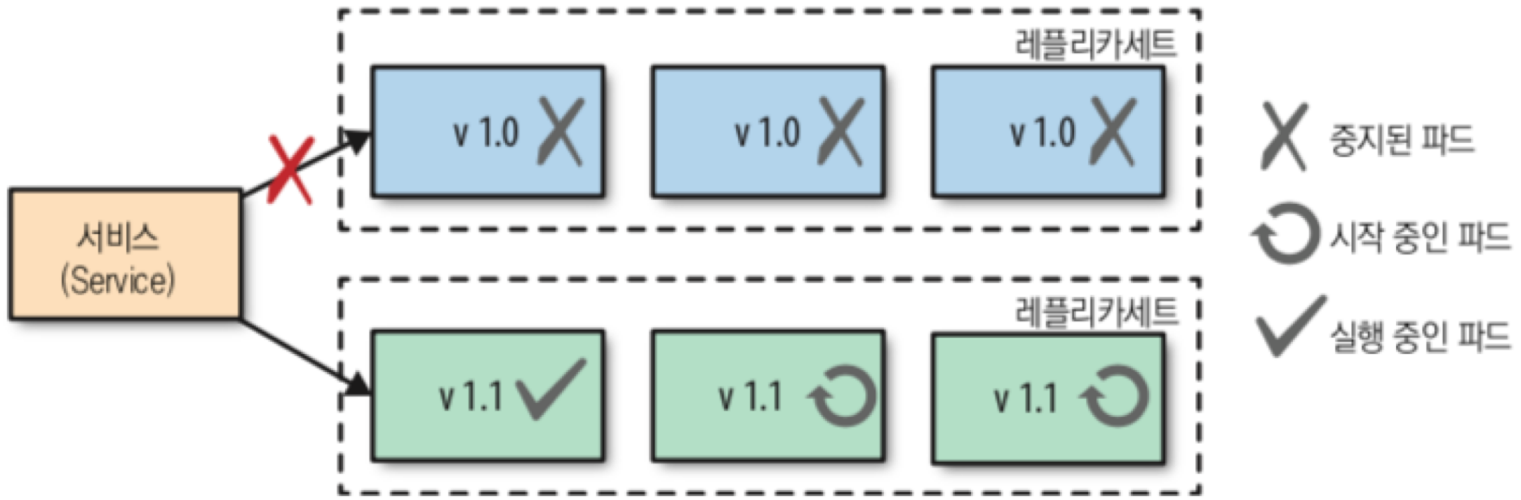
- 두 버전이 동시에 존재
- 레이블 셀렉터를 지원하는 ReplicaSet 활용
- RollingUpdate : maxSurge, maxUnavailable



선언적 배포

2. 고정 배포 (Recreate)

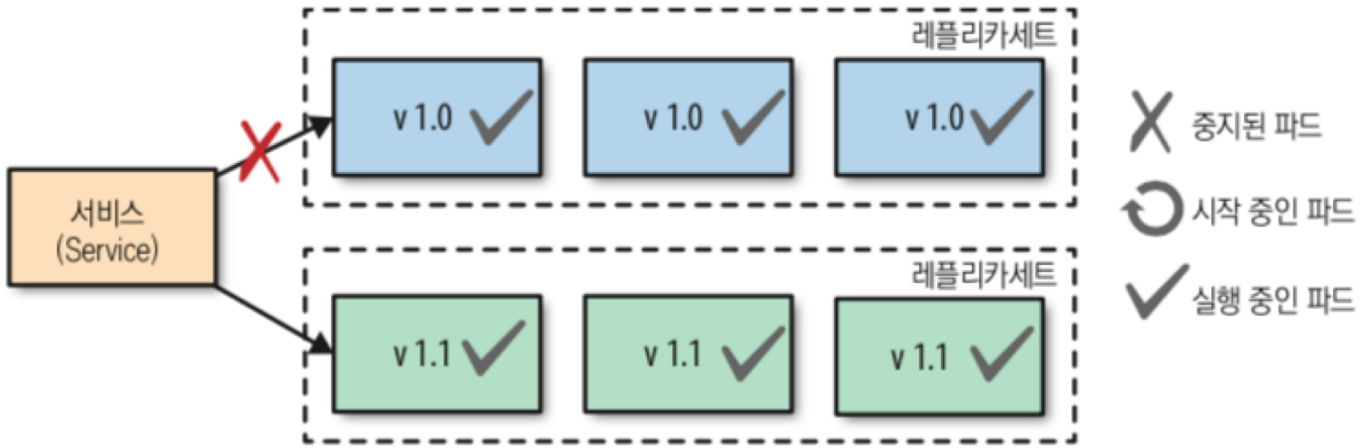
- 이전 버전과 호환되지 않음
- 서비스 중단 발생



선언적 배포

3. 블루-그린 배포 (Blue-Green deployment)

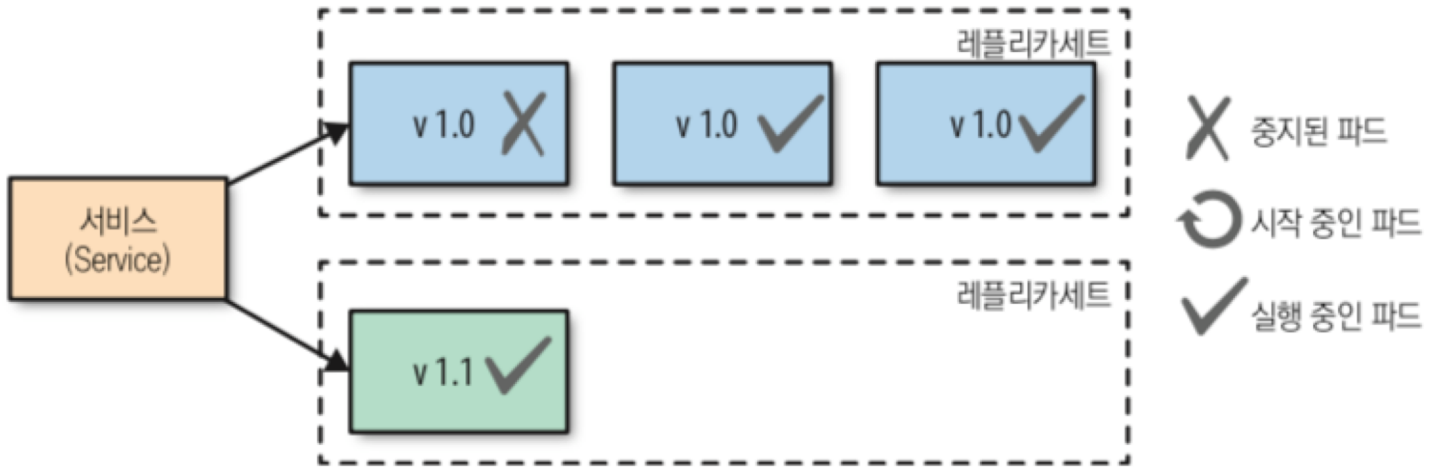
- Service 에서 Selector 를 활용
- 블루와 그린 커테이너가 모두 실행되는 동안 애플리케이션 용량은 2배로 필요



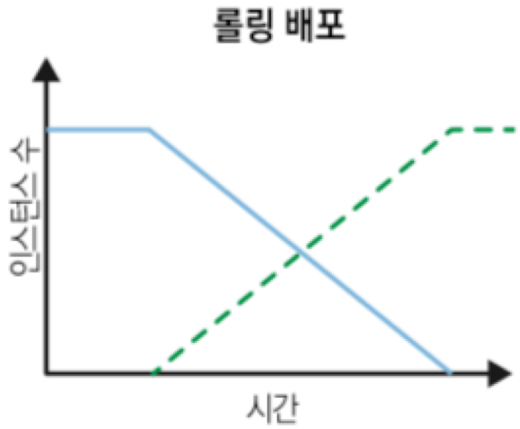
선언적 배포

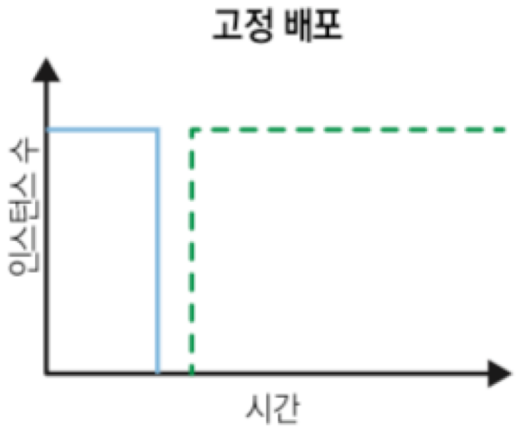
4. 카나리아 릴리스 (Canary release)

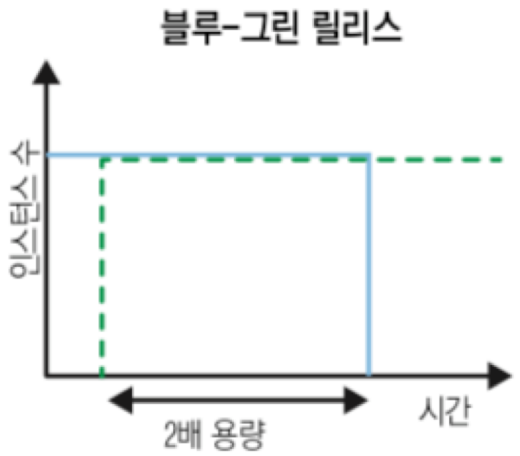
- 이전 인스턴스의 작은 하위집합만 새로운 인스턴스로 교체
- 운영에 새 버전을 도입할 때 위험을 줄여주

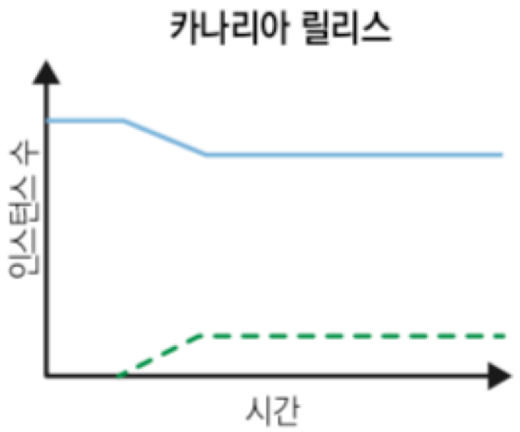


선언적 배포


롤링 배포
인스턴스 수
시간


고정 배포
인스턴스 수
시간


블루-그린 릴리스
인스턴스 수
시간
2배 용량


카나리아 릴리스
인스턴스 수
시간

정상상태 점검

❖ 문 제

프로세스 상태 확인으로는 애플리케이션 정상상태를 알 수 없다. (hang 발생 경우)

애플리케이션이 예상대로 동작하는지를 알 수 있어야 한다.

❖ 해결책

1. 프로세스 정상상태 확인
   - Kubelet 이 컨테이너 프로세스에 대해 모니터링
2. 라이브니스 점검 (Liveness probe)
   - HTTP GET 요청 (200 ~ 399 응답 코드면 정상)
   - TCP 소켓 연결 (소켓 연결에 성공하면 정상)
   - Exec 실행 (종료코드가 0 이면 정상)

   * 실패하면 container 를 restart

정상상태 점검

3. 레디니스 점검 (Readiness probe)

- HTTP GET 요청 (200 ~ 399 응답 코드면 정상)
- TCP 소켓 연결 (소켓 연결에 성공하면 정상)
- Exec 실행 (종료코드가 0 이면 정상)

* 실패하면 container 를 Service Endpoint 에서 제거

정상상태 점검

```
livenessProbe:
  httpGet:
    path: /actuator/health
    port: 8080
  initialDelaySeconds: 30
```

- HTTP로 localhost:8080/actuator/health
- Liveness 수행하기 전 30초를 기다림

```
readinessProbe:
  exec:
    command: [“stat”, “/var/run/ready”]
```

- 어플리케이션이 서비스를 요청을 처리할 준비가 되었는지 확인할 수 있는 파일의 존재여부 확인
- stat 은 파일이 존재하지 않으면 에러 반환

수명주기 관리

❖ 문 제

플랫폼이 애플리케이션을 관리하기 위해 명령어를 보낼 수 있어야 한다.

애플리케이션은 이벤트를 선별하여 반응할 수 있어야 한다.

❖ 해결책

* Pod 레벨이 아닌 개별 Container 레벨에 적용

1. 시그텀 신호 (SIGTERM)
   - 쿠버네티스가 컨테이너를 멈추기로 결정했을 때 컨테이너에 보내는 신호 (e.g., 라이브니스 점검 실패)
2. 시그킬 신호 (SIGKILL)
   - 시그텀으로 종료되지 않으면 시그킬을 보냄
   - 기다리는 시간 default 30초 : .spec.terminationGracePeriodSeconds 필드 값으로 정의

수명주기 관리

3. postStart
   - postStart 는 컨테이너의 ENTRYPOINT와 동시 실행, 프로세스는 누가 먼저 실행될지 모름
   - postStart 가 완료될 때 까지 Pod 는 Pending, Container 는 waiting 상태 값
   - postStart 가 exec 로 0 이 아닌 종료 코드를 리턴하면 컨테이너 프로세스는 쿠버네티스에 의해 죽게됨
   - 중복 실행 가능성도 있음, 최소 한 번 실행 의미로 설계
   - 실패 시 재시도 하지 않음
4. preStop
   - 컨테이너가 종료되기 전에 컨테이너로 전송되는 블록킹 호출
   - 시그텀과 동일하기 때문에 컨테이너에서 시그텀 응답이 불가능할 때 사용 가능
   - 어떤 결과가 나오더라도 컨테이너가 삭제되거나 프로세스가 종료되는 상황은 막을 수 없음
5. Init Container
   - Pod 레벨에 적용
   - Init Container 완료 후 main Container 가 실행됨

자동 배치

❖ 문 제

마이크로서비스 기반 시스템은 서비스를 효율적으로 분배하여 배치하기가 어렵움

컨테이너를 배치하는 방법은 가용성, 성능뿐만 아니라 분산 시스템의 용량에더 영향을 줌

❖ 해결책

* 스케줄러를 통해 파드를 배치한다.
* 스케줄러는 API 서버로 부터 파드 정의 조회, 파드 노드를 할당
* 초기 파드 배치, 스케일 업, 비정상 노드에서 정상노드로 이동 등에 관여

1. 가용한 노드 자원
   - Allocatable = Node Capacity - Kube-Reserved - System-Reserved [sshd, udev OS 시스템 데몬]
   - 개별 컨테이너가 있으면 자원 요청만 있는 플레이스홀더 파드(placeholder pod)를 생성하여 자원 관리

자동 배치

2. 컨테이너 자원 요구
   - request / limit
3. 배치 (Placement) 정책
   - policy 적용
   - custom scheduler 생성

     .spec.schedulerName 으로 지정

예제 6-2 스케줄러 정책 예제

```
{
    "kind" : "Policy",
    "apiVersion" : "v1",
    "predicates" : [                                    ❶
        {"name" : "PodFitsHostPorts"},
        {"name" : "PodFitsResources"},
        {"name" : "NoDiskConflict"},
        {"name" : "NoVolumeZoneConflict"},
        {"name" : "MatchNodeSelector"},
        {"name" : "HostName"}
    ],
    "priorities" : [                                    ❷
        {"name" : "LeastRequestedPriority", "weight" : 2},
        {"name" : "BalancedResourceAllocation", "weight" : 1},
        {"name" : "ServiceSpreadingPriority", "weight" : 2},
        {"name" : "EqualPriority", "weight" : 1}
    ]
}
```

❶ 술어는 자격 없는 노드를 필터링하여 제외하는 규칙이다. 예를 들어 PodFitsHostsPorts는 이 포트 사용이 가능한 노드에만 특정 고정된 호스트 포트를 요청하도록 파드를 스케줄링한다.

❷ 우선순위는 설정에 따라 사용 가능한 노드를 정렬하는 규칙이다. 예를 들어 LeastRequestedPriority는 요청된 자원이 적은 노드에 더 높은 우선순위를 부여한다.

자동 배치

4. nodeSelector
   - 사용자 정의 Label -> disktype: ssd
   - 기본 Label -> kubernetes.io/hostname
5. 노드 어피니티
   - nodeSelector 를 일반화한 접근 방법
   - required..., preferred...
   - matchExpressions, matchFields

     operator: In, NotIn, Exists, DoesNotExist, Gt, Lt

예제 6-4 노드 어피니티를 가진 파드

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:          ❶
        nodeSelectorTerms:
        - matchExpressions:                                    ❷
          - key: numberCores
            operator: Gt
            values: [ "3" ]
      preferredDuringSchedulingIgnoredDuringExecution:         ❸
      - weight: 1
        preference:
          matchFields:                                         ❹
          - key: metadata.name
            operator: NotIn
            values: [ "master" ]
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
```

3개 초과

❶ 스케줄링 프로세스에서 고려해야 할 노드에 3개 이상의 코어(노드 레이블에 의해 표시된)가 있어야 한다는 엄격한 요구사항이다. 노드의 조건이 변경되어도 스케줄링 동안에 이 규칙이 재적용되지는 않는다.

❷ 레이블과 일치한다.

❸ 유연한 요구사항으로, 가중치를 갖는 셀렉터 목록이다. 모든 노드에 대해 일치하는 셀렉터들의 모든 가중치 합계를 계산한다. 엄격한 요구사항과 일치하는 한, 값이 가장 큰 노드가 선택된다.

❹ 필드(jsonpath로 지정된)와 일치한다. 연산자로 In과 NotIn만 사용 가능하고, 값 목록에는 하나의 값만 허용된다.

자동 배치

6. 파드 어피니티와 파드 안티어피니티

- 파드 분산 및 파드 묶음 배치
- required..., preferred...

예제 6-5 파드 어피니티를 갖는 파드

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:        ❶
      - labelSelector:                                       ❷
          matchLabels:
            confidential: high
        topologyKey: security-zone                           ❸
    podAntiAffinity:                                         ❹
      preferredDuringSchedulingIgnoredDuringExecution:       ❺
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchLabels:
              confidential: none
          topologyKey: kubernetes.io/hostname
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
```

❶ 대상 노드에서 실행 중인 또다른 파드와 관련된 파드 배치에 대한 필수 규칙이다.

❷ 함께 배치할 파드를 찾기 위한 레이블 셀렉터다.

❸ confidential=high 레이블을 가진 파드가 실행 중인 노드는 security-zone 레이블이 있어야 한다. 여기에 정의된 파드는 동일한 레이블과 값을 갖는 노드에 스케줄링된다.

❹ 파드가 배치되어서는 안 되는 노드를 찾기 위한 안티어피니티[5] 규칙이다.

❺ 이 파드가 confidential=none 레이블을 가진 파드가 실행중인 노드에 배치되어서는 안 된다고(그러나 배치될 수도 있다) 설명하는 규칙

7. 테인트와 톨러레이션

- 스케줄링에는 사용할 수 없는 노드에 스케줄링 허용 : 옵트인 (opt-in)
- effect: NoSchedule, PreferNoSchedule, NoExecute

예제 6-6 테인트된 노드

```
apiVersion: v1
kind: Node
metadata:
  name: master
spec:
  taints:                                    ❶
  - effect: NoSchedule
    key: node-role.kubernetes.io/master
```

❶ 파드가 이 테인트를 톨러레이션한 경우를 제외하고는 해당 노드에 스케줄링이 불가능하다고 표시하기 위해 노드의 명세에 테인트를 지정한다.

예제 6-7 테인트 노드에 대한 파드 톨러레이션

```
apiVersion: v1
kind: Pod
metadata:
  name: random-generator
spec:
  containers:
  - image: k8spatterns/random-generator:1.0
    name: random-generator
  tolerations:
  - key: node-role.kubernetes.io/master      ❶
    operator: Exists
    effect: NoSchedule                       ❷
```

❶ node-role.kubernetes.io/master 키로 테인트된 노드를 톨러레이션(즉 스케줄링을 고려하여)한다. 운영 클러스터에서는 마스터로 파드가 스케줄링되는 것을 막기 위해 마스터 노드에 테인트를 설정한다. 하지만 톨러레이션은 이 파드가 마스터로 스케줄링될 수 있게 허용한다.

❷ 테인트가 NoSchedule 효과를 지정할 경우에만 톨러레이션이 적용된다. 이 필드는 비워둘 수 있으며, 이 경우 톨러레이션은 모든 효과에 적용된다.

- 주기적 잡 (Periodic Job)
- 데몬 서비스 (Daemon Service)
- 스테이트풀 서비스 (Stateful Service)
- 서비스 디스커버리 (Service Discovery)
- 자기 인식 (Self-awareness)

주기적 잡

❖ 문 제

주기적인 잡은 스케줄링이 회복성과 고가용성을 갖춰야 하는데 이를 위해 많은 리소스가 필요

시간 기반의 잡 스케줄러는 애플리케이션의 일부이므로 전체 애플리케이션의 가용성이 높아야 함

❖ 해결책

* kind: CronJob
* .spec.schedule : [Minute] [Hour] [Day_of_the_Month] [Month_of_the_Year] [Day_of_the_Week]
  3분마다 "*/3 * * * *", 매시간 "0 * * * *"
* .spec.concurrencyPolicy: Allow(이전 잡이 돌고 있어서 새로운 잡 생성), Forbid, Replace
* .spec.successfulJobHistoryLimit: 완료한 잡의 히스토리를 몇 개까지 유지
* .spec.failedJobHistoryLimit: 실패한 잡의 히스토리를 몇 개까지 유지

데몬 서비스

❖ 문 제

애플리케이션 레벨의 데몬 서비스, 예를 들어 JVM 데몬 스레드는 백그라운드로 실행되고, 우선순위가 낮으며, 애플리케이션 수명에 관여하지 않고 가비지 컬렉션 등을 수행한다

❖ 해결책

* kind: DaemonSet
* nodeSelector 로 필요한 모든 노드에 pod 를 실행
* 스케줄러가 사용되지 않기 때문에 노드의 unschedulable 필드는 데몬세트와 관련이 없음
* 지정된 노드 혹은 지정된 노드 그룹에 파드를 하나씩 생성

스테이트풀 서비스

❖ 문 제

확장성이 뛰어난 모든 스테이트리스 서비스 뒤에는 일반적 데이터 저장소 형태의 스테이트풀 서비스가 있다. 다수의 클러스터화된 스테이트풀 애플리케이션은 스토리지, 네트워킹, 식별자, 순서성 등을 요구

❖ 해결책

* kind: StatefulSet
* 효율적 관리를 위해 CustomResourceDefinition 과 Operator 를 사용

스테이트풀 서비스

1. 스토리지

   - volumeClaimTemplates

```
volumeClaimTemplates:
  - metadata:
      name: logs
    spec:
      accessModes: [“ReadWriterOnce” ]
      resources:
        requests:
          storage: 10Gi
```

스테이트풀 서비스

2. 네트워킹

- headless service

- mariadb-0 pod 에 접근할 수 있는 서비스 명

mariadb-0.mariadb.default.svc.cluster.local

- 최대한 하나 (at-most-one) 보장

노드에 문제가 생겼을 때 파드가 종료되었음을 확인할 수 없다면 다른 노드에 새로운 파드를 스케줄하지 않는다

아래 명령으로 새로운 pod 생성 가능

kubectl delete pod _<pod>_ --grace-period=0 --force

```
metadata:
  name: mariadb
spec:
  clusterIP: None
  selector:
    app: db
  ports:
  - name: tcp
    port: 3360
```

서비스 디스커버리

❖ 문 제

동적인 쿠버네티스 파드의 클러스터 내외부 endpoint를 추적, 등록, 디스커버리하는 것은 굉장히 어려운 일이다.

❖ 해결책

1. 내부 서비스 디스커버리

- Service 를 통한 방법 (Selector 를 이용)

. 환경 변수 (XXX_SERVICE_HOST=10.109.72.32, SERVICE_PORT=80)

. DNS 참조를 통한 디스커버리 (mariadb.default.svc.cluster.local)

. 레디니스 점검 (pod 의 readiness 가 false 이면 endpoint 목록에서 제거)

서비스 디스커버리

2. 수동 서비스 디스커버리

- 외부 서비스 연결시에 selector 를 제외
- type: ExernalName 으로 설정 가능

```
kind: Service
spec:
  type: ExeternalName
  externalName: my.database.example.com
  ports:
  - protocal: TCP
    port: 80
```

```
kind: Service
spec:
  type: ClusterIP
  ports:
  - protocal: TCP
    port: 80
---
kind: Endpoints
Subsets:
  - address:
    - ip: 1.1.1.1
    - ip: 2.2.2.2
    ports:
    - port: 8080
```

자기 인식

❖ 문 제

애플리케이션 자체에 대한 정보와 실행 중인 환경에 대한 정보가 필요할 수 있다

파드 이름, 파드 IP주소, 애플리케이션이 배치된 호스트 이름처럼 런타임 시에만 알 수 있는 정보

❖ 해결책

1. Downward API 사용

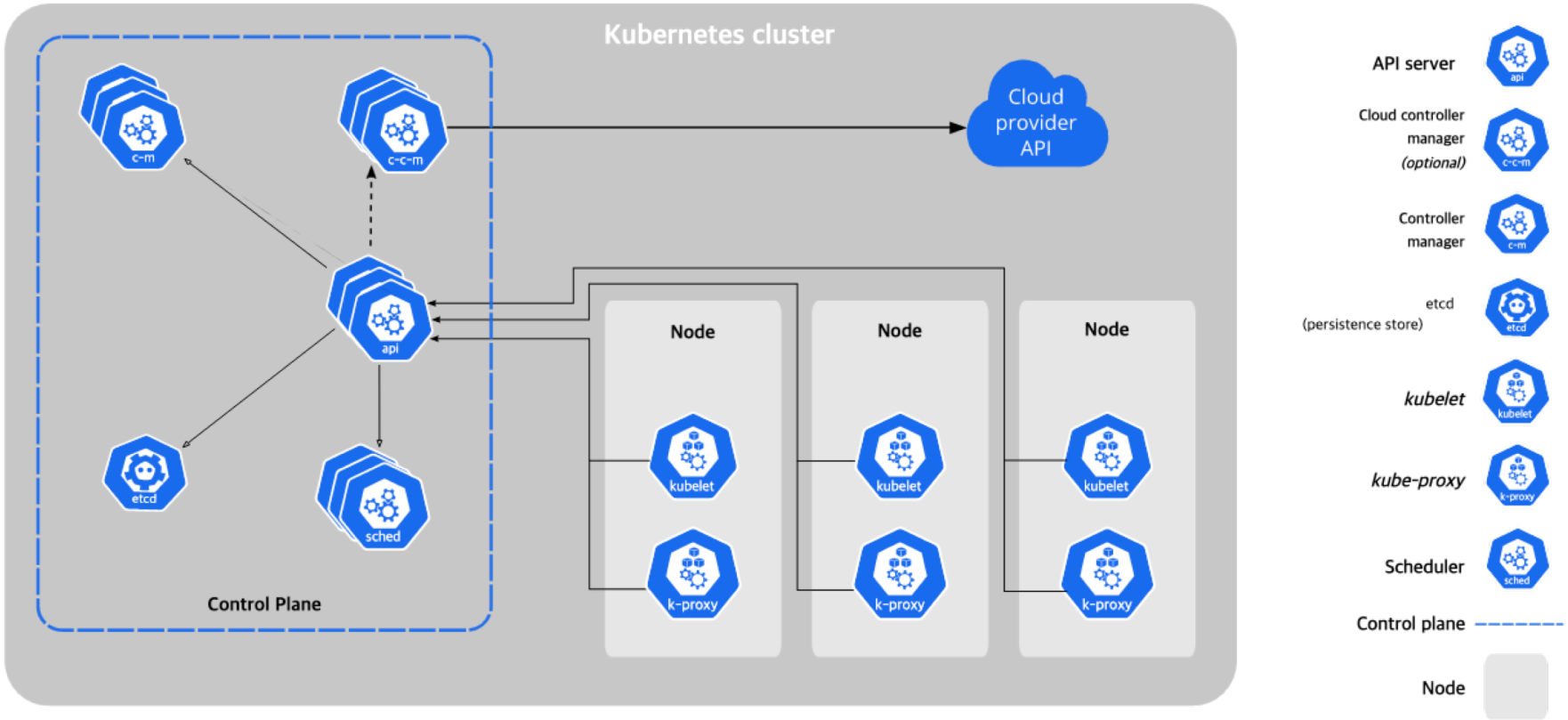
```
env:
- name: POD_IP
  valueFrom:
    fieldRef:
      fieldPath: status.podIP
- name: MEMORY_LIMIT
  valueFrom:
    resourceFieldRef:
      containerName: mariadb
      resource: limits.memory
```

자기 인식

* fieldRef.fieldPath
  - spec.nodeName, status.hostIP, metadata.name, metadata,namespace, status.podIP
  - spec.serviceAccountName, metadata.uid, metadata.labels['key'], metadata.annotations['key']

* resourceFieldRef.resource
  - request.cpu, limits.cpu, requests.memory, limits.memory

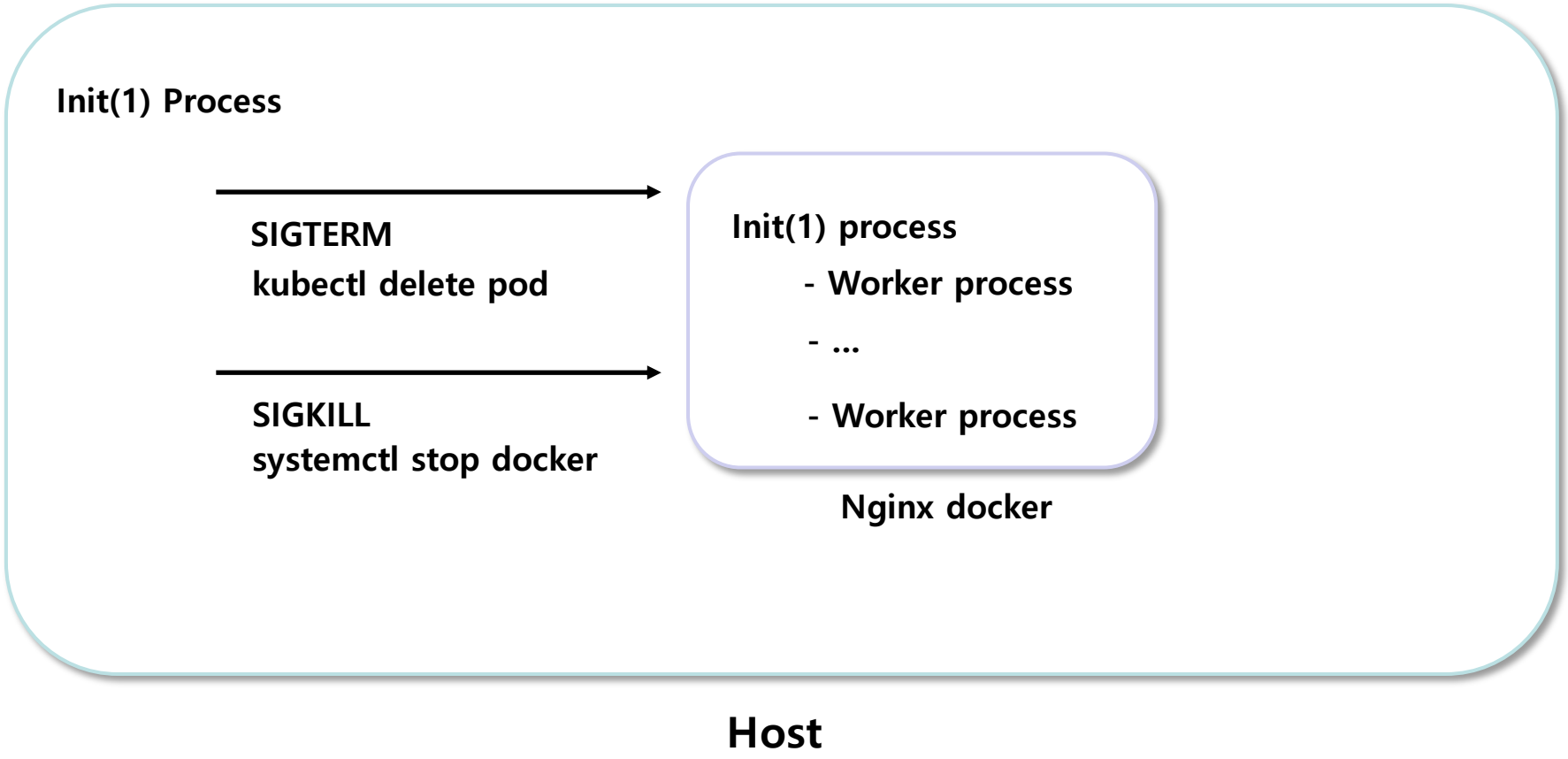
kubectl 명령이 느려질 때



https://kubernetes.io/docs/concepts/overview/components/

1. etcd resource 부족 (cpu, memory disk, network etc. -> too many leader election)
2. API-Server 처리 능력 부족
3. cotroller manager & scheduler tls 이슈 (too many restart)

Docker Orphan process



system paremeter

1. ipatables forward (service load balance, port forward)

- net.ipv4.ip_forward=1

2. inode watch error

- file 서버와 같이 활용 될 때
- node 가 Ready 와 Not Ready 를 반복
- syslog
- lsof -K | grep inotify | (less||more||pg)
- fs.inotify.max_user_watches=1048576

kubelet cpu pinning

- **kubelet.env**

  --runtime-cgroups=/kube_whitelist --kubelet-cgroups=/kube_whitelist

- **kubectl.service**

  ExecStartPre=-/usr/local/sbin/kube-cgroup.sh

- **kube-cgroup.sh**

  "/sys/fs/cgroup/system/kube_whitelist"

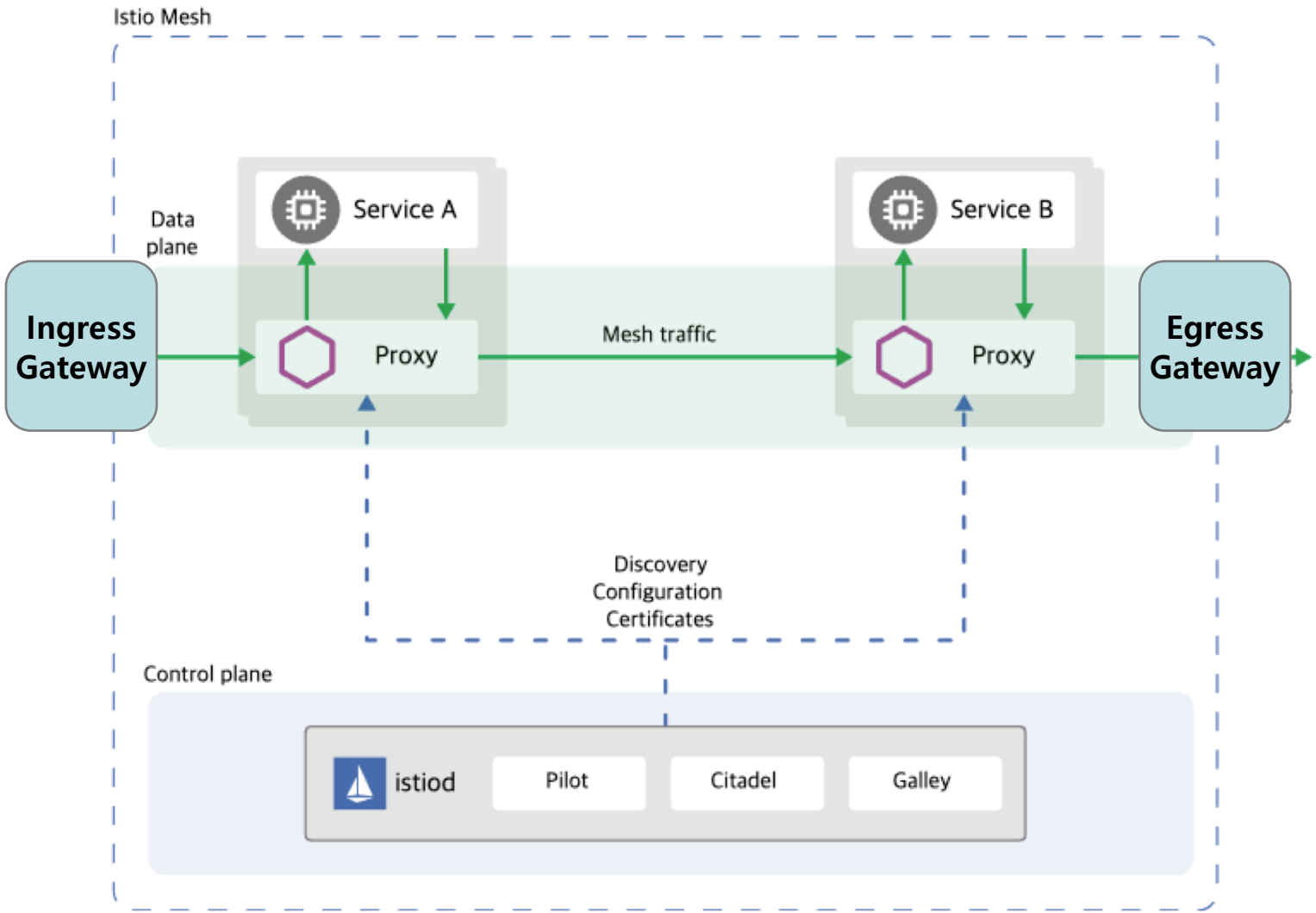
  "/sys/fs/cgroup/cpuset/kube_whitelist"

  "/sys/fs/cgroup/cpu/kube_whitelist"

  "/sys/fs/cgroup/memory/kube_whitelist"

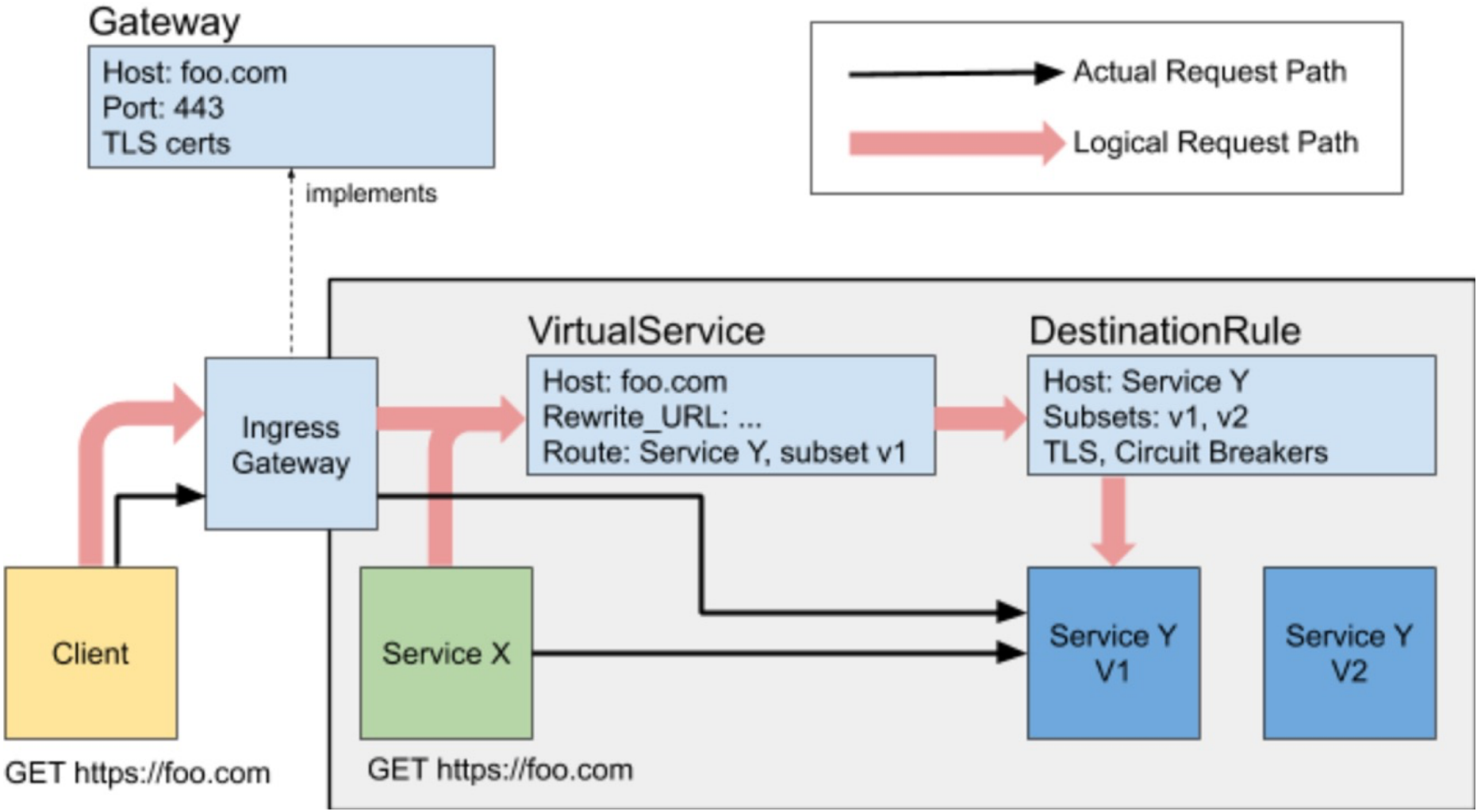
  "/sys/fs/cgroup/cpuset/kube_whitelist/cpuset.cpus"

Istio Architecture



Istio Architecture

Istio Traffic Management



Book: Istio up and running

Istio 설치

$ kubectl label ns default **istio.io/rev=1-9-1**

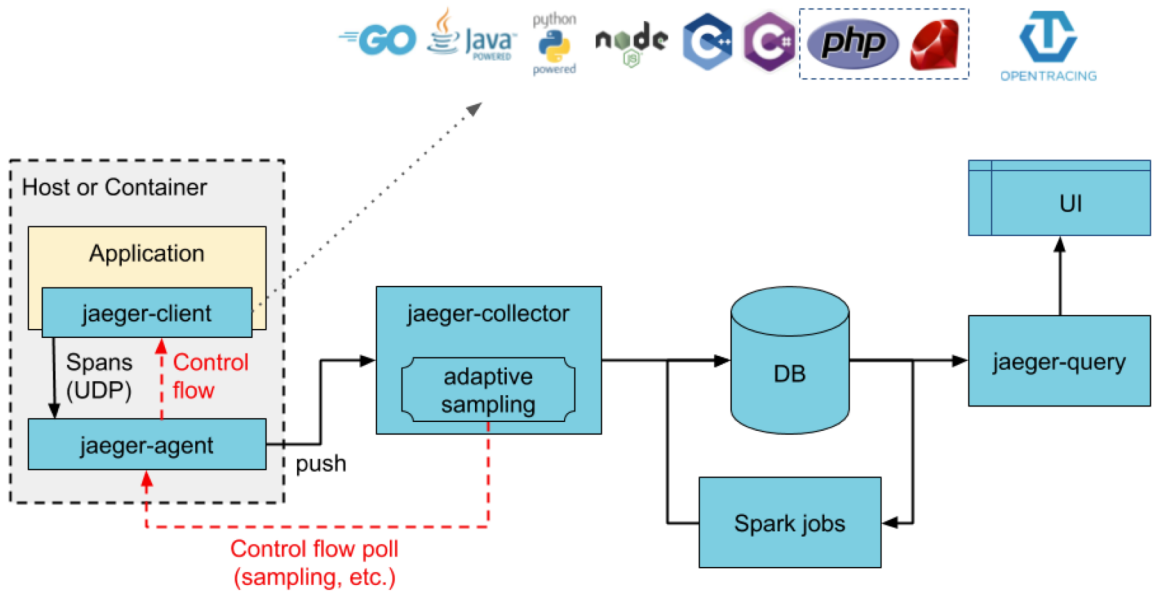

```
revision: "1-9-1"
operatorNamespace: istio-operator
watchedNamespaces: istio-system
operator:
  resources:
    limits:
      cpu: 4
      memory: 4Gi
    requests:
      cpu: 2
      memory: 2Gi
```

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
...
spec:
  revision: "1-9-1"
  profile: default
  meshConfig:
    accessLogFile: /dev/stdout
    enableTracing: true
    enablePrometheusMerge: true
    defaultConfig:
      discoveryAddress: istiod-1-9-1.istio-system.svc:15012
      tracing:
        zipkin:
          address: jaeger-operator-jaeger-collector.istio-system:9411
        sampling: 100.0
  components:
    ...
```

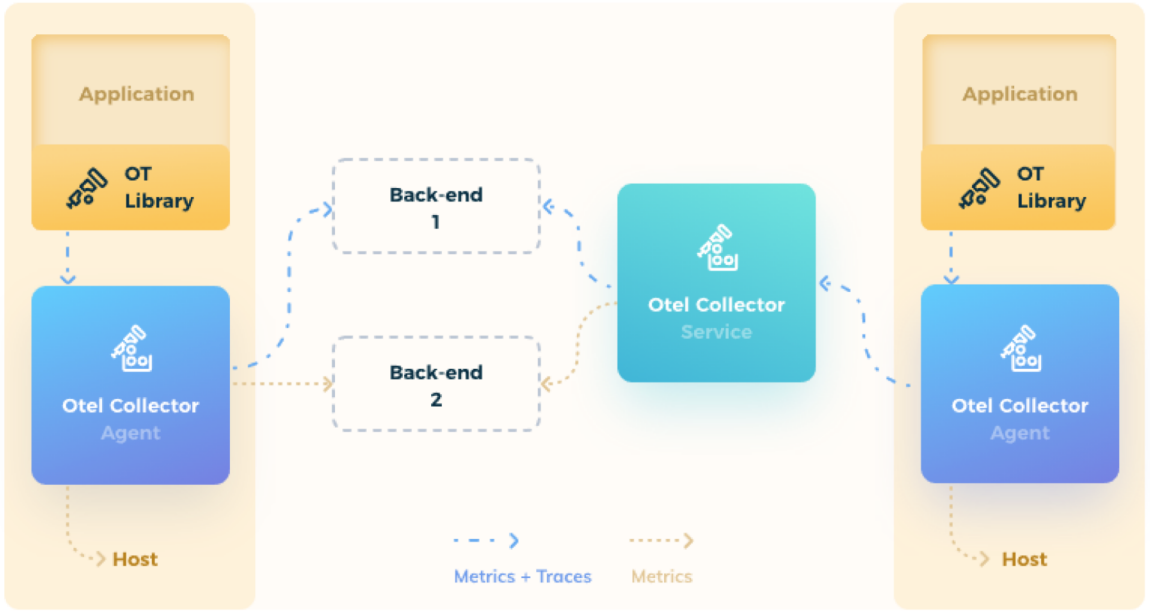
```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
...
spec:
  revision: "1-9-1"
  profile: empty
  components:
    ingressGateways:
    - enabled: true
```

Jaeger Architecture

OpenTracing + OpenCensus = OpenTelemetry

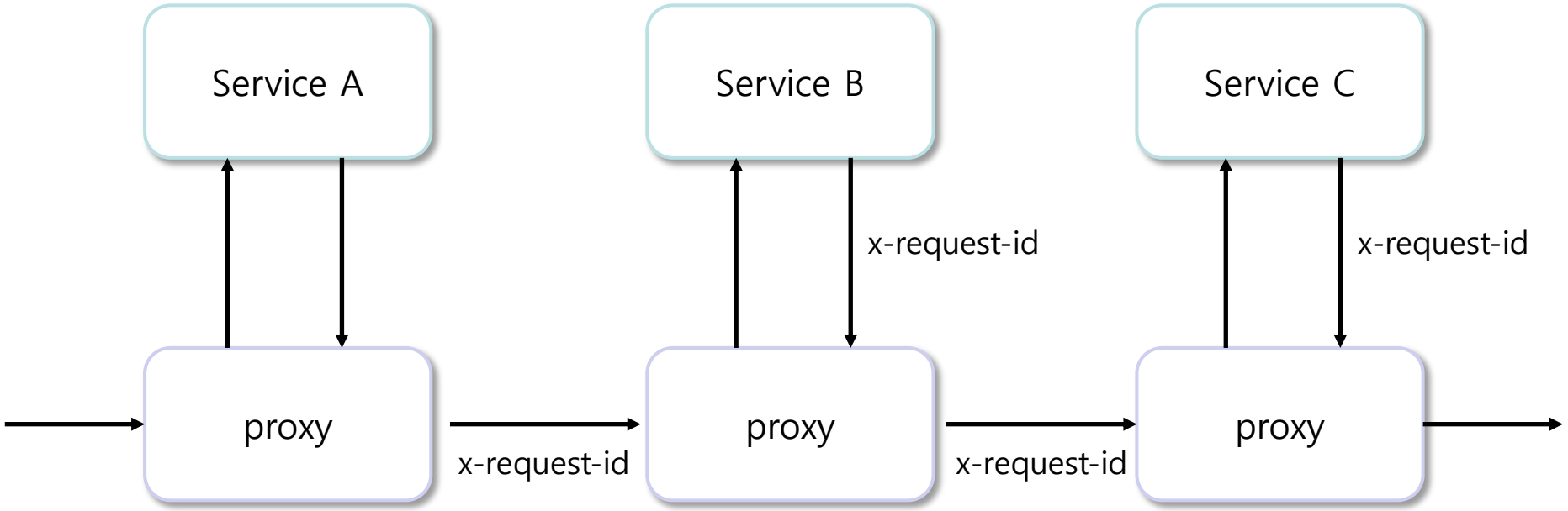


https://www.jaegertracing.io/img/architecture-v1.png



https://raw.github.com/open-telemetry/opentelemetry.io/main/iconography/Reference_Architecture.svg

Trace context propagation



[HTTP Header]

- x-request-id
- x-b3-traceid
- x-b3-spanid
- x-b3-parentspanid
- x-b3-sampled
- x-b3-flags
- x-ot-span-context