「Kubernetes による Cloud Native な開発」と「VM 時代の開発」の違い

JEITA ソフトウェアエンジニアリング技術ワークショップ 2019

Masaya Aoyama

CyberAgent

Masaya Aoyama (@amsy810)

Infrastructure Engineer

+ CREATIONLINE / DENSO - 技術アドバイザ
+ SAKURA Internet Research Center – 客員研究員

主業務:
K8s as a Service の実装
K8s / CloudNative 関連のアーキテクト

Publicity （一部抜粋）

書籍 『Kubernetes 完全ガイド』 『みんなの Docker/K8s』

基調講演 『Japan Container Days v18.04』 『Google Cloud K8s Day』

招待講演 『情報処理学会 コンピュータシステムシンポジウム』
『AWS Dev Day Tokyo』 『IBM Think Japan』 『JEITA委員会』

登壇 『KubeCon + CNCon China 2019』 『Open Source Summit 2019』

資格 『CKAD #2』 『CKA #138』

Community

Co-chair 『Cloud Native Days Tokyo 2019 （旧 Japan Container Days）』

Organizer 『Cloud Native Meetup Tokyo』
『Kubernetes Meetup Tokyo』
『KubeCon Japanese exchange meeting』

Contribute to OpenStack and Kubernetes

Kubernetes に対する興味は日本でも高い

North America 2019

INITIAL DEMOGRAPHIC DATA

SAN DIEGO

11,981

ATTENDEES

65% 1ST-TIME ATTENDEES

153

MEDIA + ANALYSTS

33

CO-LOCATED EVENTS

2,631

COMPANIES

83%

MEN

11%

WOMEN

0.4%

NON-BINARY/
OTHER GENDERS

DID NOT ANSWER - 5.6%

TOP 3 COUNTRIES REPRESENTED

USA · CANADA · JAPAN

TOP THREE PROJECT INTERESTS

KUBERNETES

PROMETHEUS

HELM

TOP THREE JOB FUNCTIONS

DEVELOPER, IT OPERATIONS, SALES/MARKETING

1,801 CFP SUBMISSIONS

209 BREAKOUT + LIGHTNING TALK SESSIONS

14 KEYNOTES

124 MAINTAINER TRACK SESSIONS

CFP ACCEPTANCE RATE: 12%

WOMEN + NON-BINARY/OTHER GENDERS SPEAKERS: 16%

WOMEN + NON-BINARY/OTHER GENDERS KEYNOTE SPEAKERS: 42%

Sponsored by CNCF / The Linux Foundation JP / Fujitsu / CyberAgent

本日は Kubernetes にすると一昔前の VM での開発と比べてどの辺りが異なるのかという軸で入門セッションをします（いい話がメイン）

なお、本日紹介する話の対比は一例です
* VM でもこうすればできる…
* VM でもまだこうなっていない…　はあると思います。

イメージとしては、下記の対比で紹介します
* 一昔前の VM 時代の開発
* 現在の開発

そもそも Cloud Native とは？

Cloud Nativeとは？

CNCF Cloud Native Defenition v1.0, CNCF, 2018-11-28
(https://github.com/cncf/toc/blob/master/DEFINITION.md)

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

The Cloud Native Computing Foundation seeks to drive adoption of this paradigm by fostering and sustaining an ecosystem of open source, vendor-neutral projects. We democratize state-of-the-art patterns to make these innovations accessible for everyone.

Cloud Nativeとは？

CNCF Cloud Native Defenition v1.0, CNCF, 2018-11-28
(https://github.com/cncf/toc/blob/master/DEFINITION.md)

Cloud native te ... calable
applications in ... e, and hybrid
clouds. Contai ... tructure, and
declarative AP

These techniq ... manageable,
and observabl ... eers to make
high-impact ch

The Cloud Nat ... his paradigm by
fostering and sustaining an ecosystem o ... pen source, vendor-neutral projects. We
democratize state-of-the-art patterns to ... e these innovations accessible for
everyone.

- 疎結合なシステム
- 復元力がある
- 管理しやすい
- 可観測である
- 堅牢な自動化により、頻繁かつ期待通りに最小限の労力で大きな変更が可能

↓

OpenかつScalableなシステムを実現

Cloud Native は Kubernetes？

誤解されることがあるので改めて伝えますが、「違います」

Kubernetes != Cloud Native

- Kubernetes を使うだけで Cloud Native になるわけではない
  - もちろんレール（お作法）に乗ることで近づきはします
- VM でも Cloud Native な開発をすることも可能です
- その他のマネージドサービスを利用した Cloud Native な開発も可能です
  - CloudRun、Cloud Functions
  - ECS、Fargate、Lambda

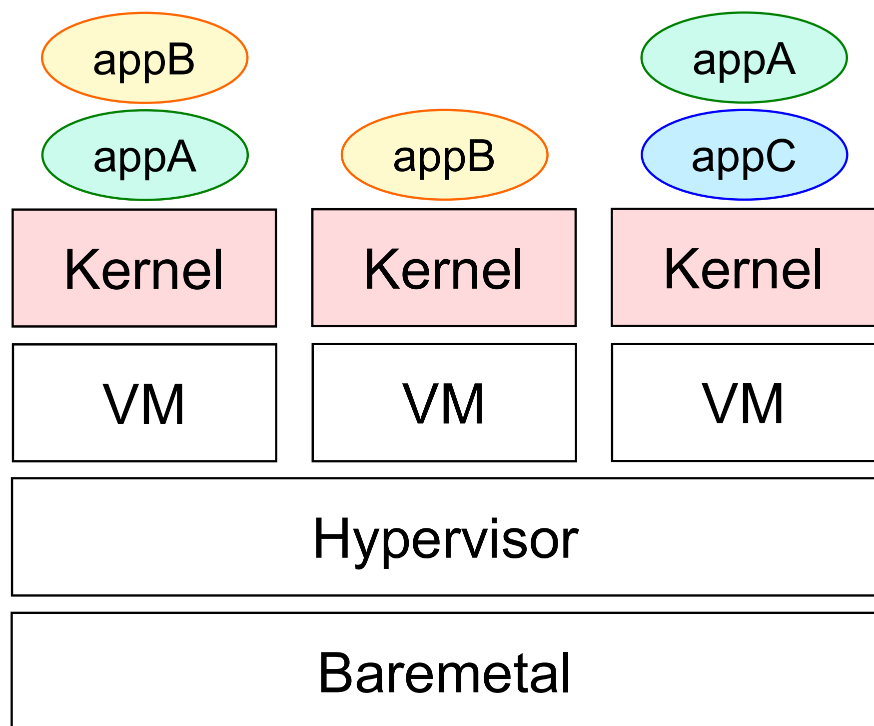
VMとコンテナは何が違うのか？

仮想化レベルの違いと隔離性

[VM] 仮想化レベルの違いと隔離性

VM 毎に Kernel が用意されて実行されるため 隔離性が高い

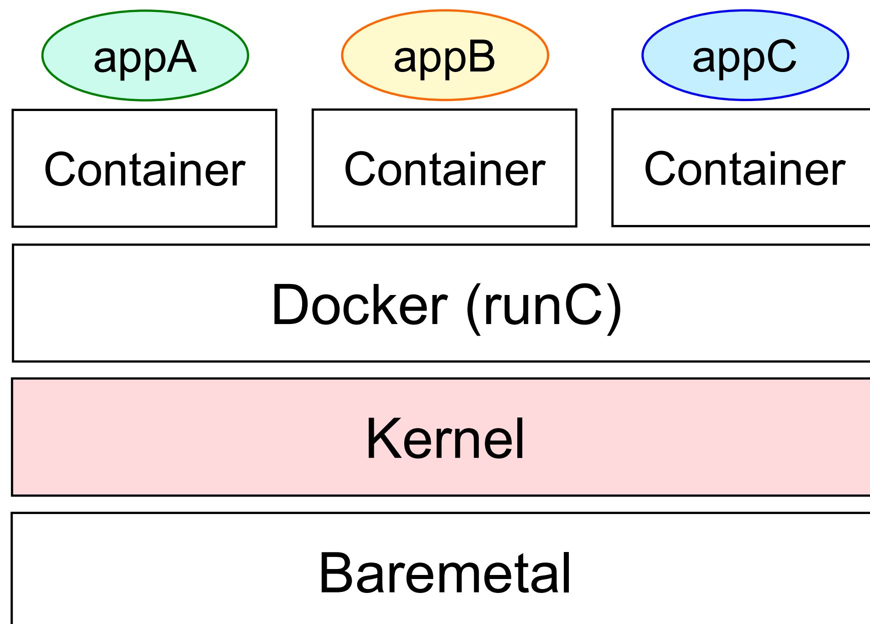
完全仮想化 / 準仮想化 によって厳密な隔離性は異なる



[CloudNative] 仮想化レベルの違いと隔離性

一般的なコンテナランタイムの runC（プロセス型）の場合、
Kernel を共有しているため 隔離性が低い（ただし 高速な起動・低オーバーヘッド）

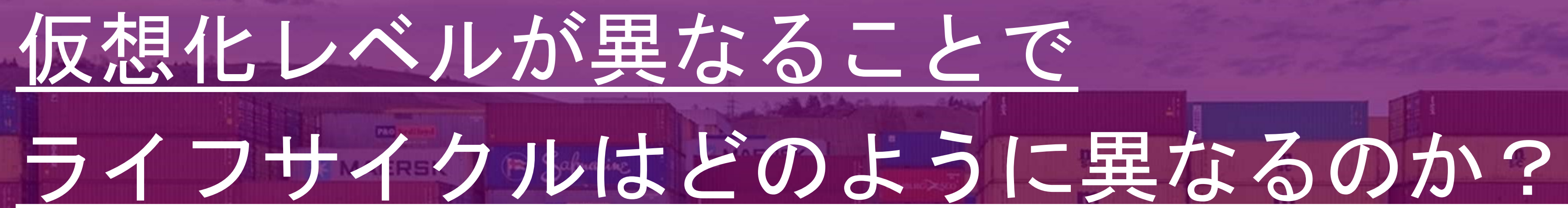
cgroups：リソース使用量の制御
linux namespace：リソースの分離



隔離性の高いコンテナランタイムは存在

- gVisor（サンドボックス型）
- nabla-containers（ユニカーネル型）
- Firecracker（microVM 型）
- kata-containers（VM 型）

おすすめ「ユビキタスデーターセンターOSの文脈におけるコンテナ実行環境の分類」
https://hb.matsumoto-r.jp/entry/2019/02/08/135354



仮想化レベルが異なることで ライフサイクルはどのように異なるのか？

ライフサイクル

[VM] ライフサイクル

VM は基本的に数ヶ月〜数年単位で利用することも多い

VM は reboot などを行うことで使い回すことが多い

混在する同一 VM が実は環境やバージョンが異なることも

[CloudNative] ライフサイクル

VM を「停止しないでください」　は実際ある
コンテナの場合は「停止しないでください」は極力極力 NG

Kubernetes や Docker の場合、
コンテナは様々なタイミングで気軽に停止される

終了時には SIGTERM が通知されるため、
アプリケーションがハンドリングできるように開発する

さて話は少し変わり
大規模なインフラ管理を効率的に行う手段
といえば、Infrastructure as Code

インフラ構成のコード化

[VM] インフラ構成のコード化

Bootstrap & Orchestrate

Terraform

CloudFormation / Heat

Configuration

Chef / Ansible / Puppet / Salt

Packer

Deploy

Jenkins / Capistrano

パターン 1

- Terraform で VM を起動
- CAPS でセットアップ
- Jenkins でアプリのデプロイ

・イメージビルド時間 : 0 min
・デプロイ時間 : 10 min〜60 min
・同一環境になることが保証されない
・複数のツールの習得が必要

[VM] インフラ構成のコード化

Bootstrap & Orchestrate

Terraform

CloudFormation / Heat

Configuration

Chef / Ansible / Puppet / Salt

Packer

Deploy

Jenkins / Capistrano

パターン 2

- Packer で VM イメージのビルド
- Terraform で VM を起動
- Jenkins でアプリのデプロイ

・イメージビルド時間 : 10 min - 30 min

・デプロイ時間 : 5 min

・同一環境になることが保証される

・複数のツールの習得が必要

[CloudNative] インフラ構成のコード化

Bootstrap & Orchestrate

Kubernetes Manifests

Configuration

Dockerfile

Kubernetes Manifests

Deploy

Kubernetes Manifests

パターン 1

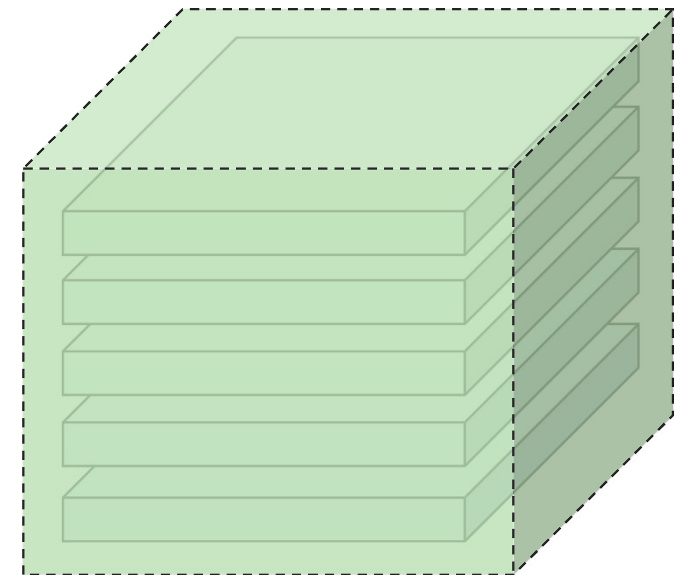
- Dockerfile でイメージのビルド
- K8s Manifest でコンテナを起動
  - ミドルウェア
  - アプリケーション
  - サービス間の接続性も管理

・イメージビルド時間：1 min - 10 min
・デプロイ時間：5 sec – 1 min
・同一環境になることが保証される
・少ない数のツールの習得で十分
　イメージ化がとても容易

[CloudNative] インフラ構成のコード化

Dockerfile

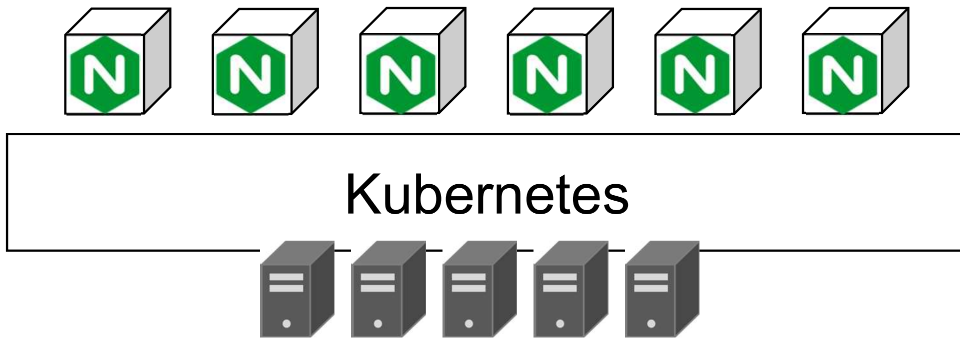
```
FROM centos:7

RUN yum -y install epel-release

RUN yum -y install nginx

COPY nginx.conf /etc/nginx/

ENTRYPOINT ["nginx", "-g", "daemon off;"]
```



イメージのビルド

Build Once, Run Anywhere

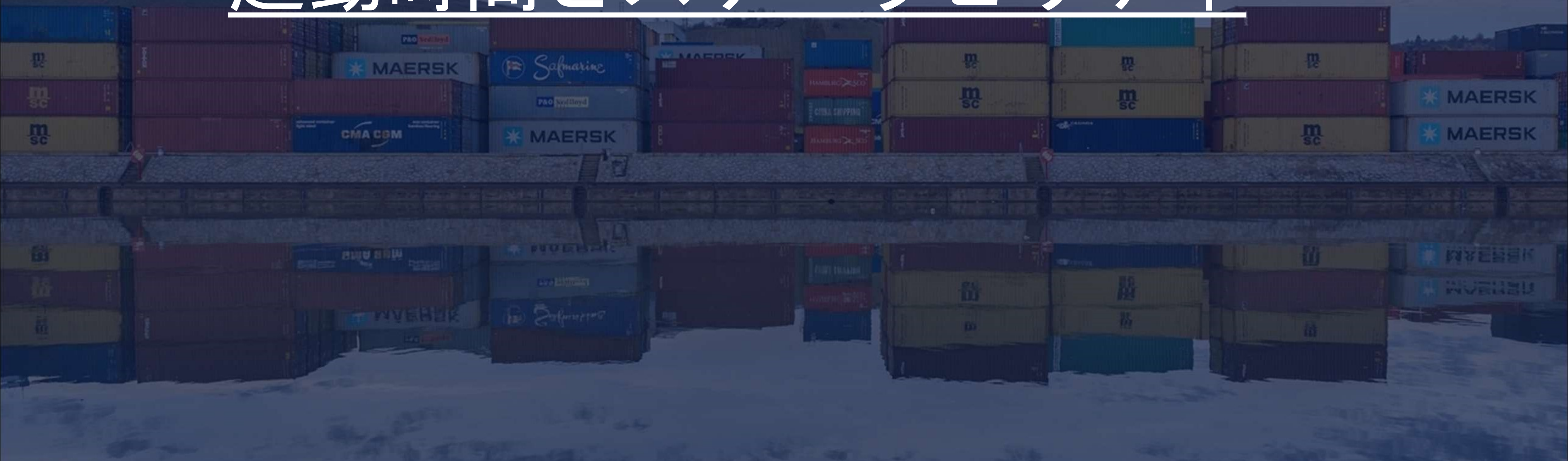
[CloudNative] インフラ構成のコード化




```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deployment
spec:
  replicas: 3
  template:
    spec:
      containers:
        - name: app-container
          image: nginx:1.16
```

再現性が高いことと起動速度が早いことでスケーラビリティはどのように変わるのか

起動時間とスケーラビリティ



[VM] 起動時間とスケーラビリティ

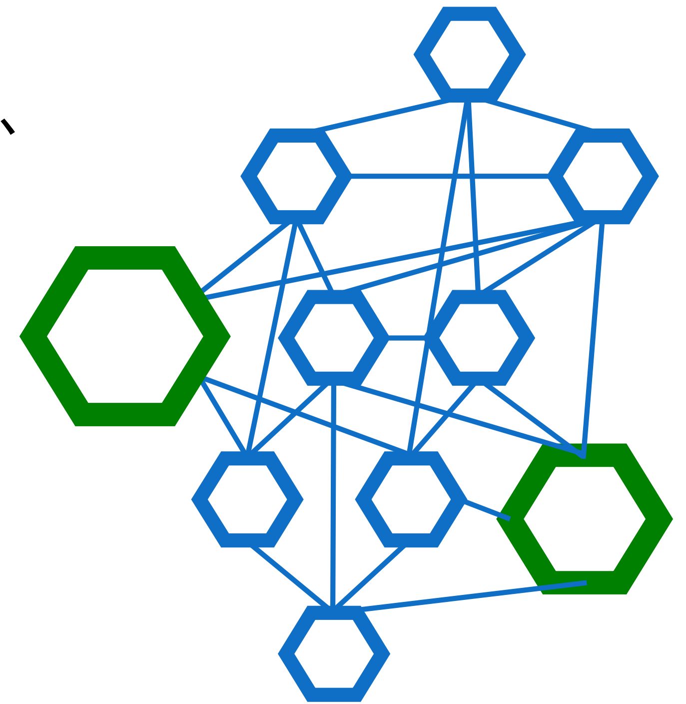
VM の場合、起動時間は **1 分 ～ 3 分**程度

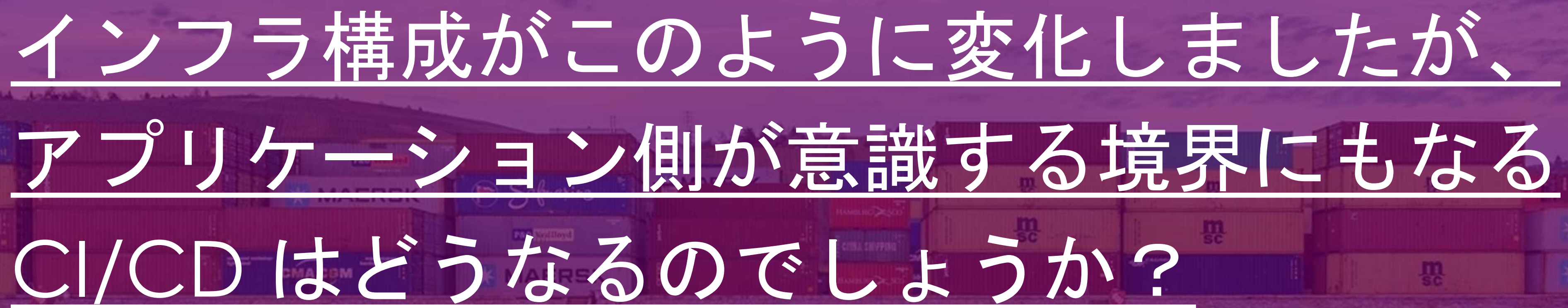
モノリシックなアプリケーション構成の場合には、
特定の機能だけスケールアウトが必要な状態でも
全体をスケールさせる必要がある

[CloudNative] 起動時間とスケーラビリティ

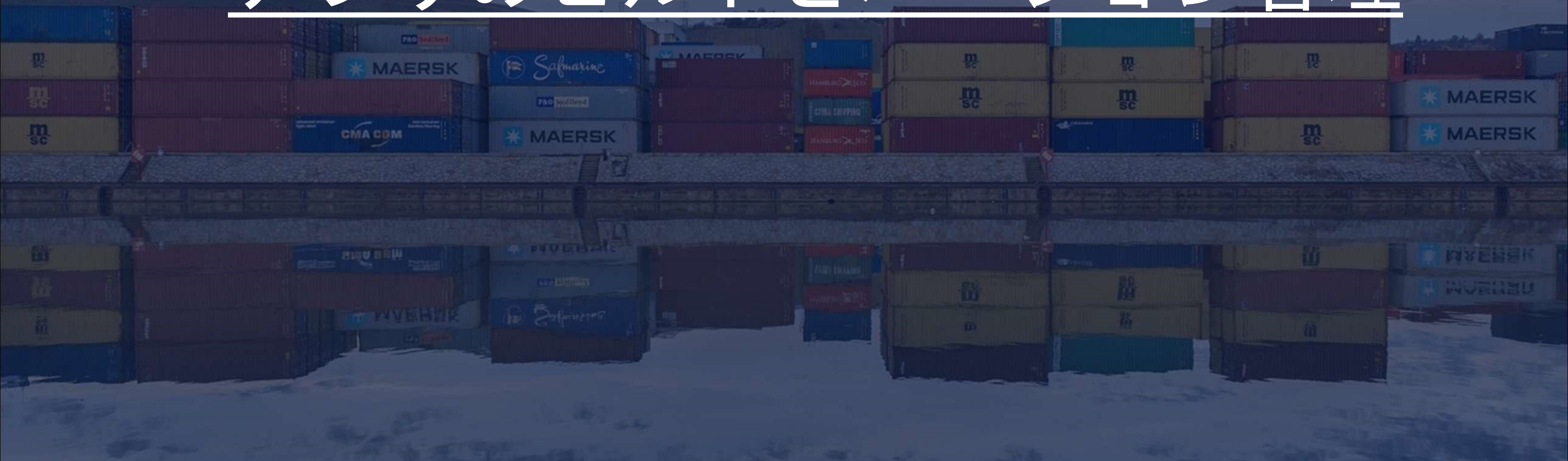
コンテナの場合、起動時間は **1 sec ～ 5 sec** 程度
　≒ プロセスを立ち上げるのに近しい

合わせてマイクロサービス化を行っている場合、
特定の機能だけをスケーリングするため
・より早い立ち上がりが可能
・集約率が向上するケースも

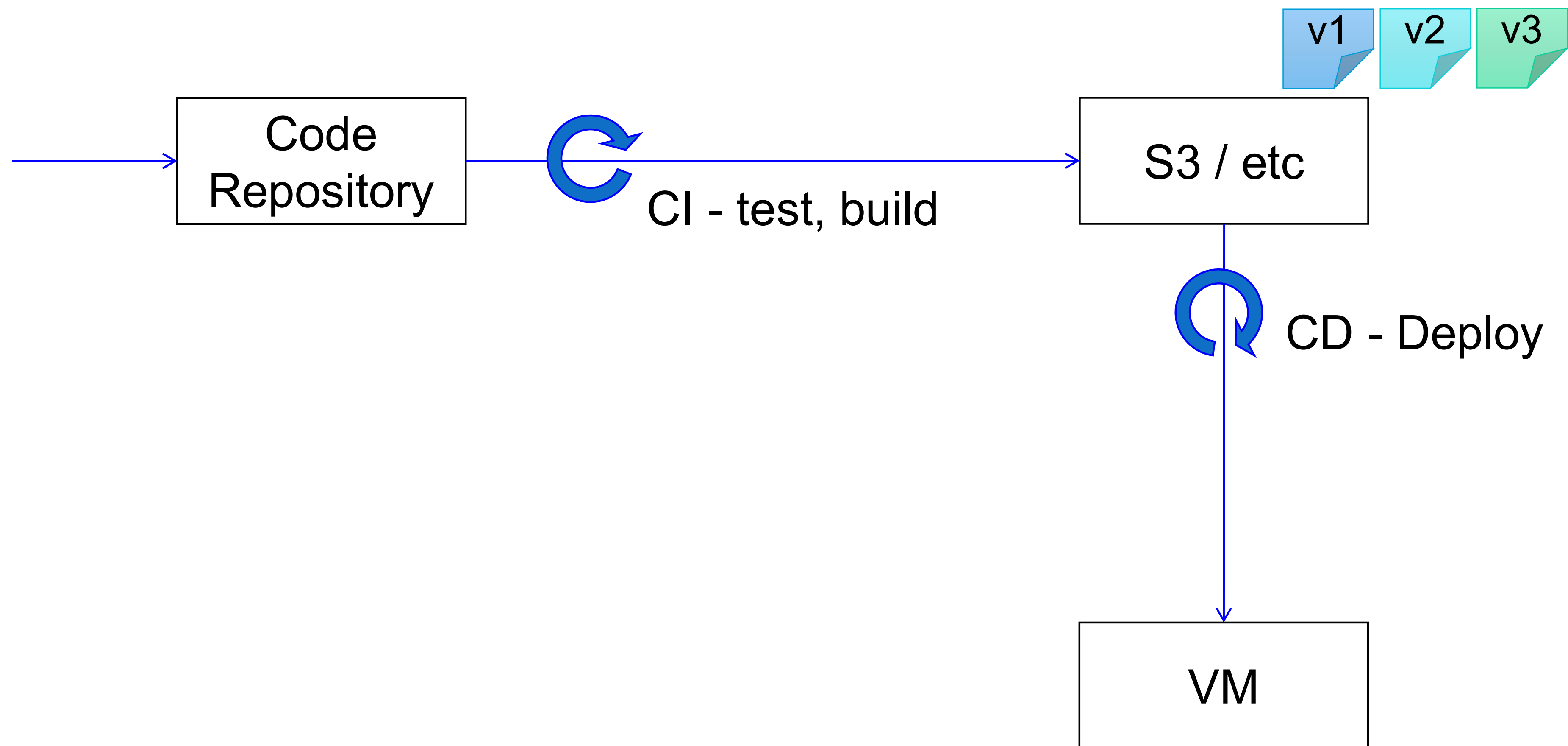



インフラ構成がこのように変化しましたが、
アプリケーション側が意識する境界にもなる
CI/CD はどうなるのでしょうか？

アプリのビルドとバージョン管理

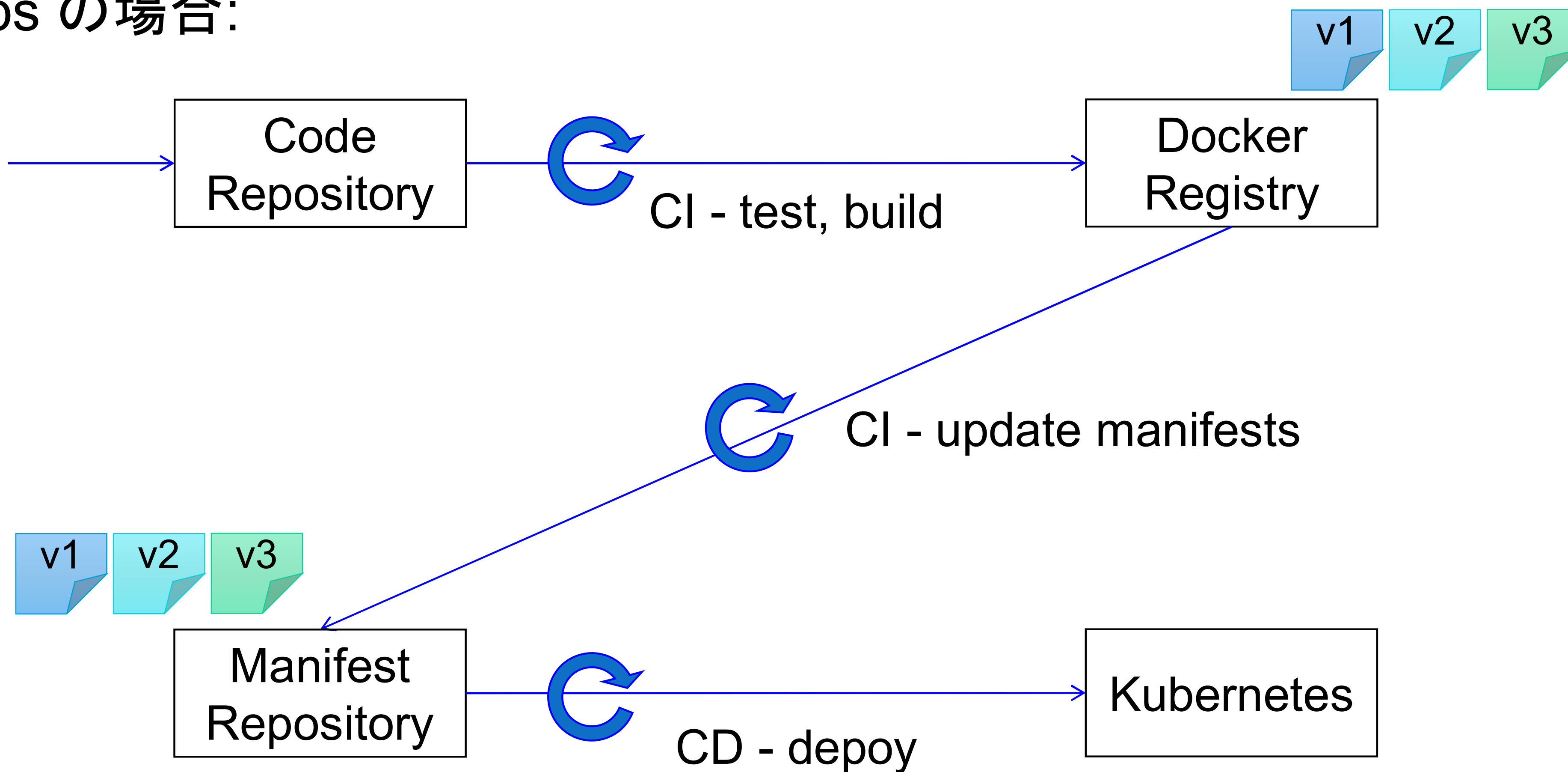


[VM] アプリのビルドとバージョン管理



[CloudNative] アプリのビルドとバージョン管理

GItOps の場合:

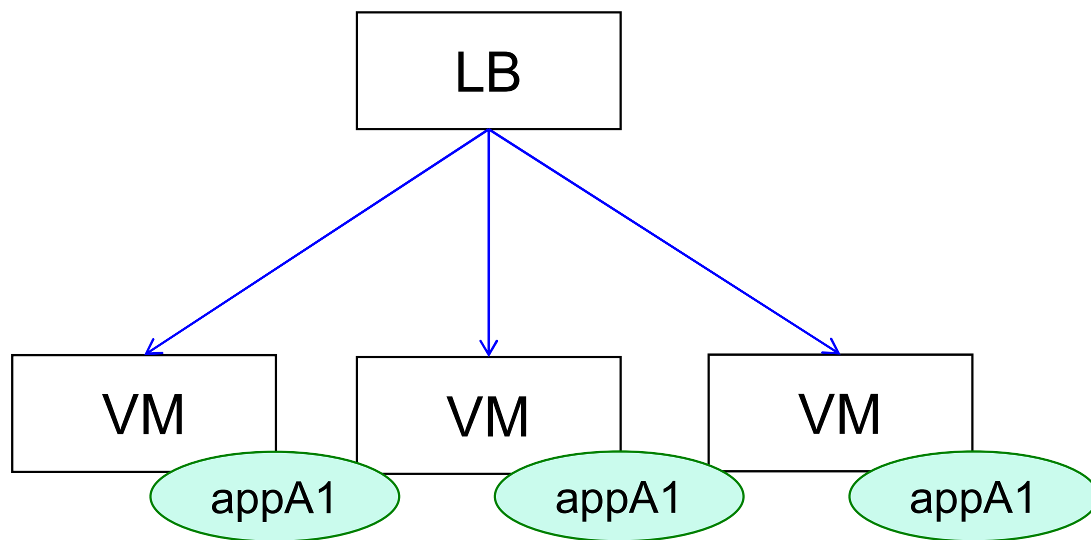


CI/CD を契機に
どのようにアプリケーションやインフラが
更新されるのか？

ローリングアップデート

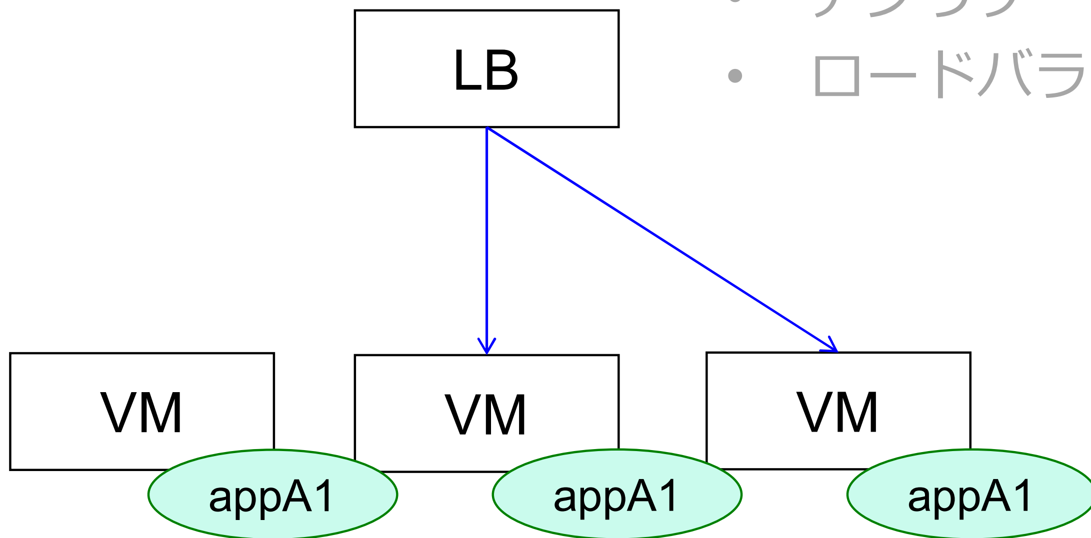
[VM] ローリングアップデート

通常のアプリケーションはスケーラブルにするために
ロードバランサ配下に配置されていることが一般的



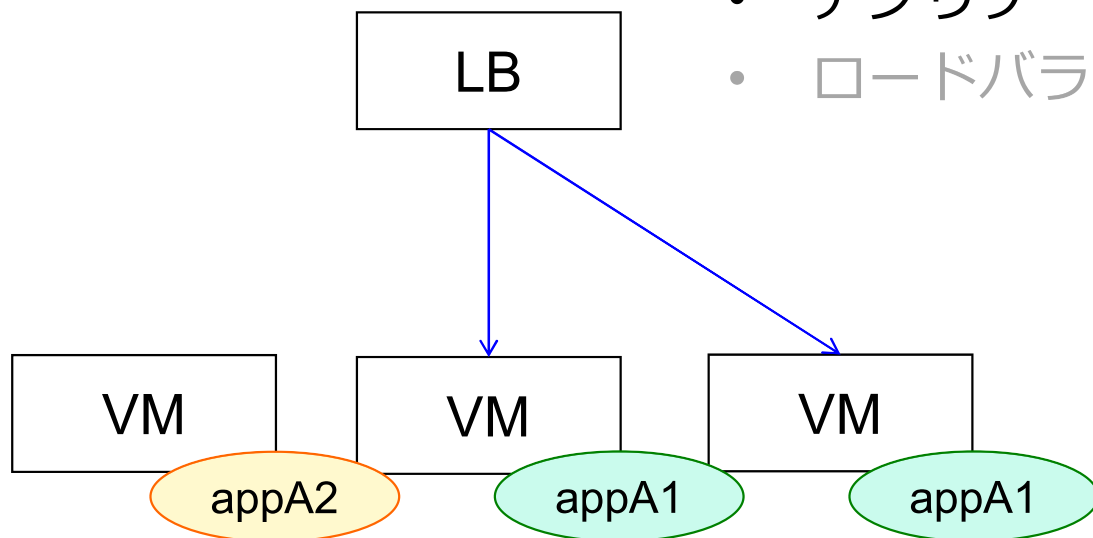
[VM] ローリングアップデート

- ロードバランサのメンバーから対象 VM を外す
- アプリケーションを停止
- VM 上にアプリケーションをデプロイ
- アプリケーションを起動
- ロードバランサのメンバーから対象 VM を追加

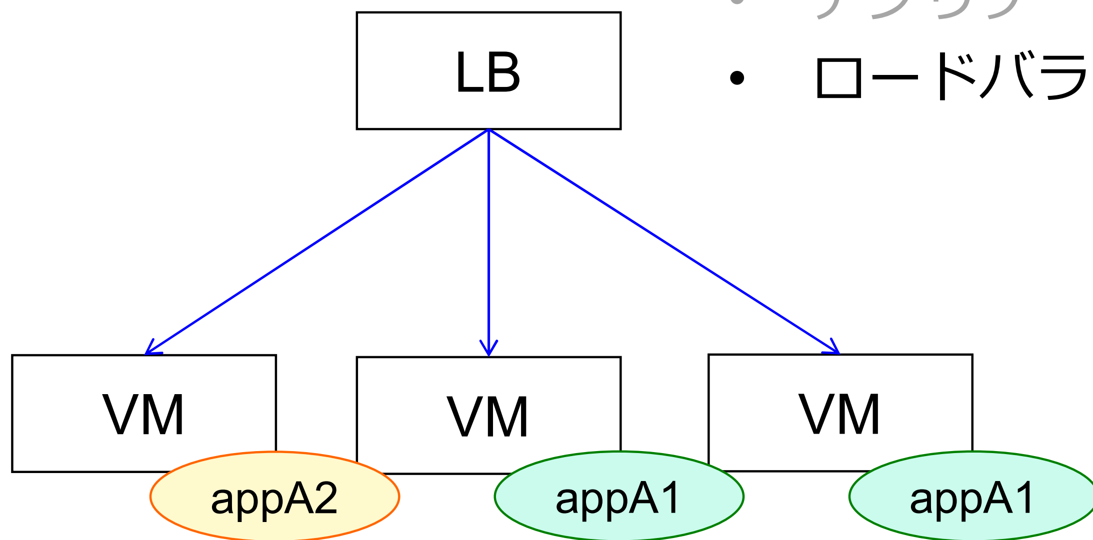


[VM] ローリングアップデート

- ロードバランサのメンバーから対象 VM を外す
- アプリケーションを停止
- VM 上にアプリケーションをデプロイ
- アプリケーションを起動
- ロードバランサのメンバーから対象 VM を追加

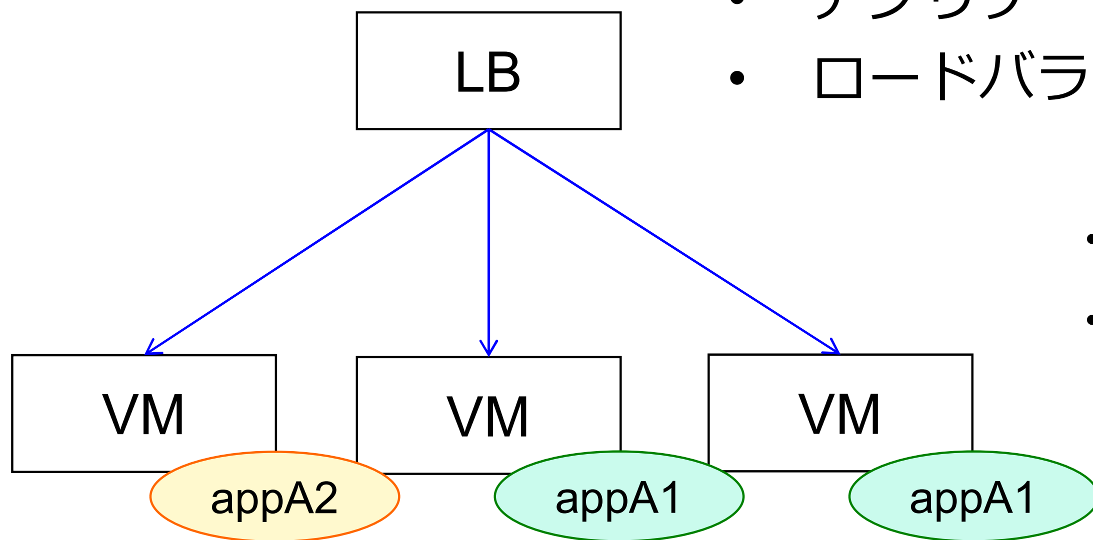


[VM] ローリングアップデート



- ロードバランサのメンバーから対象 VM を外す
- アプリケーションを停止
- VM 上にアプリケーションをデプロイ
- アプリケーションを起動
- ロードバランサのメンバーから対象 VM を追加

[VM] ローリングアップデート



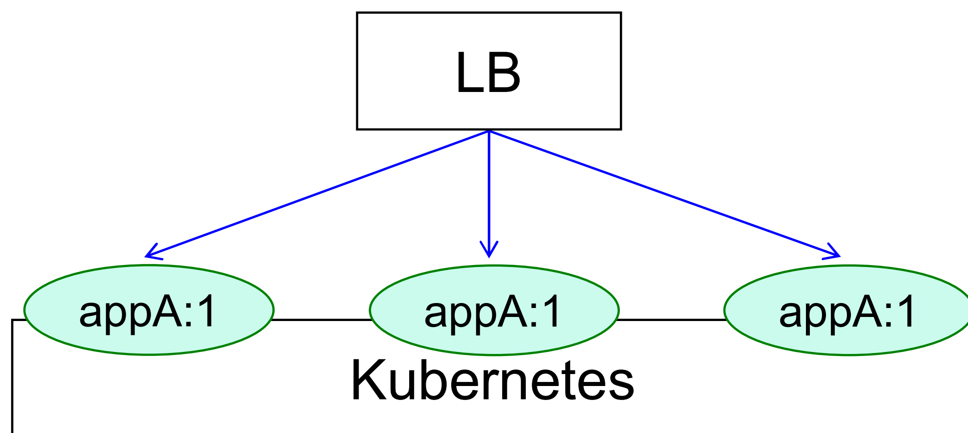
- ロードバランサのメンバーから対象 VM を外す
- アプリケーションを停止
- VM 上にアプリケーションをデプロイ
- アプリケーションを起動
- ロードバランサのメンバーから対象 VM を追加

・闇の独自スクリプトを利用しがち
・アップデートの細かい制御が困難
  ・更新間隔
  ・タイムアウト
  ・失敗時のロールバック

[CloudNative] ローリングアップデート

Kubernetes の場合にはマニフェストを書き換えて
「kubectl apply」コマンドで再適用するだけ

あとは Kubernetes が自動的にアップデート処理を行う

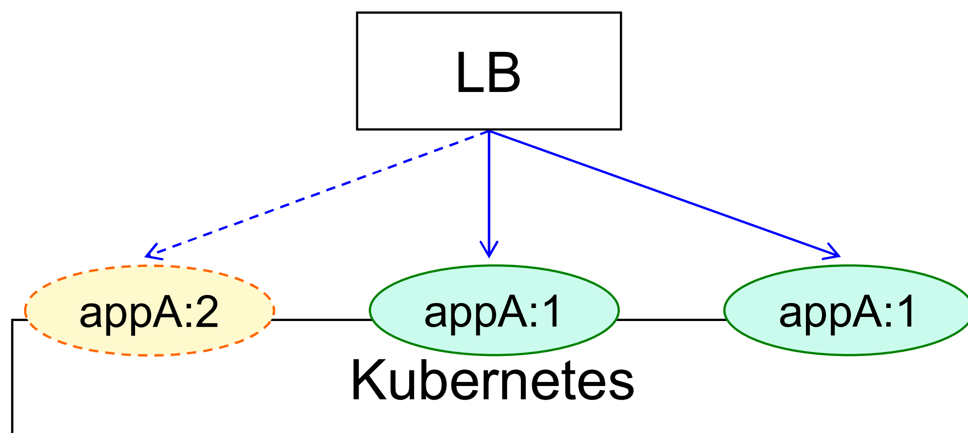



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 1
  template:
    spec:
      containers:
        - name: app-container
          image: app:A1
```

[CloudNative] ローリングアップデート

Kubernetes の場合にはマニフェストを書き換えて
「kubectl apply」コマンドで再適用するだけ

あとは Kubernetes が自動的にアップデート処理を行う

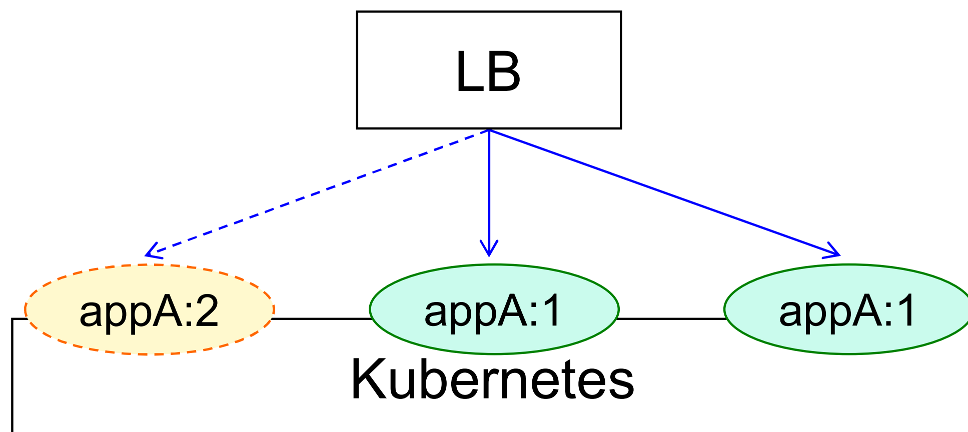



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 1
  template:
    spec:
      containers:
        - name: app-container
          image: appA:2
```

[CloudNative] ローリングアップデート

- デファクトとなった手段でアップデートが可能
- 更新間隔、タイムアウト、ロールバックなど細かい制御

CloudFormation や OpenStack Heat などでも作り込めば同等のことは可能



ローリングアップデートでも密接に関わる
ロードバランサなどの管理はどう異なるのか

LoadBalancer との連携

[VM] LoadBalancer との連携

VM の起動とは別にロードバランサの作成も必要

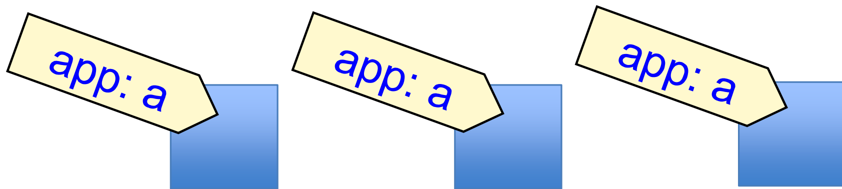
メンバーの更新や管理も行う必要がある

基本的にインフラレイヤーのメンバーが管理する必要がある

[CloudNative] LoadBalancer との連携

起動したコンテナには ラベル が付与されている

Kubernetes や周辺エコシステムでは
ラベルを元に様々な処理を行う




```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-deployment
spec:
  replicas: 3
  template:
    metadata:
      labels: {app: a}
    spec:
      containers:
        - name: app-container
          image: app:A
```

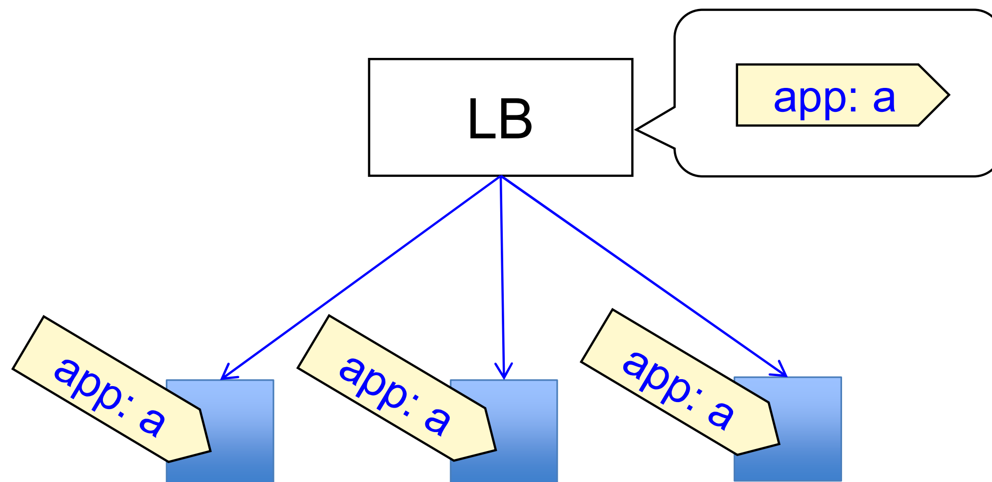
[CloudNative] LoadBalancer との連携

自動的に LoadBalancer を作成し、自動的に連携を行う

- コンテナやノードの追加時にも自動的にメンバ更新
- ローリングアップデート時のメンバ管理

```
apiVersion: v1
kind: Service
metadata:
  name: sample-svc
spec:
  type: LoadBalancer
  ports:
    - name: "http-port"
      protocol: "TCP"
      port: 8080
      targetPort: 80
  selector:
      app: a
```

開発者も管理が可能



ロードバランサの転送先を決定する
ヘルスチェックはどのように異なるのか

ヘルスチェック

[VM] ヘルスチェック

LB からのヘルスチェックが落ちたノードへは転送しないケースが多いが

細かい挙動チェックによる自動回復などは行っていないケースが多い

[CloudNative] ヘルスチェック

Kubernetes ではコンテナ単位に細かいヘルスチェックを利用可能

livenessProbe

　失敗時にコンテナの再起動を行う

readinessProbe

　失敗時はトラフィックを流さない

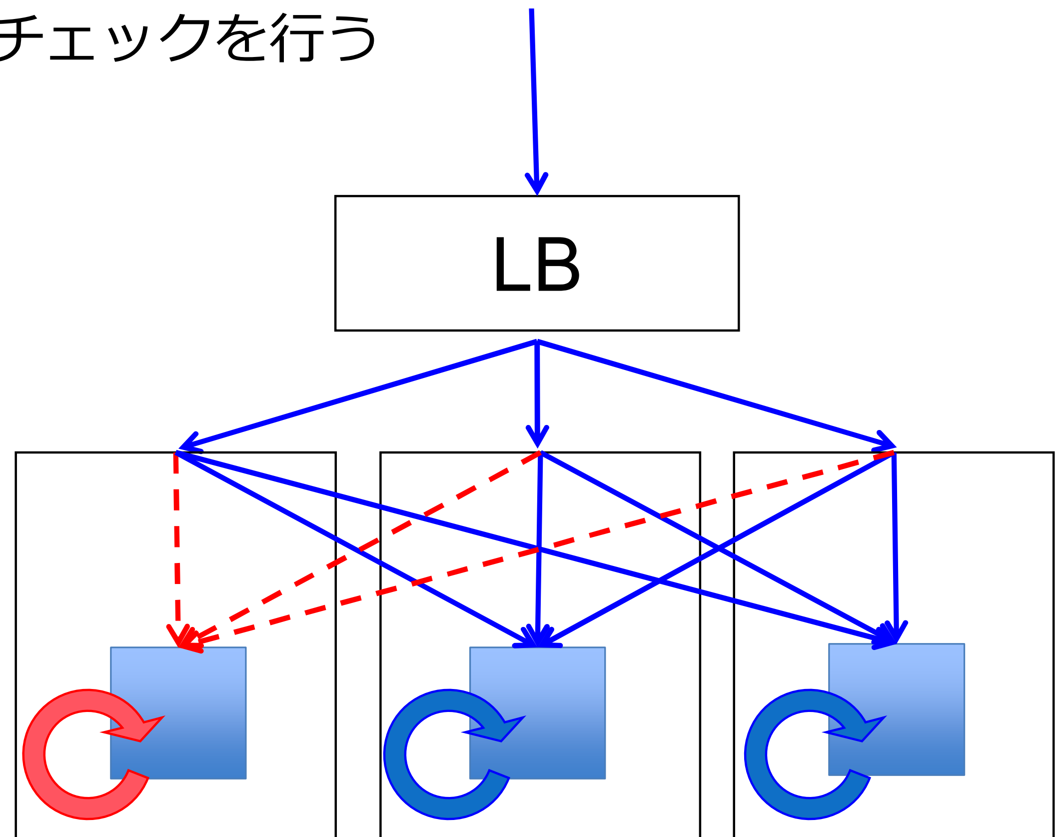
（ヘルスチェックエンドポイントを用意するようになるので副次的に品質が上がる）

```
containers:
   - name: nginx-container
     image: nginx:1.12
     ports:
     - containerPort: 80
     livenessProbe:
       httpGet:
         path: /healthz
     readinessProbe:
       httpGet:
         path: /readiness
```

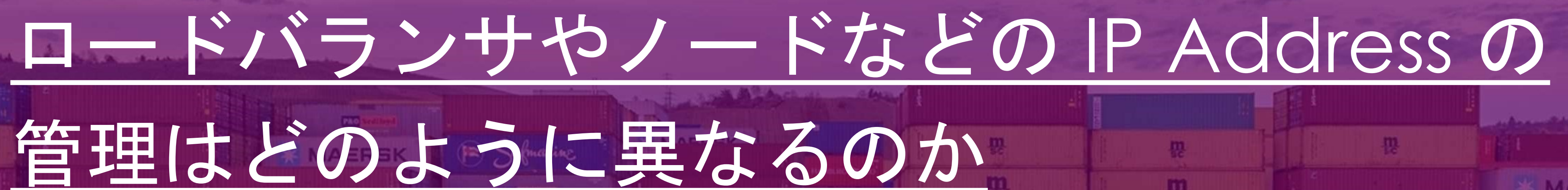
[CloudNative] ヘルスチェック

従来の LoadBalancer からのヘルスチェックとは異なり、

Kubernetes のノード側で細かいヘルスチェックを行う



ロードバランサやノードなどの IP Address の管理はどのように異なるのか



IP Address の管理

[VM] IP Address の管理

VM ごとに IP アドレスを把握・管理し、Ansible / Chef などで構成管理を実行

基本的には IP Address の管理からは逃れづらい

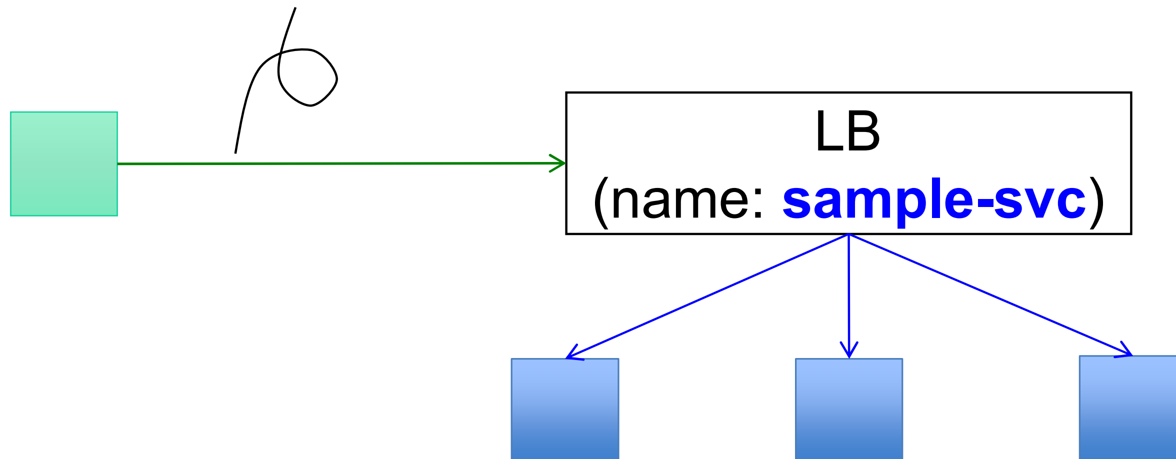
IP Address 管理のスプレッドシートから脱却したい

[CloudNative] IP Address の管理

Kubernetes では IP Address を一切管理しない

Kubernetes 内部の通信は
サービスディスカバリによって成立

http://**sample-svc**.default.svc.cluster.local



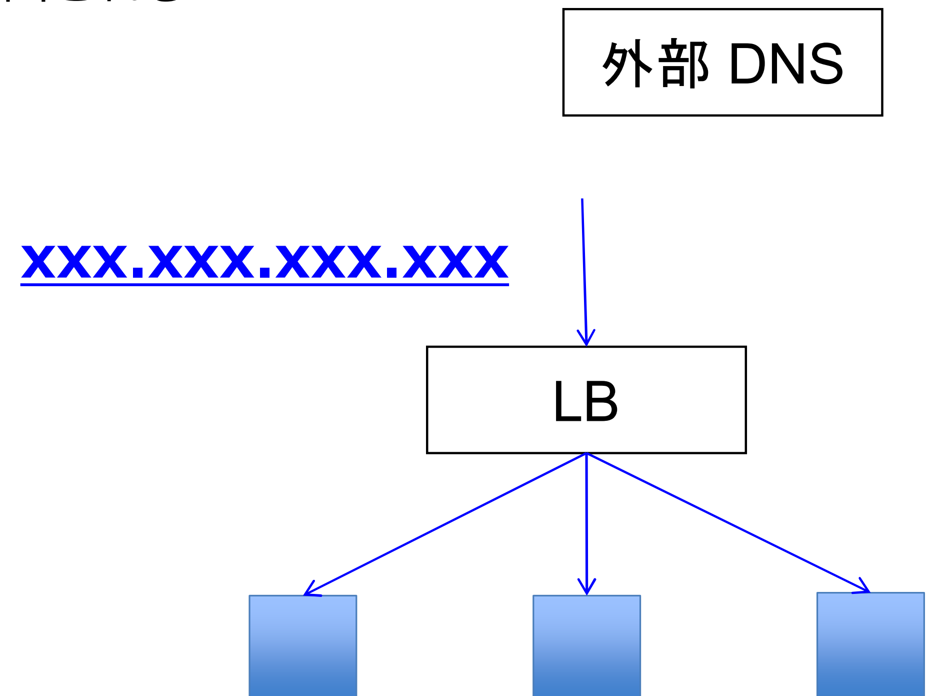

```yaml
apiVersion: v1
kind: Service
metadata:
  name: sample-svc
spec:
  type: LoadBalancer
  ports:
    - name: "http-port"
      protocol: "TCP"
      port: 8080
      targetPort: 80
  selector:
    app: a
```

[CloudNative] IP Address の管理

Kubernetes 外部の通信も external-dns を利用して

グローバル IP と外部 DNS に自動登録させることが可能

1. ロードバランサにグローバル IP が自動的に払い出される
2. 外部の DNS に自動的にグローバル IP と登録

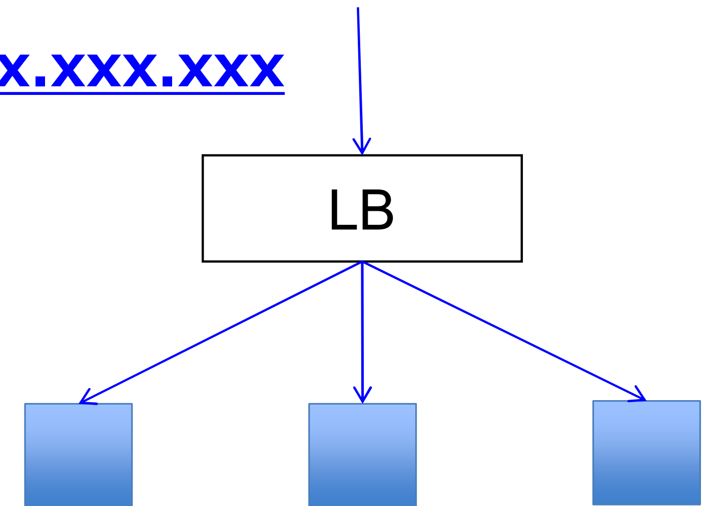


[CloudNative] IP Address の管理

Kubernetes 外部の通信も external-dns を利用して

グローバル IP と外部 DNS に自動登録させることが可能

1. ロードバランサにグローバル IP が自動的に払い出される
2. 外部の DNS に自動的にグローバル IP と登録



[CloudNative] IP Address の管理

Kubernetes 外部の通信も external-dns を利用して

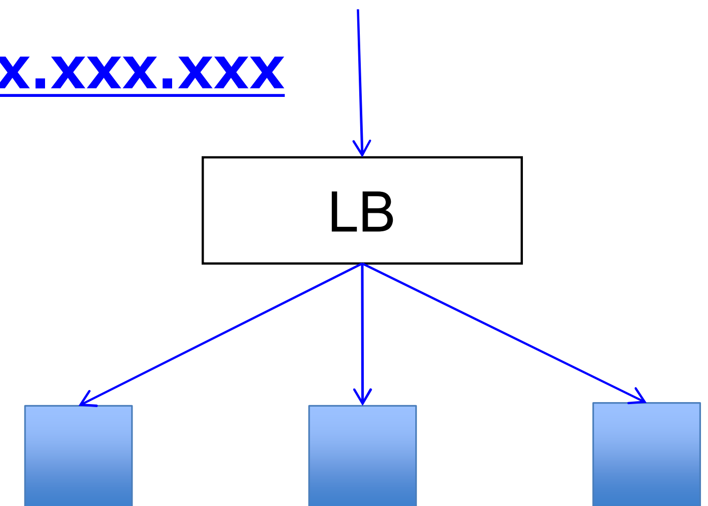
グローバル IP と外部 DNS に自動登録させることが可能

1. ロードバランサにグローバル IP が自動的に払い出される
2. 外部の DNS に自動的にグローバル IP と登録

amsy.dev =>
xxx.xxx.xxx.xxx

外部 DNS

xxx.xxx.xxx.xxx



さらに cert-manager を利用することで

ACME を利用して証明書の自動発行/更新も可能

L7でSSLなどの終端を行っている場合など
証明書の管理はどのように異なるのか

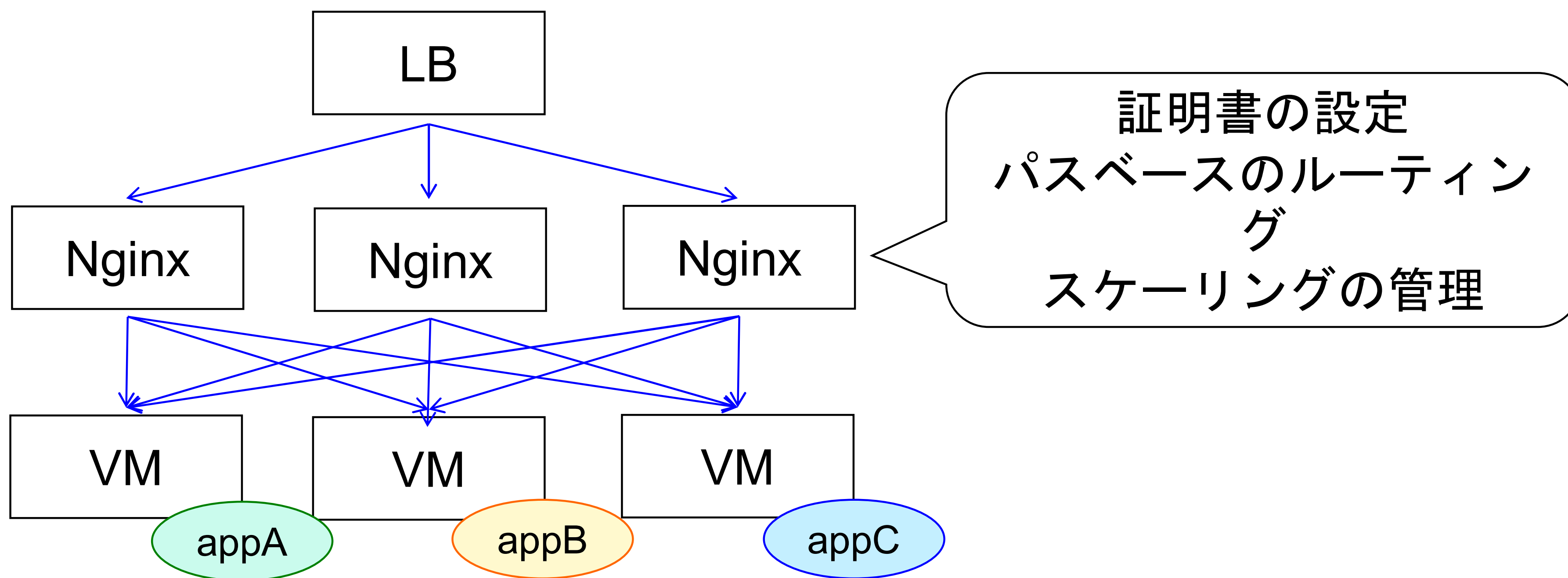
証明書の管理

[VM] 証明書の管理

SSL 終端やパスベースの振り分けはリバースプロキシの Nginx を立てて運用

（クラウド環境の場合は L7 LB を利用することも）

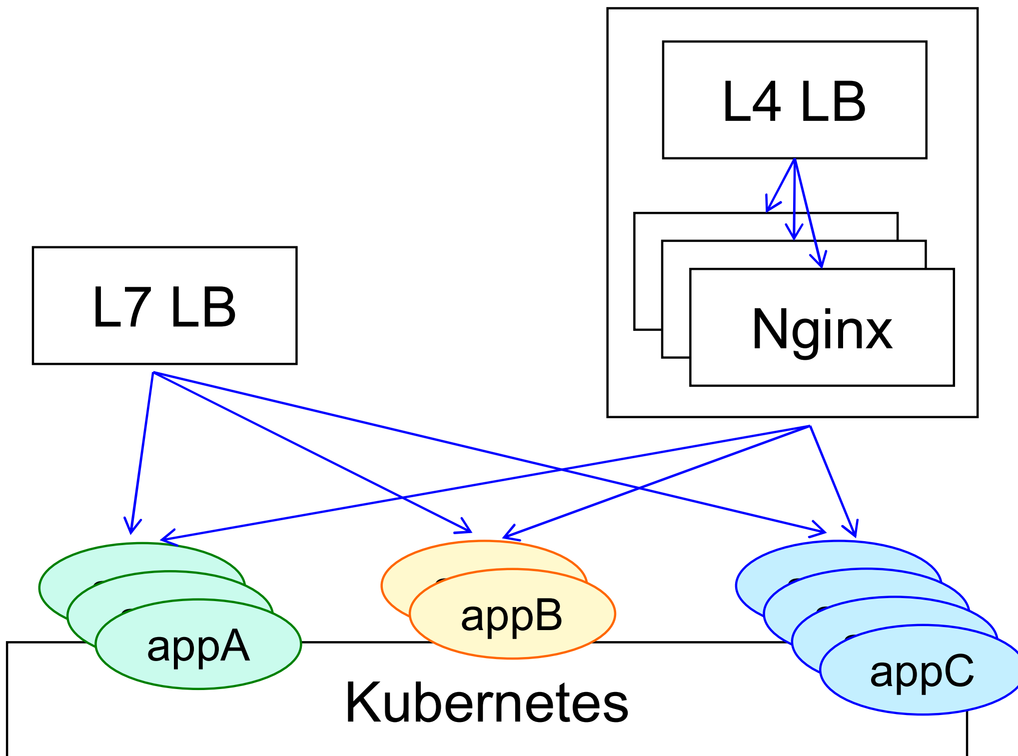
VM の証明書管理は Chef / Ansible などで更新処理



[CloudNative] 証明書

L7 LoadBalancer も Manifest で管理

Nginx と自動連携するものもある




```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: sample-ingress
spec:
  rules:
  - host: sample.example.com
    http:
      paths:
      - path: /path1/*
        backend:
          serviceName: sample-ingress-svc-1
          servicePort: 8888
      - path: /path2/*
        backend:
          serviceName: sample-ingress-svc-2
          servicePort: 8888
  tls:
  - hosts:
    - sample.example.com
    secretName: tls-sample
```

[CloudNative] 証明書

証明書はすべて Secret リソースで管理

更新はファイルを変更して
「kubectl apply」で登録するだけ

```
apiVersion: v1
kind: Secret
metadata:
  name: tls-sample
type: kubernetes.io/tls
data:
  tls.crt: LS0tLS1CRUd...=
  tls.key: LS0tLS1CRUd...=
```

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: sample-ingress
spec:
  rules:
  - host: sample.example.com
    http:
      paths:
      - path: /path1
        backend:
          serviceName: sample-ingress-svc-1
          servicePort: 8888
      - path: /path2/*
        backend:
          serviceName: sample-ingress-svc-2
          servicePort: 8888
  tls:
  - hosts:
    - sample.example.com
    secretName: tls-sample
```

話は少し変わり、
障害時の対応はどのように異なるのか

自動回復・セルフヒーリング

[VM] 自動回復・セルフヒーリング

VM 時代の自動回復・セルフヒーリングといえば、

systemd や supervisord などの復旧に任せるなど

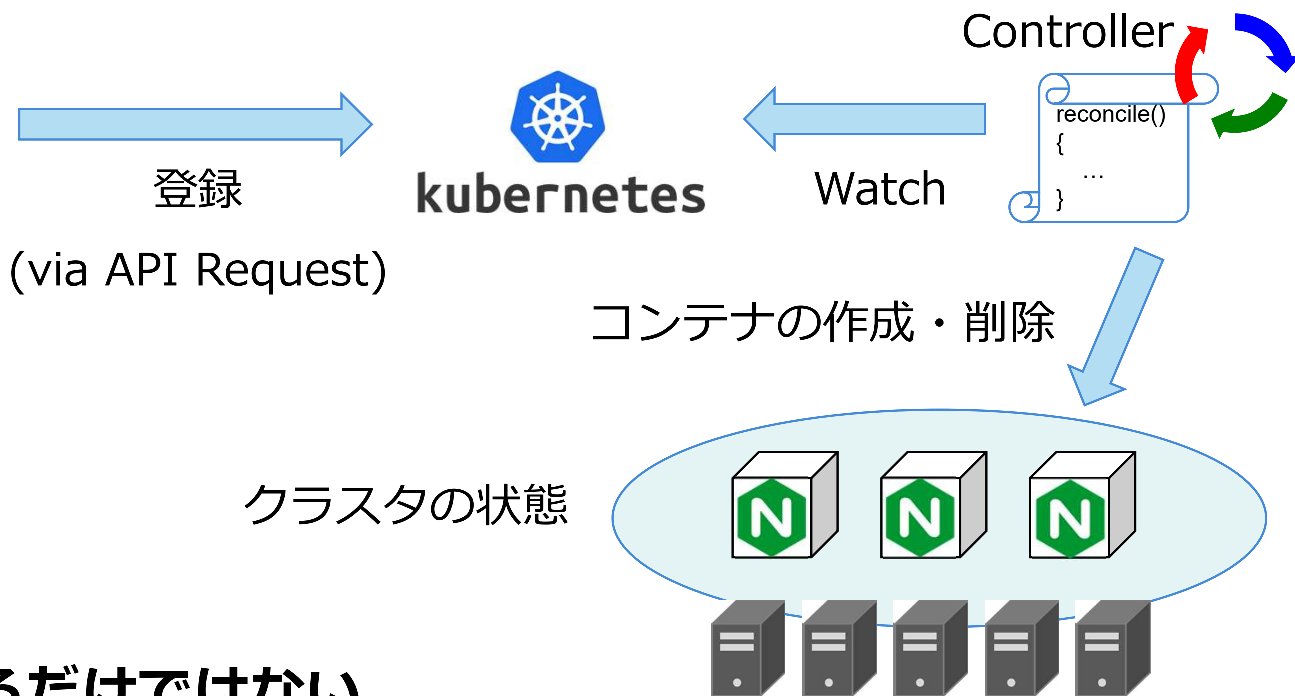
LB からのヘルスチェックが落ちたノードへは転送しないケースが多いが

細かい挙動チェックによる自動回復などは行っていないケースが多い

[CloudNative] 自動回復・セルフヒーリング

Kubernetes では **あるべき理想の状態（Desired State）** へと収束させる
何か問題が発生した場合でも、**Reconcile loop** により **セルフヒーリング**される

```yaml
kind: ReplicaSet
metadata:
  name: sample-rs
spec:
  replicas: 3
  template:
    spec:
      containers:
        - name: nginx-container
          image: nginx:1.12
```



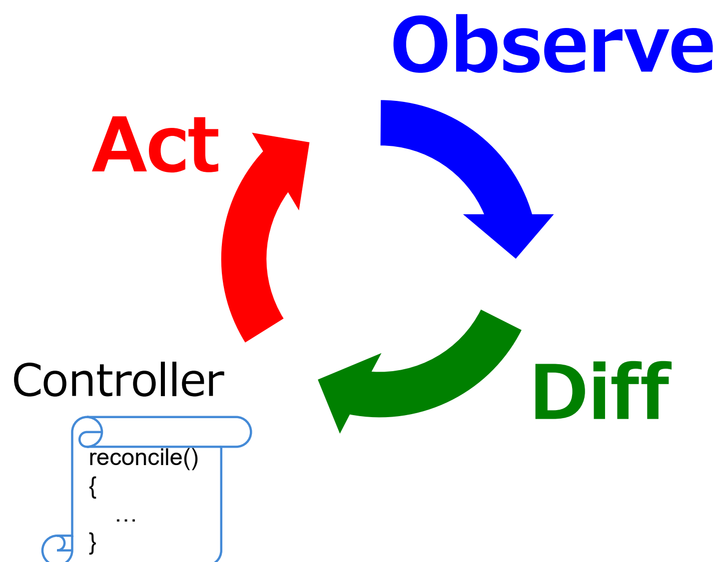
<u>登録された時に、ただ起動させるだけではない</u>

※ 厳密には Controller も API を用いて変更します。

[CloudNative] 自動回復・セルフヒーリング

Kubernetes の内部には様々な Controller と呼ばれるプログラムが動作している

Reconcile loop（調整ループ）とは、Controller 内で実行されている
あるべき状態へと収束させるループ処理 のこと



Observe: 現状を確認

Diff: 理想と現実の差分を計算

Act: 差分に対する処理を実施

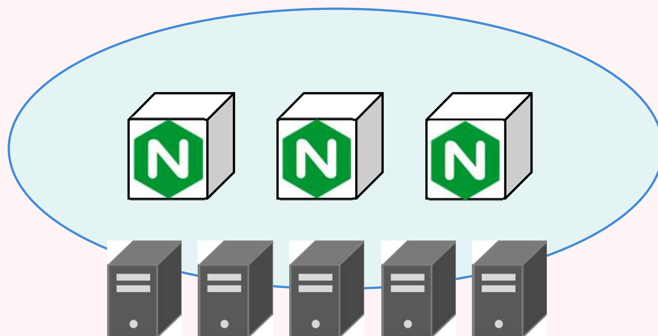
[CloudNative] 自動回復・セルフヒーリング

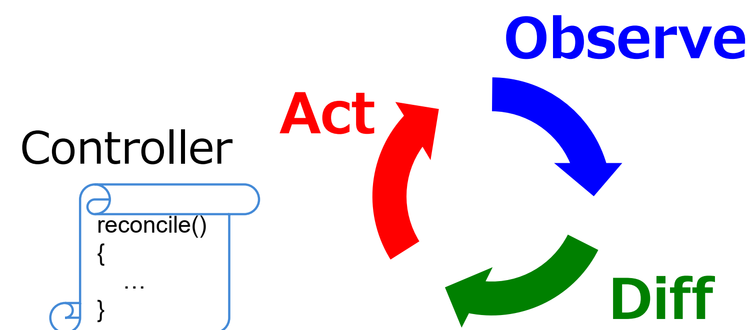
例えば、コンテナ（Pod）を 3 つ起動させる ReplicaSet リソースの場合

理想の状態

```yaml
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.12
```

クラスタの状態





Observe: 現状を確認

Diff: 理想と現実の差分を計算

Act: 差分に対する処理を実施

[CloudNative] 自動回復・セルフヒーリング

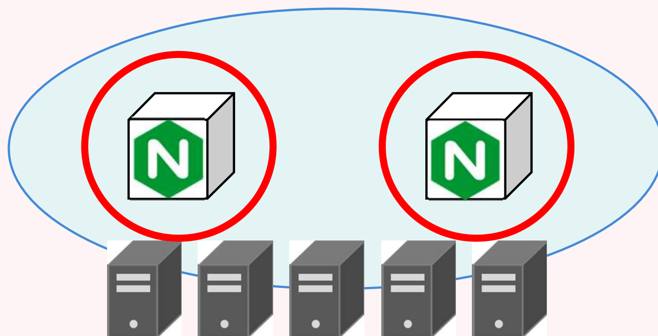
たとえば 2 つしかコンテナ（Pod）が起動していない場合…

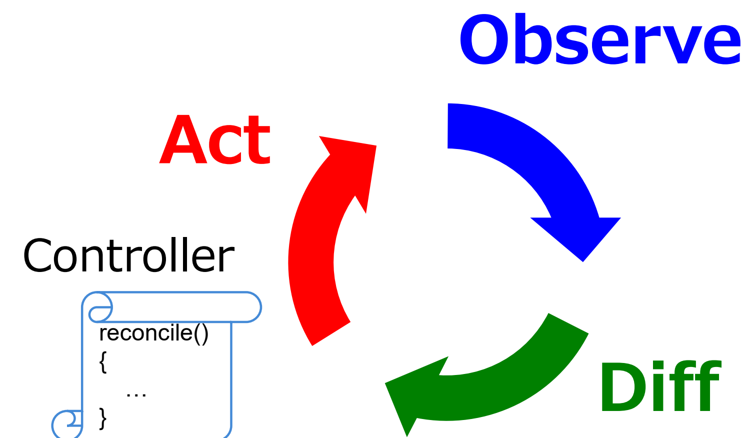
Observe: 理想=3、現状=2

理想の状態

```
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.12
```

クラスタの状態





Observe: 現状を確認

Diff: 理想と現実の差分を計算

Act: 差分に対する処理を実施

[CloudNative] 自動回復・セルフヒーリング

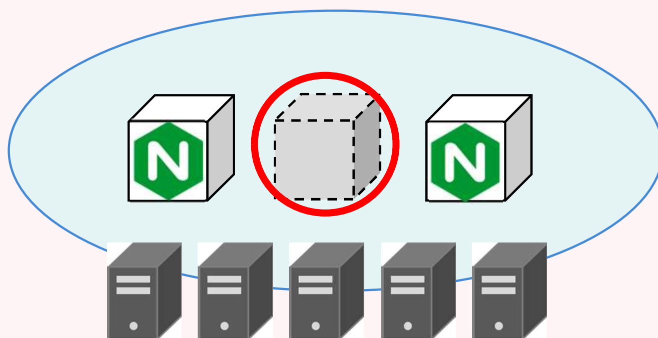
たとえば 2 つしかコンテナ（Pod）が起動していない場合…

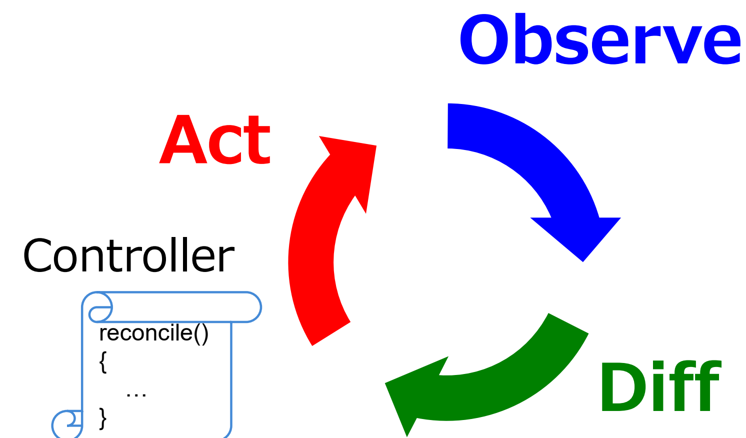
Diff: 1 つコンテナ（Pod）が足りない

理想の状態

```yaml
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.12
```

クラスタの状態





Observe: 現状を確認

Diff: 理想と現実の差分を計算

Act: 差分に対する処理を実施

[CloudNative] 自動回復・セルフヒーリング

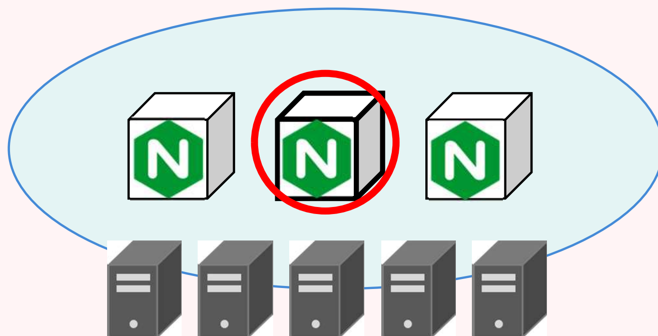
たとえば 2 つしかコンテナ（Pod）が起動していない場合…

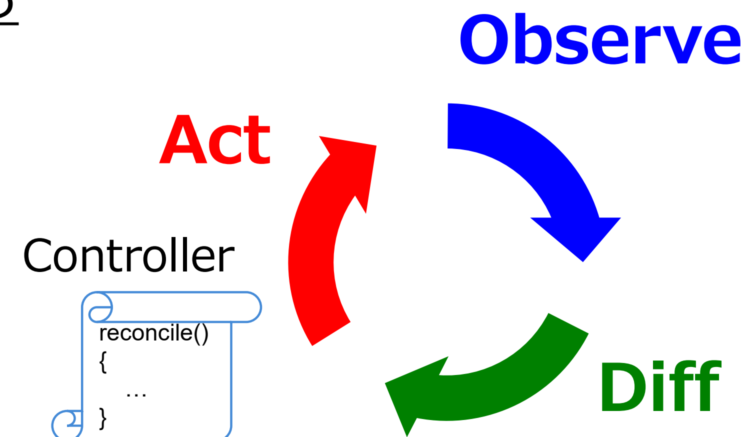
Act: 1つ nginx:1.12 のコンテナ（Pod）を作成する

理想の状態

```yaml
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: nginx-container
        image: nginx:1.12
```

クラスタの状態





Observe: 現状を確認

Diff: 理想と現実の差分を計算

Act: 差分に対する処理を実施

障害に対して弱いステートフルなアプリケーションを動作させるにはどちらが適しているのか

ステートフルなアプリケーション

[VM] ステートフルなアプリケーション

ライフサイクルが長いため、
従来どおりステートフルなアプリケーションにも向いている

[CloudNative] ステートフルなアプリケーション

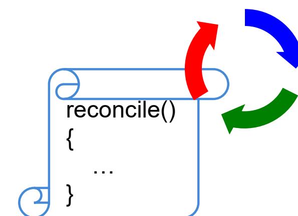
Kubernetes では Controller の仕組みを利用して、
Database の自動管理を行うことが可能（**にはなっている**）

```
apiVersion: mysql.oracle.com/v1alpha1
kind: Cluster
metadata:
  name: mysql
spec:
  members: 3
  multiMaster: true
  rootPasswordSecret:
    name: mysql-root-user-secret
```

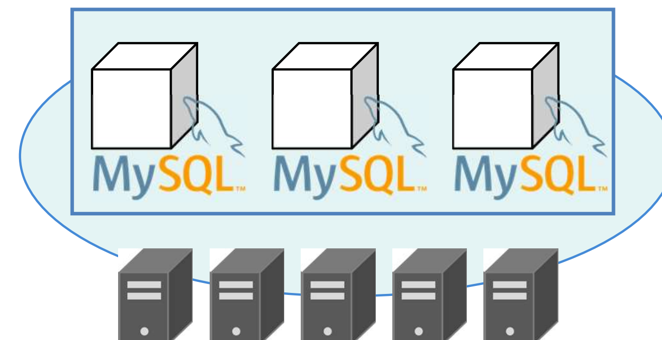
登録

(via API Request)

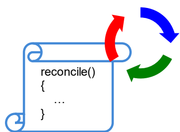
監視



MySQL Cluster の管理



Controller



独自のリソースに対してどのような処理をするか

Controller を書くことで運用が容易に

（運用ナレッジのプログラム化）

※ 厳密には Controller も API を用いて変更します。

[CloudNative] ステートフルなアプリケーション

Database on Kubernetes を実現するエコシステムは沢山ある

- MySQL Operator（presslabs, oracle）
- Vitess Operator

現実的にはコンテナの短いライフサイクルの影響で
ステートフルなアプリケーションには向かないことも多い

現時点では OSS で提供されている Operator（Controller）の成熟度が高くはない
プロダクションでステートフルなアプリケーションを動かすには注意が必要

軽量な KVS、Queue 辺りからはじめて、RDB などはその後が良さそう

では実際に Kubernetes をやるべきなのか？

学習コストと技術の遷移

[VM] 学習コストと技術の遷移

既に一般的になっているので、学習コストは少ない
知見を持っているエンジニアも数多く存在する

新しいプロダクト（エコシステム）が出てくることも少ないので、
技術刷新する機会も少なくて済む（良いことかは別）

[CloudNative] 学習コストと技術の遷移

学習コストは低くはない
　本日紹介した機能をまた一つ一つ理解していく必要がある

ただし、アプリケーション開発の作法を知る良いきっかけにはなる
　これはどのみちやらなければならないこと

Kubernetes を採用するかはさておき、
Kubernetes を学んで何に注意してシステムを構成するべきかを考えることは有用

また、エコシステムが非常に豊富で開発も活発
Kubernetes 上では様々なことが簡単にできるようになってきている

まとめ

Kubernetesは選定するべきなのか

Kubernetes を選定する理由

大前提として Cloud Native を目指すため

その上で、

- **ベンダーロックインが少ない**
  - クラウド上の Managed Kubernetes Service
  - オンプレミス上のベンダー Kubernetes 製品
- **エコシステムが充実している**
  - Service Mesh
  - { Stateful App / Serverless / ML / Edge } on Kubernetes
- **高い拡張性**
  - Platform for Platform
  - Domain-specific な Operator（運用自動化）

Let's start

new **cloud native development** :)

Thank you for your attention

follow me: **@amsy810**