令和7年度
「専門職業人材の最新技能アップデートのための専修学校リカレント教育（リ・スキリング）推進事業」

コンテナサーバーセキュリティ教材資料
解答編

本コンテナサーバーセキュリティ教材資料解答編は、文部科学省の教育政策推進事業委託費による委託事業として、一般社団法人全国専門学校情報教育協会が実施した令和７年度「専門職業人材の最新技能アップデートのための専修学校リカレント教育（リ・スキリング）推進事業」の成果物です。

情報技術者の技能アップデートのためのリカレント教育推進事業

目次

演習１　Kubernetes環境の診断と防御

解答編

演習１　Kubernetes環境の診断と防御　解答編

演習概要

脆弱なKubernetes環境に対して、Kubernetesの基本的な3つのセキュリティ対策を実装します。
対策前後での挙動の違いを通して、コンテナ環境のセキュリティ強化について理解を深めます。

Before：脆弱なK8s環境

- **過剰な権限**
  ServiceAccountに不要な権限が付与されている場合、攻撃者に悪用される可能性あり
- **無制限の通信**
  全Pod間通信が許可されている場合、横展開で他Podが攻撃されるリスクあり
- **特権Pod**
  ホストシステムへのアクセスが可能で、エスカレーションの危険性あり

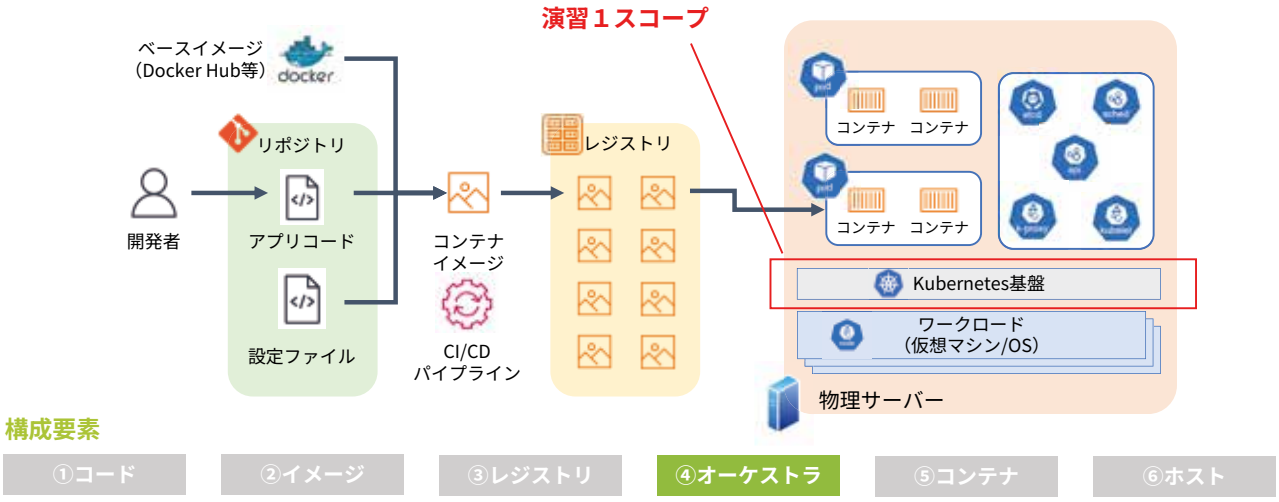
After：安全な構成

- **最小権限**
  RBACで必要最小限の権限のみを付与し、攻撃範囲を限定
- **通信制御**
  NetworkPolicyで必要な通信のみを許可し、攻撃の拡散を防止
- **権限制限**
  PSSで特権実行を禁止し、権限昇格攻撃からシステムを保護

演習１　Kubernetes環境の診断と防御　解答編

演習１の位置づけ

演習１ではコンテナオーケストレーションをターゲットにセキュリティ対策を行います。



演習１　Kubernetes環境の診断と防御　解答編

前提知識：RBAC（Role-Based Access Control）

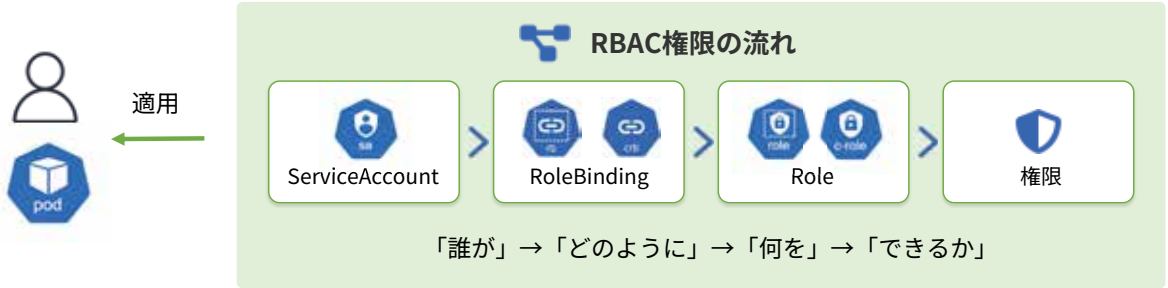
ユーザーやリソースにロール（役割）を割り当て、そのロールに紐づく権限を付与することで、システムやデータへのアクセスを管理する手法です。RBACを活用してアクセス制御を実現します。

RBACの基本概念

- ServiceAccount:　Podやユーザーに割り当てる実行アカウント
- RoleBinding/ClusterRoleBinding:　アカウントと権限の紐付け
- Role/ClusterRole:　権限の定義（namespace内/クラスタ全体）
- 最小権限の原則: 必要最小限の権限のみを付与する

RBACの適用例

- 参照Pod
  他Podに対してGETは許可
  それ以外は拒否とし、ReadOnlyにする



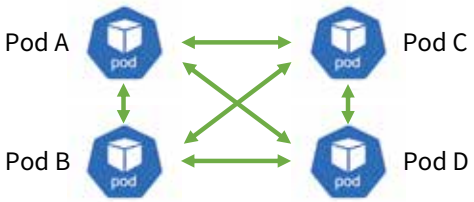
演習１ Kubernetes環境の診断と防御　解答編

前提知識：NetworkPolicy

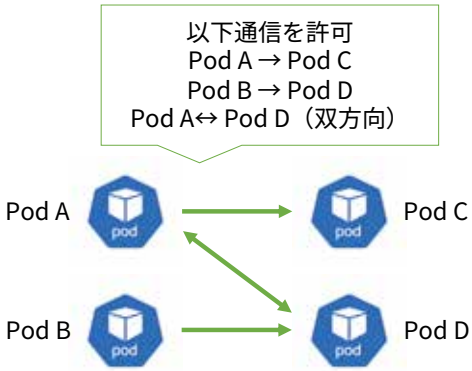
Pod間の通信を制御する Kubernetes の仕組みです。
ファイアウォールのような役割を持ち、明示的に許可された通信以外はブロックします。

NetworkPolicyの基本概念

- デフォルト： 全Pod間通信が許可（セキュリティリスク）
- NetworkPolicy： Pod間通信を制限するルール
- Ingress/Egress制御： 入出力の通信を個別に設定可能
- 柔軟な選択肢： Pod/Namespace/IPブロックで指定



適用前：すべてのPod間通信が許可（デフォルト）
セキュリティリスクが高い



適用後：必要な通信のみ許可

演習１ Kubernetes環境の診断と防御　解答編

前提知識：PSS（Pod Security Standards）

Podのセキュリティ設定を一定の基準で制御する仕組み。「危険なPod設定をクラスタに入れない」ためのチェック機構です。Namespace単位で適用可能です。

Privileged ⚠️	Baseline ✅	Restricted 🛡️
制限なし	一般的なワークロード向け	強固なセキュリティ
全機能にアクセス可能	既知の特権昇格を防止	非特権ユーザーのみ実行
セキュリティ上のリスクあり	利便性とセキュリティのバランス	ハードニングベストプラクティス
主な制約 制約なし - 特権昇格、ホストアクセスなど全て許可	**主な制約** 特権コンテナ禁止、ホストネームスペース不可、ホストパス不可、特定の機能のみ許可	**主な制約** Baselineの全制約＋特権昇格禁止、非rootユーザー必須、seccompプロファイル必須、権限厳格化

演習１　Kubernetes環境の診断と防御　解答編

演習１の進め方

脆弱なKubernetes環境に対して、Kubernetesの基本的な3つのセキュリティ対策を実装します。
対策前後でセキュリティが強化されることを確認しましょう。

1-0	**脆弱なK8s環境の構築**	広い権限、オープンな通信が可能なK8s環境を構築 K8s環境におけるセキュリティ上の問題点を確認
1-1	対策1 **ServiceAccount権限の最小化**	RBACを適用し、Podからの操作権限を最小化 他Podに対する操作を制限し、想定外な動作を排除
1-2	対策2 **Pod間通信の最小許可**	NetworkPolicyを適用し、Pod間通信を制限 許可されたPod通信のみ正常に機能
1-3	対策3 **PSS適用によるrootユーザ拒否**	PSSを適用し、Podの特権昇格を防止 ホストへのアクセスなどを制限

1-0：脆弱なK8s環境構築

解答編

演習１　Kubernetes環境の診断と防御　解答編

1-0：脆弱なK8s環境構築

脆弱なKubernetes環境を構築して、脆弱な環境であることを確認しましょう。
マニフェストを作成して、kubectlコマンドで各リソースを起動してください。

環境構築手順

※k8sクラスターは構築済の前提

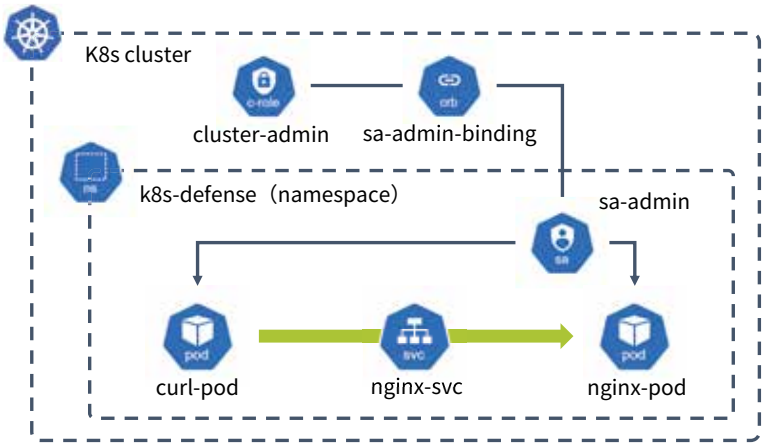
1. Namespace作成
   namespace : k8s-defense

1. ClusterRoleBinding/ClusterRole作成
   crb: sa-admin-binding
   c-role: cluster-admin

2. ServiceAccountの作成
   sa: sa-admin

1. curl Podの作成
   pod: curl-pod

2. nginx Pod/serviceの作成
   pod: nginx-pod
   service: nginx-svc



構築するk8s環境

演習１　Kubernetes環境の診断と防御　解答編

1-0：Namespace / ServiceAccount / ClusterRoleBinding のコード

Namespace、ServiceAccount、ClusterRoleBindingのマニフェストは以下の通りです。
sa-adminはクラスタ管理者権限を持っていて、適用したリソースに脆弱な権限を付与します。

```yaml
# Namespace
apiVersion: v1
kind: Namespace
metadata:
  name: k8s-defense
```

namespace-k8s-defense.yaml

```yaml
# ServiceAccount（過剰権限を持つ）
apiVersion: v1
kind: ServiceAccount

metadata:
  name: sa-admin
  namespace: k8s-defense
```

service-account-admin.yaml

```yaml
# ClusterRoleBinding（クラスタ管理者権限を付与）
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding

metadata:
  name: sa-admin-binding

subjects:
- kind: ServiceAccount
  name: sa-admin
  namespace: k8s-defense

roleRef:
  kind: ClusterRole
  # kubernetes標準で用意されているClusterRoleを指定
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

cluster-role-bindng-admin.yaml

演習１　Kubernetes環境の診断と防御　解答編

1-0：curl Podとnginx Pod/Serviceのコード

curl Pod、nginx Podおよびserviceのマニフェストは以下の通りです。
両Podには、脆弱なsa-adminを適用してください。

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: curl
 namespace: k8s-defense
 labels:
  app: curl

spec:
 # 脆弱なSAの適用
 serviceAccountName: sa-admin
 containers:
 - name: curl
  image: curlimages/curl
  command: ["sleep", "3600"]
```

curl-pod.yaml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: nginx
 namespace: k8s-defense
 labels:
  app: nginx

spec:
 # 脆弱なSAの適用
 serviceAccountName: sa-admin
 containers:
 - name: nginx
  image: nginx:latest
  ports:
  - containerPort: 80
```

nginx-pod.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
 name: nginx
 namespace: k8s-defense
spec:
 selector:
  app: nginx
 ports:
  - protocol: TCP
   port: 80
   targetPort: 80
```

nginx-service.yaml

演習１　Kubernetes環境の診断と防御　解答編

1-0：作業手順

K8s環境の構築手順は以下の通りです。
各マニフェストを準備して実行すれば、脆弱なK8s環境が構築されます。

作業手順

1. **Namespaceの作成**

   $ kubectl apply -f namespace-k8s-defense.yaml

2. **ServiceAccount、ClusterRoleBindingの作成**

   $ kubectl apply -f cluster-role-bindng-admin.yaml
   $ kubectl apply -f service-account-admin.yaml

3. **Curl Pod、Nginx Pod / Serviceの作成**

   $ kubectl apply -f curl-pod.yaml
   $ kubectl apply -f nginx-pod.yaml
   $ kubectl apply -f nginx-service.yaml

マニフェスト一覧

Namespace
- namespace-k8s-defense.yaml

ServiceAccount / ClusterRoleBinding
- cluster-role-bindng-admin.yaml
- service-account-admin.yaml

Pod / Service
- curl-pod.yaml
- nginx-pod.yaml
- nginx-service.yaml

演習１　Kubernetes環境の診断と防御　解答編

1-0：動作確認

K8s環境の構築が完了したら、各リソースを起動して、動作確認しましょう。

動作確認手順

1. **curl Podからnginx Podへのアクセス確認**
   curl Podからcurlコマンドを実行し、nginx のHTMLが表示されること
   $ kubectl exec -n k8s-defense -it curl -- curl -m 3 http://nginx

2. **sa-adminの操作権限確認**
   ServiceAccountの操作権限を確認し、podからget / delete できるか、権限を確認
   $ kubectl auth can-i get pods --as=system:serviceaccount:k8s-defense:sa-admin -n k8s-defense
   $ kubectl auth can-i delete pods --as=system:serviceaccount:k8s-defense:sa-admin -n k8s-defense

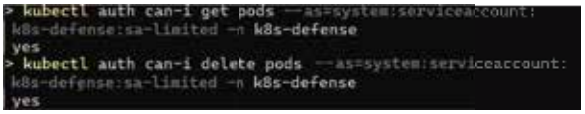
3. **sa-adminの操作権限一覧確認**
   ServiceAccountの操作権限を一覧で確認
   $ kubectl auth can-i --list --as=system:serviceaccount:k8s-defense:sa-admin

演習１　Kubernetes環境の診断と防御　解答編

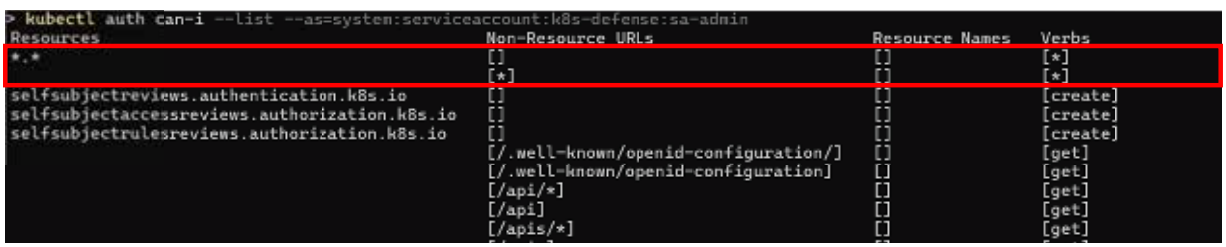
1-0：動作確認

1. **curl Podからnginx Podへのアクセス確認**
   nginxのHTMLが返却されます。

2. **sa-adminの操作権限確認**
   get pods → yes、delete pods → yes



3. **sa-adminの操作権限一覧確認**
   sa-adminには、**Resources: *.*　Verbs: [*]**、が付与され、どのリソースにも自由にアクセス可能な状態です。



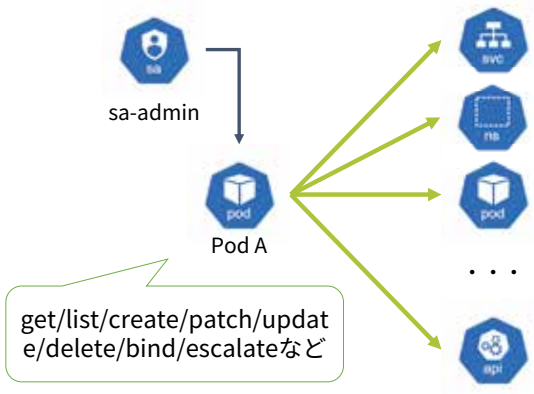
1-1： ServiceAccount権限の最小化

解答編

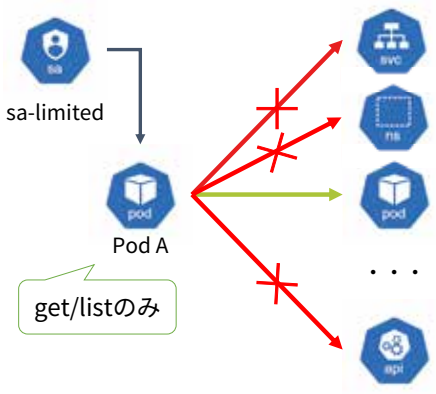
演習１ Kubernetes環境の診断と防御　解答編

1-1：期待される挙動

sa-adminは様々なリソースに様々な操作ができ、脆弱な状態でした。
sa-limitedに付け替え、Podへのget / listのみに絞ることがゴールです。



権限最大時（脆弱）



権限最小化時

演習１　Kubernetes環境の診断と防御　解答編

1-1：ServiceAccount権限の最小化

ServiceAccountの権限を最小化して、Pod操作が制限されることを確認しましょう。
マニフェストを作成して、kubectlコマンドで各リソースを起動してください。

検証手順

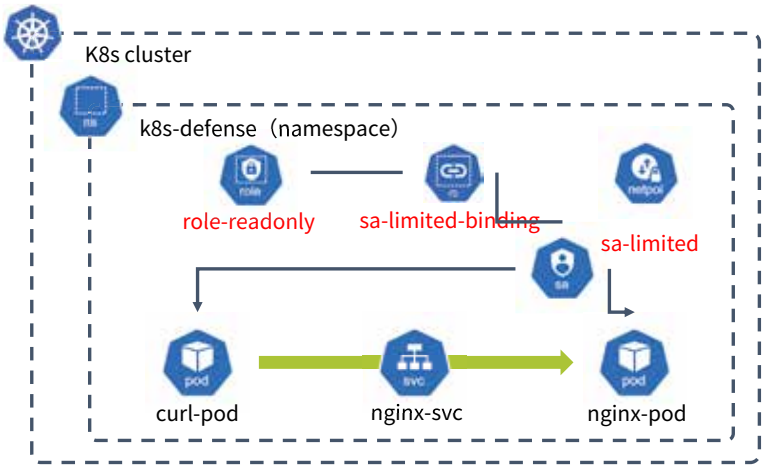
※k8sクラスターは構築済の前提

1. RoleBinding/Role作成
   rb: sa-limited-binding
   role: role-readonly

2. ServiceAccountの作成
   sa: sa-limited

1. curl Pod / nginx Podの修正
   SAをsa-adminからsa-limitedに変更

2. ServiceAccountの権限確認
   sa-limitedの操作権限が
   制限されていることを確認



構築するk8s環境

演習１　Kubernetes環境の診断と防御　解答編

1-1：ServiceAccount / ClusterRole / Roleのコード

ServiceAccount / ClusterRole / Roleのマニフェストは以下の通りです。
sa-limitedには、Podに対してget / list のみ許可したロールを付与します。

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sa-limited
  namespace: k8s-defense
```

service-account-limited.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: role-readonly
  namespace: k8s-defense
rules:
- apiGroups: [""]
  resources: [“pods”] # Podに対して
  verbs: [“get”, “list”] # GET, LISTのみ許可
```

role-readonly.yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: sa-limited-binding
  namespace: k8s-defense
subjects:
- kind: ServiceAccount
  name: sa-limited # SAを指定
roleRef:
  kind: Role
  name: role-readonly # roleを指定
  apiGroup: rbac.authorization.k8s.io
```

role-bindng-limited.yaml

演習１　Kubernetes環境の診断と防御　解答編

1-1：curl Podとnginx Pod/Serviceのコード

curl Pod、nginx Podのマニフェストは以下の通りです。
適用するServiceAccountを「sa-admin」から「sa-limited」に変更しています。

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: curl
 namespace: k8s-defense
 labels:
  app: curl

spec:
 # セキュアなSAに変更（sa-limited）
 serviceAccountName: sa-limited
 containers:
 - name: curl
  image: curlimages/curl
  command: ["sleep", "3600"]
```

curl-pod.yaml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: nginx
 namespace: k8s-defense
 labels:
  app: nginx

spec:
 # セキュアなSAに変更（sa-limited）
 serviceAccountName: sa-limited
 containers:
 - name: nginx
  image: nginx:latest
  ports:
  - containerPort: 80
```

nginx-pod.yaml

演習１　Kubernetes環境の診断と防御　解答編

1-1：作業手順

権限最小化に向けた手順は以下の通りです。
各マニフェストを準備して実行すれば、Podに対してセキュアなSAが適用されます。

K8s構築手順

1　既存Podの削除（デプロイしている場合）

```
$ kubectl delete -f curl-pod.yaml
$ kubectl delete -f nginx-pod.yaml
```

2　ServiceAccount、ClusterRoleBindingの作成

```
$ kubectl apply -f role-readonly.yaml
$ kubectl apply -f role-binding-limited.yaml
$ kubectl apply -f service-account-limited.yaml
```

3　Curl Pod、Nginx Pod の再作成

```
$ kubectl apply -f curl-pod.yaml
$ kubectl apply -f nginx-pod.yaml
$ kubectl apply -f nginx-service.yaml　※削除している場合
```

関連するマニフェスト一覧

ServiceAccount / RoleBinding / Role
- role-readonly.yaml
- role-bindng-limited.yaml
- service-account-limited.yaml

Pod / Service
- curl-pod.yaml
- nginx-pod.yaml
- nginx-service.yaml

演習１　Kubernetes環境の診断と防御　解答編

1-1：動作確認

K8s環境の構築が完了したら、各リソースを起動して、動作確認しましょう。
sa-adminと比較して、sa-limitedの権限が制限されていることを確認してください。

作業手順

1. **curl Podからnginx Podへのアクセス確認**
   curl Podからcurlコマンドを実行し、nginx のHTMLが表示されること（変化なし）
   $ kubectl exec -n k8s-defense -it curl -- curl -m 3 http://nginx

2. **sa-limitedの操作権限確認**
   ServiceAccountの操作権限を確認し、podからget/listのみ出来るか、権限の確認
   $ kubectl auth can-i get pods --as=system:serviceaccount:k8s-defense:sa-limited -n k8s-defense
   $ kubectl auth can-i delete pods --as=system:serviceaccount:k8s-defense:sa-limited -n k8s-defense

3. **sa-limitedの操作権限一覧確認**
   ServiceAccountの操作権限を一覧で確認し、権限が制限されていることを確認
   $ kubectl auth can-i --list --as=system:serviceaccount:k8s-defense:sa-limited

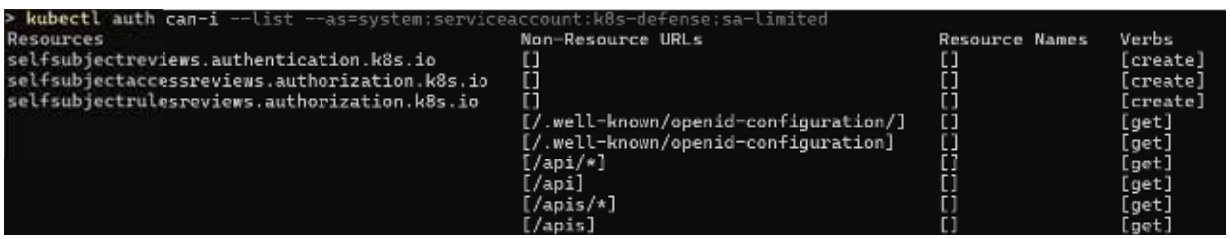
演習１　Kubernetes環境の診断と防御　解答編

1-1：動作確認

1. **curl Podからnginx Podへのアクセス確認**
   nginxのHTMLが返却されます。（変更なし）

2. **sa-limitedの操作権限確認**
   get pods → yes、delete pods → **no**

3. **sa-limitedの操作権限一覧確認**
   sa-limitedには、**Resources: *.*　Verbs: [*]、が存在せず**、各リソースに対する操作に制限が掛かっています。



演習１　Kubernetes環境の診断と防御　解答編

1-1：動作確認（追加）

4 Pod削除の実行結果（追加確認）

curl Podにexecして、curlコンテナからNginx Podを削除できるかを確認します。


```
$ kubectl exec -n k8s-defense -it curl -- sh
~ $ curl -sSk -X DELETE ¥
    -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" ¥
    https://kubernetes.default.svc/api/v1/namespaces/k8s-defense/pods/nginx ¥
    --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
```

```
$ kubectl get pod -n k8s-defense
NAME   READY  STATUS   RESTARTS  AGE
curl   1/1    Running  0         7s
nginx  1/1    Running  0         25m
$ kubectl exec -n k8s-defense -it curl -- sh # curl-podにexecでログイン
~ $ curl -sSk -X DELETE ¥
>     -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" ¥
>     https://kubernetes.default.svc/api/v1/namespaces/k8s-defense/pods/nginx ¥
>     --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt   # nginx-podを削除
{
  "kind": "Pod",
…省略…
"qosClass": "BestEffort"
  }
}
~ $ exit
$ kubectl get pod -n k8s-defense
NAME   READY  STATUS   RESTARTS  AGE
curl   1/1    Running  0         52s
```

sa-admin適用時（Pod削除成功）

```
$ kubectl get pod -n k8s-defense
NAME   READY  STATUS   RESTARTS  AGE
curl   1/1    Running  0         17m
nginx  1/1    Running  0         17m
$ kubectl exec -n k8s-defense -it curl – sh   # curl-podにexecでログイン
~ $ curl -sSk -X DELETE ¥
>     -H "Authorization: Bearer $(cat /var/run/secrets/kubernetes.io/serviceaccount/token)" ¥
>     https://kubernetes.default.svc/api/v1/namespaces/k8s-defense/pods/nginx ¥
>     --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt # nginx-podを削除
{
  "kind": "Status",
  "apiVersion": "v1",
…省略…
"code": 403
~ $ exit
$ kubectl get pod -n k8s-defense
NAME   READY  STATUS   RESTARTS  AGE
curl   1/1    Running  0         18m
nginx  1/1    Running  0         18m
```

sa-limited適用時（Pod削除失敗 - 403 Forbidden）

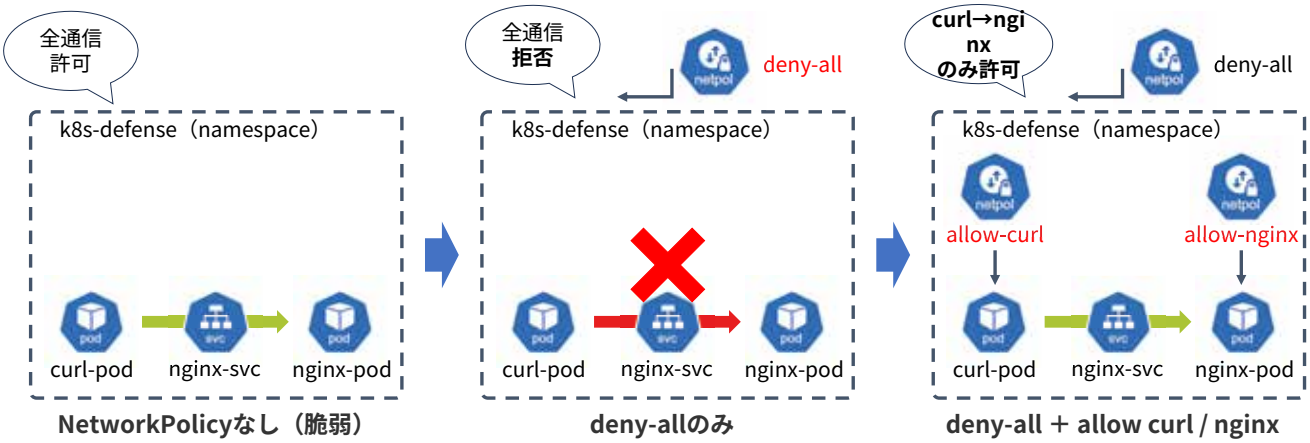
1-2：Pod間通信の最小許可

解答編

演習１　Kubernetes環境の診断と防御　解答編

1-2：期待される挙動

NetworkPolicyを2段階で適用し、Namespaceの通信を制御できることを確認しましょう。
全ての通信を閉塞した後、特定の通信経路だけ許可するのがベストプラクティスです。



演習１　Kubernetes環境の診断と防御　解答編

1-2： Pod間通信の最小許可

NetworkPolicyを設定し、Pod間の通信が制限されていることを確認しましょう。
マニフェストを作成して、kubectlコマンドで各リソースを起動してください。

検証手順

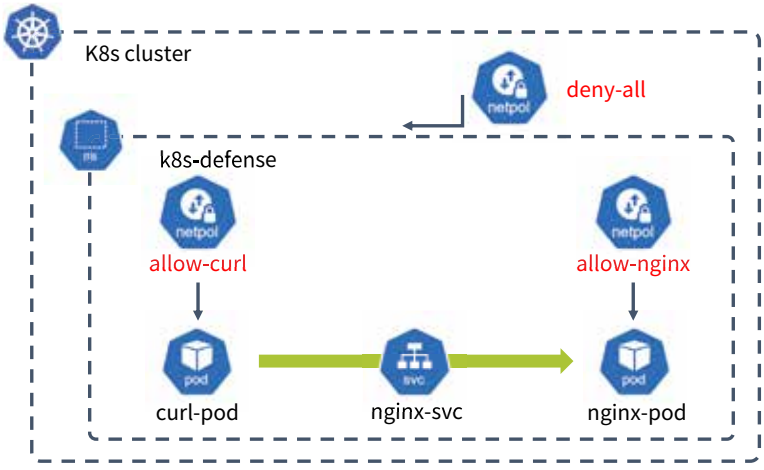
※SAはsa-limitedを適用したままでOKです

1. NetworkPolicy作成（Deny）
   Namespace内の全通信を拒否
   netpol: deny-all

2. NetworkPolicy適用後の通信可否

1. NetworkPolicy作成（Allow）
   netpol: allow-curl
   netpol: allow-nginx

2. NetworkPolicy適用後の通信可否



構築するk8s環境

演習１　Kubernetes環境の診断と防御　解答編

1-2：NetworkPolicyのコード

NetworkPolicy（deny-all）のマニフェストは以下の通りです。
NetworkPolicy適用後は、Namespace内の通信が全て拒否されます。

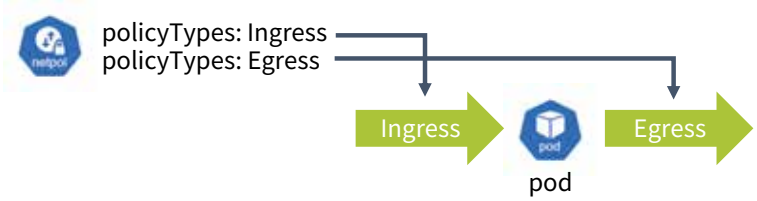
```
# deny-all policy
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: k8s-defense  #適用対象ns

spec:
  podSelector: {}
  policyTypes:
  - Ingress  # 入力
  - Egress   # 出力
```

network-policy-denyall.yaml

※Allow用のマニフェストはご自身で作成してください。

PolicyType（Ingress/Egress）

NetworkPolicyのIngressは入力、Egressは出力。
適用リソースの入出力をそれぞれ設定することで通信が可能に。



適用対象

演習１　Kubernetes環境の診断と防御　解答編

1-2：NetworkPolicyのコード

NetworkPolicy（allow-curl、allow-nginx）のマニフェストは以下の通りです。
NetworkPolicy適用後は、curl PodからNginx Podへの通信のみ許可されます。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-curl
  namespace: k8s-defense
spec:
  podSelector:
    matchLabels:
      app: curl
  policyTypes:
  - Egress

# 右に続く
```

nginxの名前解決のため、
DNSに問い合わせします
Egressに定義を追加

```
egress:
 # Nginxへの通信を許可
 - to:
   - podSelector:
       matchLabels:
         app: nginx
   ports:
   - protocol: TCP
     port: 80
 # DNS解決を許可
 - to:
   - namespaceSelector:
       matchLabels:
         kubernetes.io/metadata.name: kube-system
   ports:
   - protocol: UDP
     port: 53
```

network-policy-allowcurl.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-nginx
  namespace: k8s-defense
spec:
  podSelector:
    matchLabels:
      app: nginx
  policyTypes:
  - Ingress
  ingress:
  # curlからの通信を許可
  - from:
    - podSelector:
        matchLabels:
          app: curl
    ports:
    - protocol: TCP
      port: 80
```

network-policy-allownginx.yaml

演習１　Kubernetes環境の診断と防御　解答編

1-2：作業手順

NetworkPolicyの適用手順は以下の通りです。
各マニフェストを準備して実行すれば、Podに対してセキュアなSAが適用されます。

作業手順

1. **NetworkPolicyの適用（deny-all）**
   $ kubectl apply -f network-policy-denyall.yaml

2. **Curl Pod、Nginx Pod の作成**
   ※削除している場合
   $ kubectl apply -f curl-pod.yaml
   $ kubectl apply -f nginx-pod.yaml
   $ kubectl apply -f nginx-service.yaml

3. **NetworkPolicyの適用（allow-curl、allow-nginx）**
   ※deny-all適用後の動作確認が完了してから実行してください
   $ kubectl apply -f network-policy-allowcurl.yaml
   $ kubectl apply -f network-policy-allownginx.yaml

関連するマニフェスト一覧

NetworkPolicy
- network-policy-denyall.yaml
- network-policy-allowcurl.yaml
- network-policy-allownginx.yaml

Pod / Service
- curl-pod.yaml
- nginx-pod.yaml
- nginx-service.yaml

演習１　Kubernetes環境の診断と防御　解答編

1-2：動作確認

NetworkPolicy（deny-all）を適用したら、動作確認しましょう。
これでNamespace内の全通信が閉塞されます。

動作確認手順

1. **curl Podからnginx Podへのアクセス確認**
   curl Podからcurlコマンドを実行し、nginx に到達せずタイムアウトすること
   $ kubectl exec -n k8s-defense -it curl -- curl -m 3 http://nginx

実行結果（タイムアウト）

```
> kubectl exec -n k8s-defense -it curl -- curl -m 3 http://nginx
curl: (28) Resolving timed out after 3000 milliseconds
command terminated with exit code 28
```

演習１　Kubernetes環境の診断と防御　解答編

1-2：動作確認②

※NetworkPolicy（deny-all）は設定したままです。
続いて、NetworkPolicy（allow-curl、allow-nginx）を作成して、動作確認しましょう。
curl Pod、nginx Podの入出力を設定することで、curl-pod→nginx-podのみ通信が可能になります。

動作確認手順

1. **curl Podからnginx Podへのアクセス確認**
   curl Podからcurlコマンドを実行し、nginx のHTMLが表示されること
   $ kubectl exec -n k8s-defense -it curl -- curl -m 3 http://nginx

実行結果

```
> kubectl exec -n k8s-defense -it curl -- curl -m 3 http://nginx
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```

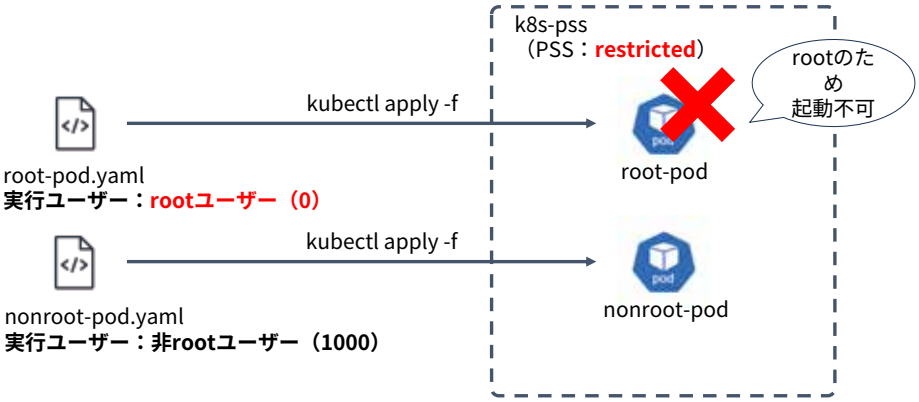
1-3：PSS適用によるrootユーザ拒否

解答編

演習１　Kubernetes環境の診断と防御　解答編

1-3：期待される挙動

k8s-pss namespaceにPSS：restrictedを適用すると、rootユーザーは実行不可となります。
root-podは起動できませんが、nonroot-podであれば起動できるのを確認しましょう。



演習１　Kubernetes環境の診断と防御　解答編

1-3： PSS適用によるrootユーザ拒否

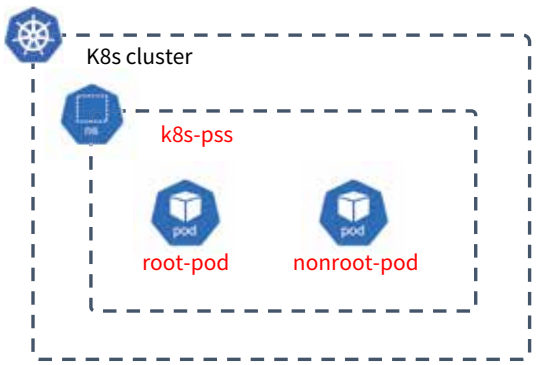
PSSを適用したNamespaceを用意し、rootユーザーのPodが起動できないことを確認します。
マニフェストを作成して、kubectlコマンドで各リソースを起動してください。

検証手順

1. 新規Namespace作成（PSS設定有）
   ns: k8s-pss

2. podの作成
   pod: root-pod
   pod: nonroot-pod

3. Podの起動可否を確認



構築するk8s環境

演習１　Kubernetes環境の診断と防御　解答編

1-3：Namespaceのコード

Namespaceとroot-podのマニフェストは以下の通りです。
Namespaceにrestrictedが適用されると、rootユーザーでの起動はできなくなります。

```yaml
# Namespace（PSS適用）
apiVersion: v1
kind: Namespace
metadata:
 name: k8s-pss
 labels:
  # enforce:restrictedでrootは起動不可
  pod-security.kubernetes.io/enforce: restricted
  # audit:baselineは一般的な制限
  pod-security.kubernetes.io/audit: baseline
```

namespace-k8s-pss.yaml

PSSのレベル

⚠️ **Privileged（特権）**
あらゆる操作を許可する最も緩いモードで、セキュリティ制限がほとんどなく管理者向けのデバッグ用途に使われます。

✅ **Baseline（ベースライン）**
一般的なアプリケーション運用で安全に動作する最低限の制限を設け、明確なリスクとなる特権操作を禁止します。

🛡️ **Restricted（制限）**
セキュリティを最優先し、root実行や特権昇格、危険なシステムコールを完全に防ぐ厳格な設定を求めます。

演習１　Kubernetes環境の診断と防御　解答編

1-3：root Podとnonroot Podのコード

root Pod、nonroot Podのマニフェストは以下の通りです。
それぞれ、rootユーザー：0、非rootユーザー：1000、で起動するよう定義しています。

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: root-pod
 namespace: k8s-pss
spec:
 containers:
 - name: root
  image: busybox
  command: ["sh", "-c", "sleep 3600"]
  securityContext:
   runAsUser: 0  # rootユーザー
```

root-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: restricted-pod
 namespace: k8s-pss
spec:
 securityContext:
  runAsNonRoot: true     # 明示的にroot不可
  seccompProfile:
   type: RuntimeDefault  # Pod全体に適用
 containers:
 - name: app
  image: busybox
  command: ["sh", "-c", "sleep 3600"]
  securityContext:
   allowPrivilegeEscalation: false
   capabilities:
    drop: ["ALL"]
```

nonroot-pod.yaml

演習１　Kubernetes環境の診断と防御　解答編

1-3：作業手順

PSSの適用手順は以下の通りです。
PSSで厳格化されたNamespaceが作成され、root実行のPodは起動できなくなります。

作業手順

1. **Namespaceの作成**
   $ kubectl apply -f namespace-k8s-pss.yaml
2. **Root Pod、Nonroot Pod の作成**
   $ kubectl apply -f root-pod.yaml
   $ kubectl apply -f nonroot-pod.yaml

関連するマニフェスト一覧

Namespace
- namespace-k8s-pss.yaml

Pod / Service
- root-pod.yaml
- nonroot-pod.yaml

演習１　Kubernetes環境の診断と防御　解答編

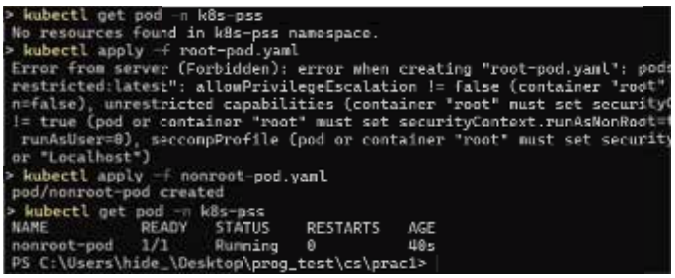
1-3：動作確認

root-pod、nonroot-podのマニフェストが準備出来たら、動作確認しましょう。
rootユーザーの拒否により、セキュリティが強化されました。

動作確認手順

1. **root Podの起動**
   root Podの起動に失敗
   $ kubectl apply –f root-pod.yaml
2. **nonroot Podの起動**
   nonroot Podの起動は成功
   $ kubectl apply –f nonroot-pod.yaml

実行結果



演習１　Kubernetes環境の診断と防御　解答編

まとめ：対策効果とベストプラクティス

RBAC：アクセス制御

対策効果：ServiceAccountに必要最小限の権限のみ付与
- 攻撃者がPodを侵害しても、操作可能範囲が限定的になる

ベストプラクティス
- 最小権限の原則を徹底する
- Role/ClusterRoleは対象（resource）と操作（verb）を明示的に限定
- kubectl auth can-iで定期的に権限を棚卸し、不要権限を削除

> RBACはK8s環境におけるアクセス制御の基本。適切に設定されていないとクラスタ全体のセキュリティが脆弱になる

NetworkPolicy：通信制御

対策効果：必要な通信のみを許可し、不要な通信を遮断
- Pod間の横方向の攻撃（横展開）を防止

ベストプラクティス
- default-deny（全拒否）を前提に、必要な通信のみ開放
- Ingress/Egressを明示的に定義
- DNSや外部依存サービスへの通信も忘れずに許可

> 特にマルチテナント環境では、Namespace間の分離をNetworkPolicyで徹底することが重要

演習１　Kubernetes環境の診断と防御　解答編

まとめ：対策効果とベストプラクティス

PSS：Pod権限制限

対策効果：特権実行を制限し、ホストへの権限昇格を防止
- コンテナからホストへの攻撃リスクを大幅に低減

ベストプラクティス：
- Namespace単位でrestrictedを基本に設定
- 非rootユーザー実行を強制
- 特別な要件がある場合のみbaseline等を使い分け

> Kubernetes 1.25以降ではPodSecurityを活用。以前の環境ではPodSecurityPolicyを適宜検討

演習２ 安全なコンテナ構築とランタイム防御

解答編

演習２　安全なコンテナ構築とランタイム防御　解答編

演習概要

コンテナイメージ・レジストリの堅牢化とランタイムを保護するセキュリティ対策を実装し、現実的かつ効果的なコンテナ防御について理解を深めます。

Before：脆弱なコンテナ

- **脆弱なイメージ**
  不要なパッケージが多く、脆弱性を多く含むイメージは、Pod侵害される可能性が高まります。
- **署名なし**
  サプライチェーンやイメージの検証が実施されない場合、改ざんなどの検知が遅れます。
- **ランタイム制御なし**
  ホストシステムへのアクセスやプロセスの乗っ取りなどにより、システム全体に影響を及ぼします。

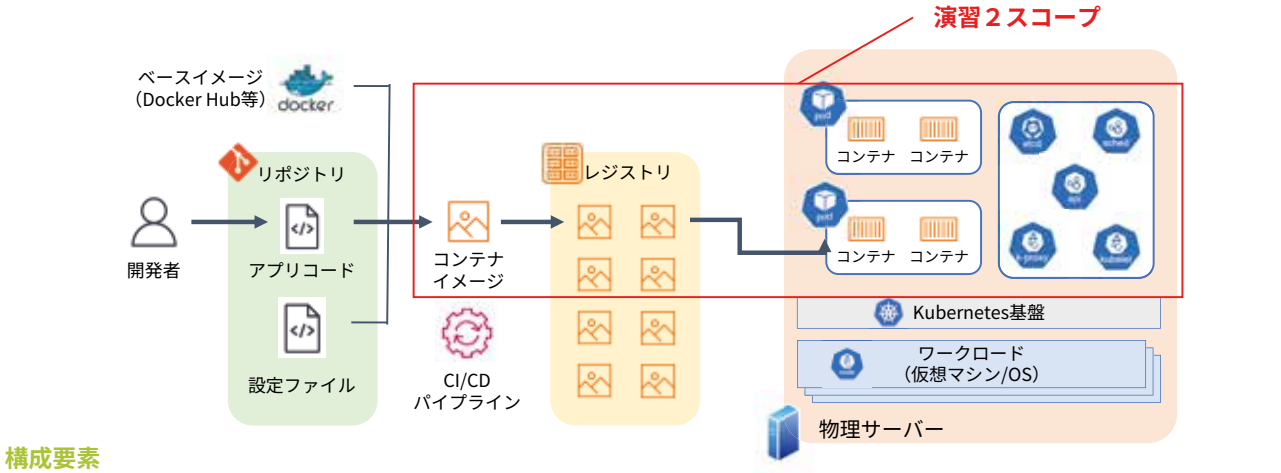
After：安全な構成

- **軽量化＋非root化**
  脆弱性が減り、コンテナの軽量化にも繋がり、セキュリティリスクが軽減されます。
- **署名と検証**
  イメージ署名とデプロイ前確認により、イメージの正しさを確認し、改ざんを検知できます。
- **権限制限**
  seccomp/AppArmorプロファイル適用により、不要な攻撃から防御できます。

演習２　安全なコンテナ構築とランタイム防御　解答編

演習２の位置づけ

演習2ではイメージ、レジストリ、コンテナをターゲットにセキュリティ対策を行います。



構成要素

①コード　②イメージ　③レジストリ　④オーケストラ　⑤コンテナ　⑥ホスト

演習２　安全なコンテナ構築とランタイム防御　解答編

前提知識：脆弱性診断（Trivy）

イメージやファイルシステムなどの脆弱性を診断し、OSパッケージやアプリケーションの依存関係などを包括的に診断するツールです。デプロイ前にセキュリティリスクを検出できれば、事前に対処できます。

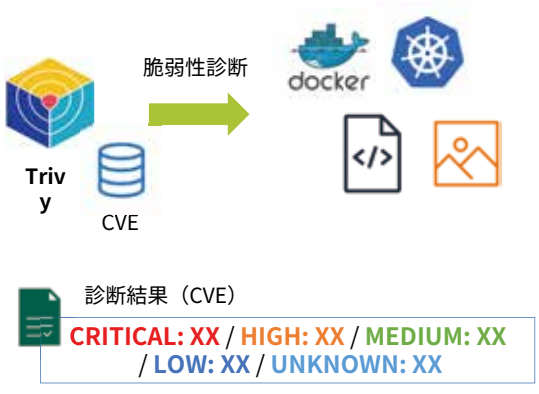
脆弱性診断の必要性

- 本番環境にデプロイする前にセキュリティ上の問題を検出
- 既知の脆弱性を含むイメージがデプロイされるリスクを低減
- 攻撃者に悪用される前に修正を適用する機会を確保

Trivyについて

演習2は、Trivyを使って脆弱性診断を行います。

- オープンソースの脆弱性診断ツール
- イメージ、ファイルシステムなどマルチスキャンが可能
- CVEベースで重大度を確認可能（Critical / Highなど）
- CI/CDパイプラインと連携可能
- 多彩な形式でレポート出力が可能



その他ツール

- Grype / Clair / Docker Scoutなど

演習２　安全なコンテナ構築とランタイム防御　解答編

前提知識：イメージ署名と検証（Cosign / Kyverno）

コンテナイメージにデジタル署名することで、イメージの改ざん防止と信頼性を保証できます。
「誰がビルドしたか」「改変されていないか」を署名と検証で確認することができます。

イメージ署名と検証の必要性

- コンテナイメージの改ざんリスクに備える
- 署名：ビルド済みイメージが正規のものか署名を付与
- 検知：正規のイメージ化判断して、環境にデプロイ

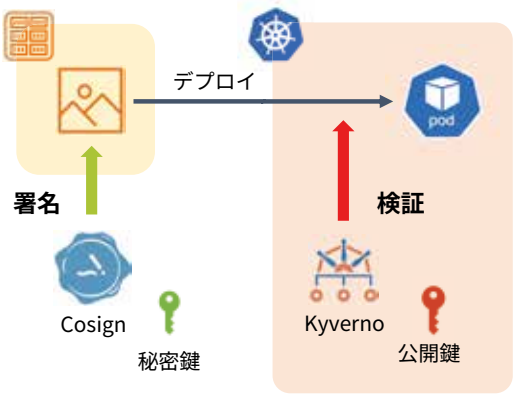
演習2では、Cosign（署名）＋Kyverno（検証）を組み合わせ、
信頼できるイメージのみ動作するクラスターを実現します。

Cosign

- イメージにデジタル**署名を付与**するツール
- 署名データはレジストリ（ECRなど）に保存
- SBOMも署名可能。

Kyverno

- KubernetesのAdmission Controllerとして動作
- デプロイ時に署名付きイメージのみ許可
- Cosign署名をポリシーで自動検証



Cosign＋Kyverno のアーキテクチャ

演習２　安全なコンテナ構築とランタイム防御　解答編

前提知識：ランタイムセキュリティ制御

ランタイム実行時の攻撃面を最小化し、侵害時の影響を局所化するために導入します。
コンテナ実行時のカーネル、ファイルシステム、ネットワークへのアクセスを適切に制限します。

seccomp

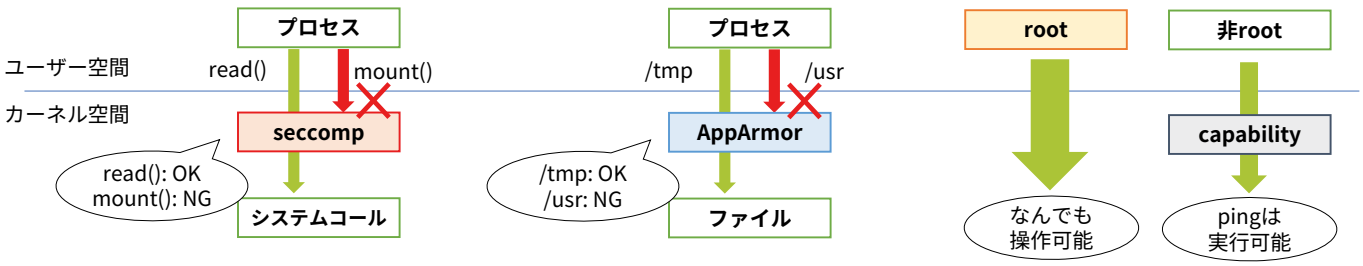
「何を呼び出せるか」を制御
- カーネルが提供する
  システムコールフィルタ
- コンテナがOSに対して呼び出せる
  「システムコール」を制限

AppArmor

「何にアクセスできるか」を制御
- Ubuntuなどで使われるプロセス単位のアクセス制御機構
- ファイル読み書き・ネットワーク・シグナル送信などを制限

capability

「何を使っていいか」を制御
- Linuxのカーネル権限を細かく分割
- 通常rootでしかできない操作を個別に付与・削除可能
- 例：CAP_NET_RAW → ping可



演習２　安全なコンテナ構築とランタイム防御　解答編

演習２の進め方

コンテナイメージ・レジストリの堅牢化とランタイムを保護するセキュリティ対策を実装し、現実的かつ効果的なコンテナ防御について理解を深めます。

2-1	対策1 **Trivyによる脆弱性診断**	Trivyでイメージをスキャンして、脆弱性を可視化 Dockerfileを修正し、脆弱性とサイズ肥大化に対処
2-2	対策2 **Cosignによる署名と検証**	Cosignによるイメージ署名および検証を導入
2-3	対策2-2 **Cosign＋Kyvernoによる検証**	Kyvernoで署名済イメージのみクラスターにデプロイ可能
2-4	対策3 **ランタイム保護による操作制御**	seccomp/capabilities削減/AppArmorの適用により、 ランタイムを保護し、システムコールの利用を制限

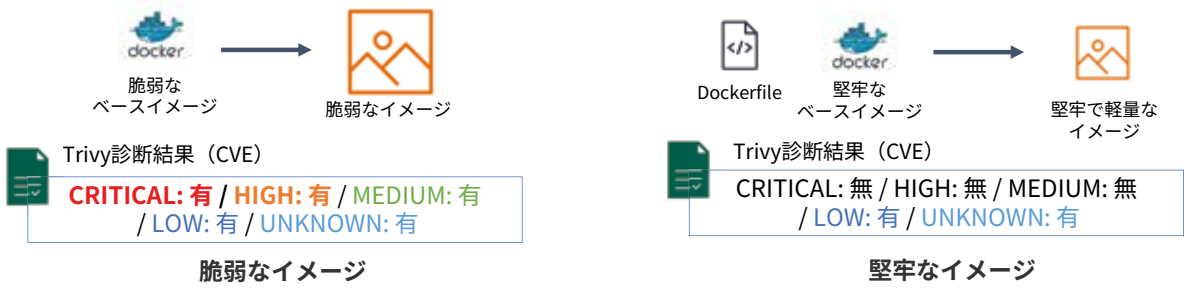
2-1：Trivyによる脆弱性診断

解答編

演習２　安全なコンテナ構築とランタイム防御　解答編

2-1：期待される挙動

Trivyを用いてコンテナイメージの脆弱性を診断し、セキュリティリスクを可視化します。
その後、Dockerfileで堅牢なイメージを作成し、脆弱性の低減と軽量化を実現します。



イメージの修正ポイント

- CRITICAL/HIGH件数: 重大な脆弱性の対処
- 脆弱なパッケージ: 不要なライブラリの削減
- サイズ: イメージサイズの軽量化（数百MB→100MB程度以下）
- ベースイメージ: 脆弱性の少ないベースの選定（alpine、distroless）
- 非root起動、権限制御

演習２　安全なコンテナ構築とランタイム防御　解答編

2-1：Trivyによる脆弱性診断

Trivyをインストールし、イメージの脆弱性診断を実行し、セキュリティリスクを可視化します。
続いてDockerfileで堅牢なイメージを作り、脆弱性の低減を確認しましょう。

演習2-1 手順

1	**Trivyのインストール**	Trivyを作業インスタンス（ローカル環境やCloudShellなど）にインストール
2	**脆弱なイメージのスキャン**	NginxのイメージをTrivyでスキャンし、セキュリティリスクを可視化
3	**堅牢なイメージの作成・診断**	Dockerfileで、ベースイメージ変更、非root化により、堅牢なイメージを作成し、脆弱性の低減、サイズ縮小を確認

演習２　安全なコンテナ構築とランタイム防御　解答編

2-1：Trivyのインストール

Trivyをローカル環境もしくはCloudShellにインストールしてください。

インストール手順

1 **Trivyのインストール**

Trivyをインストールしてください。

```
$ curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sudo sh -s -- -b /usr/local/bin
$ trivy version
```

2 **Trivyの実行準備（CloudShellの場合）**

⚠Trivyの実行時、CVEデータベースを構築するのに約1GB必要です。

⚠CloudShellの場合、home領域は容量不足となるため、暫定的に/tmpに保存する手順を載せます。

```
$ mkdir -p /tmp/trivy-db
$ export TRIVY_CACHE_DIR=/tmp/trivy-db
```

演習２　安全なコンテナ構築とランタイム防御　解答編

2-1：脆弱なイメージのスキャン

Trivyをインストール出来たら、nginx:latestをスキャンしてください。
nginx:latestが脆弱なイメージであることを確認しましょう。

動作確認手順

1 **Nginxのイメージスキャン**

Trivyでnginx:latestのイメージスキャンを行ってください。

```
$ docker pull nginx:latest
$ trivy image nginx:latest
```

実行結果

Trivyのスキャンが完了し、脆弱性診断の結果を確認しましょう。

⚠CRITICAL / HIGHの脆弱性を含むことを確認できればOK

HIGH：1件を含む、計93件を検出しました。

演習２　安全なコンテナ構築とランタイム防御　解答編

2-1：堅牢なイメージの作成・診断

Dockerfileを作成し、堅牢なイメージを作成してください。
作成したイメージをスキャンし、脆弱性およびイメージサイズの削減を確認してください。

作業手順

1　Dockerfileの作成

Dockerfileを作成してください。ただし以下要件を満たすこと。
・簡易なアプリケーションを実装（print関数、1行程度でOK）
・脆弱性の少ないイメージを選択（python：3.12-alpineなど）
・非rootで起動

2　ビルドしたイメージのスキャン

docker build / tagで作成したイメージをスキャンし、脆弱性の低減を確認してください。

```
$ docker build -t cs-app:latest .          # 本解答では、cs-app:latest でイメージを作成します。
$ trivy image cs-app:latest
```

3　イメージサイズの確認

ビルドしたイメージサイズを確認し、nginx:latest に比べて小さくなっていれば良いです。

```
$ docker image ls
```

演習２　安全なコンテナ構築とランタイム防御　解答編

参照）2-1：Dockerfileのコード

Dockerfileのサンプルコードは以下の通りです。
独自のアプリケーションや別イメージを準備頂いても構いません。

```
FROM python:3.12-alpine

# ユーザー追加
RUN adduser -D appuser

# ワークディレクトリ
WORKDIR /app

# 簡易アプリ
COPY <<'APP' app.py
print("Hello Secure World!")
APP
USER appuser

# 起動時にメッセージを表示してから待機
CMD ["sh", "-c", "python3 app.py && sleep infinity"]
```

Dockerfile

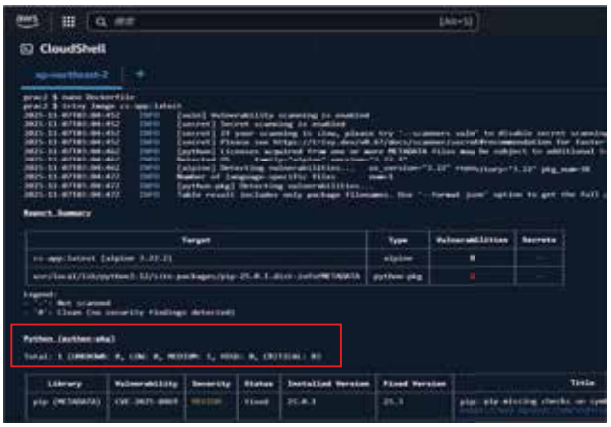
演習２　安全なコンテナ構築とランタイム防御　解答編

2-1：動作確認

② ビルドしたイメージのスキャン

cs-app:latest（python:3.12-alpineベース）
の診断結果は、**MEDIUM：1件のみ**。

重大な脆弱性が無いことを確認できました。



③ イメージサイズの確認

Nginxのサイズが152MBに対して、ビルドイメージは48.7MBでした。
これで軽量なイメージを作成できました。alpineやdistrolessは軽量なのでベースイメージを選ぶ際に有用です。

```
prac2 $ docker image ls
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
cs-app       latest    babaaa221c04   19 minutes ago   48.7MB
nginx        latest    d261fd19cb63   2 days ago       152MB
```

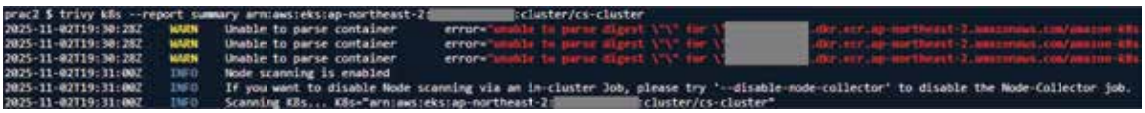
演習２　安全なコンテナ構築とランタイム防御　解答編

2-1：動作確認（追加）

Trivyはイメージ以外にも、マニフェストやKubernetesクラスターなどの脆弱性診断も可能です。
ご自身の環境でも実施してみてください。

マニフェストの脆弱性診断（trivy config <マニフェスト.yaml>）

Kubernetesクラスターの脆弱性診断（trivy k8s --report summary <k8sクラスターのコンテキスト>）



演習２　安全なコンテナ構築とランタイム防御　解答編

2-1：ECRを利用する場合

ECRを利用する場合は、リポジトリを用意してからイメージをpushしてください。
ECRは、2-2、2-3のイメージ署名と検証でも利用しますので、ここで準備しておきましょう。

ECR利用準備

1. **AWS ECRへのログイン**

```
$ aws ecr get-login-password --region $REGION ¥
  | docker login --username AWS --password-stdin ¥
    $(aws sts get-caller-identity --query Account --output text).dkr.ecr.$REGION.amazonaws.com
```

2. **リポジトリの作成**

⚠ECRのGUIから作成することも可能です。イメージの自動スキャンは有効化しましょう。

```
$ aws ecr create-repository ¥
  --repository-name cs-app --region "$REGION"
  --image-scanning-configuration scanOnPush=true
```

3. **イメージのプッシュ**

```
$ docker tag cs-app:latest xxx.dkr.ecr.ap-northeast-1.amazonaws.com/cs-app:latest
$ docker push xxx.dkr.ecr.ap-northeast-1.amazonaws.com/cs-app:latest
```

2-2：Cosignによる署名と検証

解答編

演習２　安全なコンテナ構築とランタイム防御　解答編

2-2：Cosignによる署名と検証

Cosignをインストールして、イメージに署名します。
署名されたイメージを検証し、署名されていることを確認しましょう。
⚠ローカル環境の場合はローカル、CloudShellの場合はECRに保存されたイメージに署名します。

演習2-2 手順

1. **Cosignのインストール**
   Cosignを作業インスタンスにインストール
   暗号鍵・公開鍵を作成
2. **イメージ署名**
   Cosignでローカル or ECRのイメージに署名
3. **イメージ検証**
   Cosignで署名したイメージを検証し、
   署名済みイメージであることを確認

Cosign：v2.4.1

演習２　安全なコンテナ構築とランタイム防御　解答編

2-2：Cosignのインストール

Cosignをローカル環境もしくはCloudShellにインストールしてください。

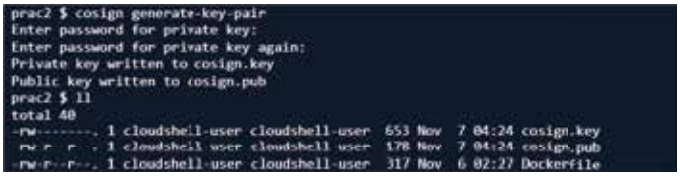
インストール手順

1. **Cosignのインストール**
   Cosignをインストールしてください。
   ⚠Cosignは**v2.4.1**を利用します。

```
$ curl -LO "https://github.com/sigstore/cosign/releases/download/v2.4.1/cosign-darwin-arm64"
$ chmod +x cosign-darwin-arm64
$ sudo mv cosign-darwin-arm64 /usr/local/bin/cosign
$ cosign version
```

2. **秘密鍵・公開鍵の作成**
   Cosignで秘密鍵・公開鍵を作成します。
   ⚠鍵作成時にパスワードを設定します。
   後程イメージに署名する際に利用します。

```
$ cosign generate-key-pair
$ ls -la cosign.*
```



演習２　安全なコンテナ構築とランタイム防御　解答編

2-2：イメージの署名

Cosignでイメージに署名しましょう。
CloudShellの場合は、事前にECRにイメージをpushし、ECR上のイメージに署名します。

イメージ署名の手順

1 **イメージ準備**

署名するイメージを準備してください。
演習２-1で作成したイメージでも大丈夫です。
⚠CloudShellの場合は、ECRに格納してください。

2 **イメージの署名**

Cosignでイメージに署名してください。鍵作成時のパスワード入力が必要です。

ローカルイメージに署名する場合（例：cs-app:latest）
$ cosign sign --key cosign.key cs-app:latest

ECR上のイメージに署名する場合
$ cosign sign --key cosign.key xxx.dkr.ecr.ap-northeast-1.amazonaws.com/cs-app:latest

演習２　安全なコンテナ構築とランタイム防御　解答編

2-2：イメージの検証

それでは、Cosignでイメージを検証し、正しく署名されているかを確認しましょう。

イメージ検証の手順

1 **イメージの検証**

Cosignでイメージを検証しましょう。

ローカルイメージに署名する場合（例　myapp:latest）
$ cosign verify --key cosign.pub myapp:latest

ECR上のイメージに署名する場合
$ cosign verify --key cosign.pub <ECRリポジトリURI>/myapp:latest

実行結果

以下、メッセージが出力されていれば、正しく署名されています。

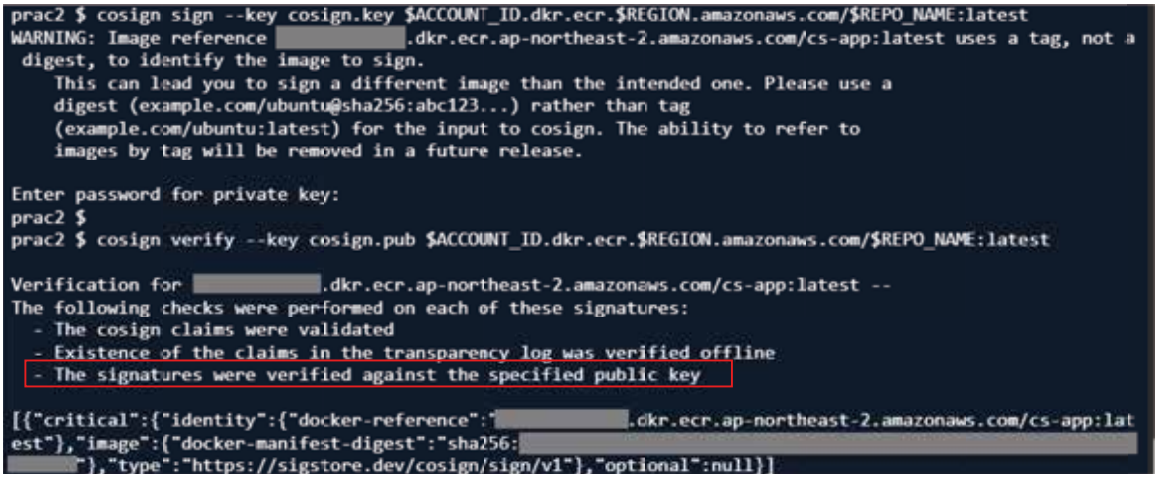
```
- The signatures were verified against the specified public key
```

演習２　安全なコンテナ構築とランタイム防御　解答編

2-2：動作確認（署名と検証）

1 イメージの署名 / イメージ検証

Cosignでの署名したイメージ検証し、イメージの正常性が確認できました。



2-3：Cosign＋Kyvernoによる検証

解答編

演習２　安全なコンテナ構築とランタイム防御　解答編

2-3：Cosign＋Kyvernoによる検証

KubernetesクラスターにKyvernoをインストールして、署名されたイメージのみデプロイできることを検証しましょう。

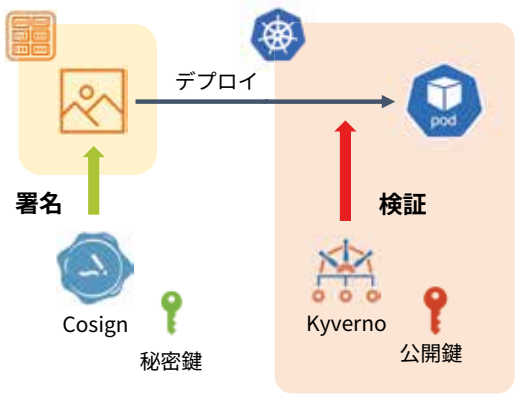
Cosign：v2.4.1
Kyverno：v3.4.4

演習2-3 手順

1. **Kyvernoのインストール**
   KyvernoをKubernetesクラスターにインストール

2. **Kyvernoの署名検証ポリシー設定**
   Kyvernoの署名検証ポリシーを設定し、
   署名のないイメージはデプロイ拒否の状態にする

3. **デプロイ検証**
   Kubernetesクラスターへのデプロイを通じて、
   署名の有無による動作の違いを確認



Cosign＋Kyverno のアーキテクチャ

演習２　安全なコンテナ構築とランタイム防御　解答編

2-3：Kyvernoのインストール

KyvernoをKubernetesクラスターにインストールしてください。
Helmでのインストールを想定していますが、curlなどを利用しても構いません。

インストール手順

1. **Helmの設定**
   Helm経由でKyvernoをインストールするため、Helmを設定します。

```
$ curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
$ helm version
$ helm repo add kyverno https://kyverno.github.io/kyverno/
$ helm repo update
```

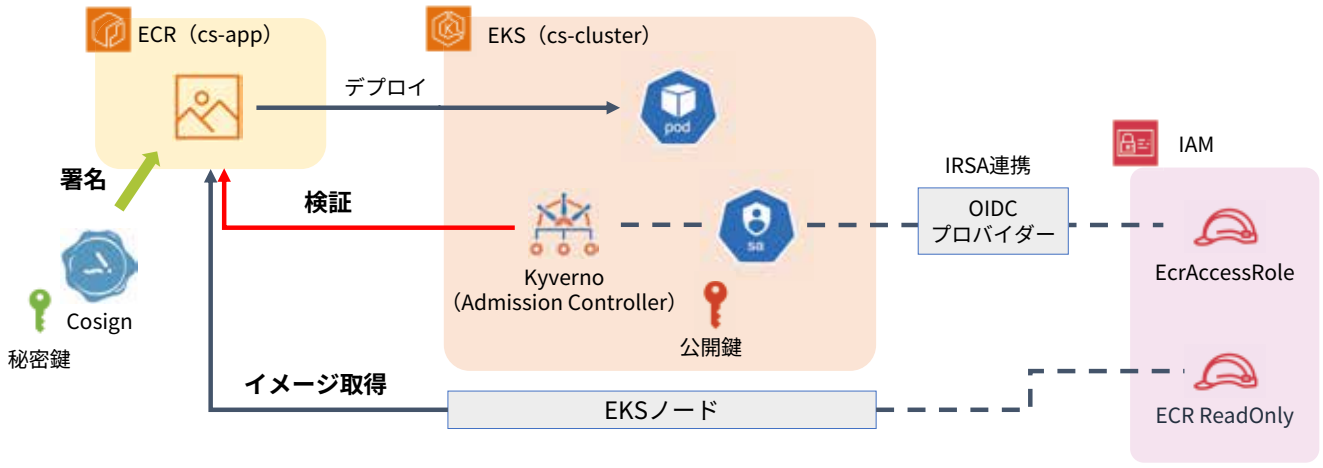
1. **Kyvernoのインストール**
   Namespace kyvernoを作成してから、Kyvernoをインストールしてください。
   ⚠kyvernoは**v3.4.4**を利用します。

```
$ kubectl create namespace kyverno
$ helm install kyverno kyverno/kyverno  --namespace kyverno  --version v3.4.4 ¥
  --set installCRDs=true  --set admissionController.generateSelfSignedCert=true
$ kubectl get pod -n kyverno
```

演習２　安全なコンテナ構築とランタイム防御　解答編

2-3：EKS環境設定

ECRに登録したイメージをEKSでデプロイする場合、EKSのリソースにECRを操作するためのIAMロールを付与する必要があります。ECR＋EKSの構成の場合は、SA / IAM / OIDC / IRSA連携も設定してください。（難易度：高）



演習２　安全なコンテナ構築とランタイム防御　解答編

2-3：EKS環境設定

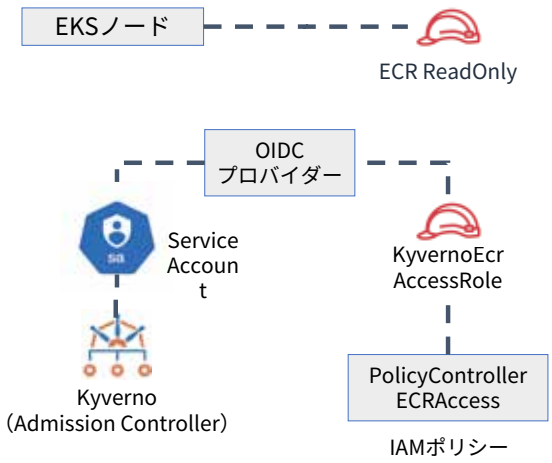
EKSノードおよびKyvernoからECRにアクセスし読み取り可能とするためにはAWS側の設定が必要です。環境構築の流れを説明します。ECR＋EKSを利用している場合は、以下を実施しましょう。

① EKSノードへのIAMポリシー追加
- EKSノードにECR読み取り権限を付与

② Kyverno用 IAMポリシーの作成
- IAMポリシーを作成し、ECRの読み取り権限を設定
- OIDCプロバイダーを作成、K8sクラスターに紐付け

③ Kyverno用 IAMロールの作成
- ServiceAccountの信頼ポリシーを作成
- IAMロールを作成し、IAMポリシーをアタッチ

④ Kyverno へのIAMロールの適用
- ServiceAccountを作成し、OIDC連携
- Kyvernoを再起動してSAを適用



演習課題４　解答

（参考）eksctlインストール

AWS環境の設定にあたり、eksctlを利用します。必要に応じてインストールしてください。
CloudShellでの実行コマンドは以下です。

Eksctlのインストール

```
#1. 最新版の eksctl をダウンロード
curl --silent --location¥
  "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" ¥
  | tar xz -C /tmp

#2. 実行パスへ移動（CloudShellユーザー領域）
sudo mv /tmp/eksctl /usr/local/bin

#3. 実行権限を付与
sudo chmod +x /usr/local/bin/eksctl

#4. バージョン確認
eksctl version
```

演習２　安全なコンテナ構築とランタイム防御　解答編

2-3：EKS環境設定手順①

EKSノードおよびKyvernoからECRにアクセスし読み取り可能とするために、AWS側の設定をします。
まずEKSノードのIAMロールにECRの読み取り権限を付与します。

EKSノードへのIAMポリシー追加

1　ノードの IAM ロール確認

ノードのIAMロールを確認します。

$ eksctl get nodegroup --cluster cs-cluster
　→Kubernetesクラスターの**<NODEGROUP>**を確認

出力例

```
prac2 $ eksctl get nodegroup --cluster cs-cluster
CLUSTER         NODEGROUP       STATUS  CREATED
cs-cluster      lab-nodes       ACTIVE  2025-10-28T
```

```
$ aws eks describe-nodegroup --cluster-name cs-cluster  --nodegroup-name <NODEGROUP> ¥
 --region ap-northeast-1 ¥  --query "nodegroup.nodeRole" --output text
  →ノードに付与されているIAMロール（出力例の太字部分）を確認
    出力例：　arn:aws:iam::<ACCOUNT_ID>:role/eksctl-cs-cluster-nodegroup-lab-no-NodeInstanceRole-XXXXXXXXX
```

2　ECR読み取り専用ポリシーの付与

ノードのIAMロールに既存のECR読み取り専用ポリシーを付与します。

```
$ aws iam attach-role-policy  --role-name <IAMロール名> ¥
 --policy-arn arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly
```

演習２　安全なコンテナ構築とランタイム防御　解答編

2-3：EKS環境設定手順②

EKSノードおよびKyvernoからECRにアクセスし読み取り可能とするために、AWS側の設定をします。
次はKyvernoがECRにアクセスするためのIAMポリシーを作成します。

Kyverno用 IAMポリシーの作成

3　Kyverno用の IAM ポリシーの作成

IAMポリシーを作成し、ECRの読み取り権限を設定します。

```
$ nano ecr-read-policy.json　# IAMポリシー用のJSONファイルを準備（次ページ参照）
$ aws iam create-policy  --policy-name PolicyControllerECRAccess  --policy-document file://ecr-read-policy.json
```

4　IRSA連携の有効化（OIDCプロバイダー）

OIDCプロバイダーを作成し、Kubernetesクラスターに紐づけます。

```
$ eksctl utils associate-iam-oidc-provider  --region=ap-northeast-1 ¥
  --cluster=cs-cluster  --approve
```

演習２　安全なコンテナ構築とランタイム防御　解答編

2-3： ecr-read-policy.json のコード

ecr-read-policy.json のサンプルコードは以下の通りです。
ECRの読み取りに関連するActionを許可するJSONファイルです。

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Action": [   # ECR読み取り関連のACTIONを設定
    "ecr:GetAuthorizationToken",
    "ecr:BatchCheckLayerAvailability",
    "ecr:GetDownloadUrlForLayer",
    "ecr:DescribeRepositories",
    "ecr:ListImages",
    "ecr:DescribeImages",
    "ecr:BatchGetImage"
   ],
   "Resource": "*"   # 付与するリソースへの制限なし
  }
 ]
}
```

ecr-read-policy.json

演習２　安全なコンテナ構築とランタイム防御　解答編

2-3：EKS環境設定手順③

EKSノードおよびKyvernoからECRにアクセスし読み取り可能とするために、AWS側の設定をします。
続いて、KyvernoがECRにアクセスするためのIAMロールを作成します。

Kyverno用 IAMロールの作成

5　ServiceAccountの信頼ポリシー作成

IRSA連携に利用する信頼ポリシーを用意し、ServiceAccountとIAMを紐づける準備をします。

```
$ aws iam list-open-id-connect-providers
   → OIDCプロバイダーARN（赤下線）、OIDCプロバイダー名（緑下線）を確認
     出力例： "OpenIDConnectProviderList": [ {
                 "Arn": "arn:aws:iam::<ACCOUNT_ID>:oidc-provider/oidc.eks.<REGION>.amazonaws.com/id/xxxxxxxxxx"

$ nano trust-policy.json   # ファイル内でOIDCプロバイダー名・ARNを入力します。
```

6　Kyverno用IAMロールの作成・IAMポリシー適用

IAMロールを作成し、IAMポリシーを適用します。

```
$ aws iam create-role  --role-name KyvernoEcrAccessRole ¥
  --assume-role-policy-document file://trust-policy.json

$ aws iam attach-role-policy  --role-name KyvernoEcrAccessRole ¥
  --policy-arn arn:aws:iam::<ACCOUNT_ID>:policy/PolicyControllerECRAccess
```

演習２　安全なコンテナ構築とランタイム防御　解答編

2-3： trust-policy.json のコード

trust-policy.json のサンプルコードは以下の通りです。Kyverno Admission ControllerのServiceAccount
をOIDCプロバイダー経由でIAMにアクセスすることを許可するJSONファイルです。

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
   "Principal": {
    "Federated": "<OIDCプロバイダーARN>"
   },
   "Action": "sts:AssumeRoleWithWebIdentity",
   "Condition": {
    "StringEquals": {
     "<OIDCプロバイダー名>:aud": "sts.amazonaws.com",
     "<OIDCプロバイダー名>:sub": "system:serviceaccount:kyverno:kyverno-admission-controller"
    }
   }
  }
 ]
}
```

trust-policy.json

演習２　安全なコンテナ構築とランタイム防御　解答編

2-3：EKS環境設定手順④

EKSノードおよびKyvernoからECRにアクセスし読み取り可能とするために、AWS側の設定をします。
最後に、Kyvernoを再起動し、IAMロールを利用可能な状態にします。

KyvernoへのIAMロールの適用

7　ServiceAccount 作成

Kyverno用のServiceAccountを作成し、IAMロールを付与します。

```
$ eksctl create iamserviceaccount  --name kyverno-admission-controller  --namespace kyverno ¥
  --cluster cs-cluster  --approve --override-existing-serviceaccounts ¥
 --attach-policy-arn arn:aws:iam::<ACCOUNT_ID>:policy/PolicyControllerECRAccess
```

8　ServiceAccount アノテーション付与

ServiceAccountにアノテーションを付与し、IRSA連携可能にします。

```
$ kubectl annotate sa kyverno-admission-controller  -n kyverno ¥
  eks.amazonaws.com/role-arn=arn:aws:iam::<ACCOUNT_ID>:role/KyvernoEcrAccessRole
```

9　Deployment（Kyverno Admission Controller ）の再起動

Kyverno Admission Controllerのデプロイメントを再起動し、ServiceAccountを反映します。

```
$ kubectl rollout restart deployment kyverno-admission-controller -n kyverno
```

演習２　安全なコンテナ構築とランタイム防御　解答編

2-3：Kyvernoの署名検証ポリシー設定

Kyvernoの署名検証ポリシーを設定し、署名されたイメージのみKubernetesクラスターにデプロイできるように設定しましょう。

署名検証ポリシー設定

1　署名ポリシー設定

Kyvernoの署名ポリシー用マニフェストを用意して、適用してください。

⚠cosignの公開鍵を確認して、マニフェストに貼り付けてください（インデントに注意）。

```
$ cat cosign.pub
$ kubectl apply -f verify-signed-images.yaml
```

cosign.pubの出力例

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQ
EAwK3p1s9E2F6rjN9HZzcv2xg3VjQ8mU7PfRzG1B6K8
-----END PUBLIC KEY-----
```

演習２　安全なコンテナ構築とランタイム防御　解答編

参照）2-3： Kyvernoの署名検証ポリシーのコード

署名検証ポリシーのサンプルコードは以下の通りです。

⚠keyセクションに、cosign.pubの公開鍵の値を貼り付けてください。

⚠署名検証対象以外のリポジトリは検証しない設定のため、必要に応じてブロックする必要があります。

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: verify-signed-images
spec:
  validationFailureAction: enforce # ポリシーの強制
  background: false
  rules:
  - name: verify-signature
    match:
      any:
      - resources:
          kinds:
          - Pod
# 右に続く
```

```
verifyImages:
  - image: "<リポジトリURI>*"
    repository: "<リポジトリURI>"
key: |
      -----BEGIN PUBLIC KEY-----
       （cosign.pubのPUBLIC KEYを貼り付けてください）
      -----END PUBLIC KEY-----
    required: true
    verifyDigest: true
    mutateDigest: true
    useCache: false
```

verify-signed-images.yaml

演習２　安全なコンテナ構築とランタイム防御　解答編

2-3：デプロイ検証

Kyvernoの設定が完了したら、設定された署名されているかを確認しましょう。

⚠署名検証ポリシーで定義したリポジトリが検証対象となります。

デプロイ検証の手順

1　イメージの登録

同一リポジトリに2つのイメージを登録し、
片方だけcosignで署名してください。

⚠ダイジェスト（sha256）の値も異なる必要があります

- 署名済イメージ（tag: latest）
- 署名がないイメージ（tag: unsigned）

ECRリポジトリのイメージ登録例

2　イメージのデプロイ確認

2つのイメージをデプロイできるか検証してください。

署名済イメージ（デプロイ成功）
$ kubectl run test-ok --image=<リポジトリURI>:latest --restart=Never

未署名イメージ（デプロイ失敗）
$ kubectl run test-ng --image=<リポジトリURI>:unsigned --restart=Never

演習２　安全なコンテナ構築とランタイム防御　解答編

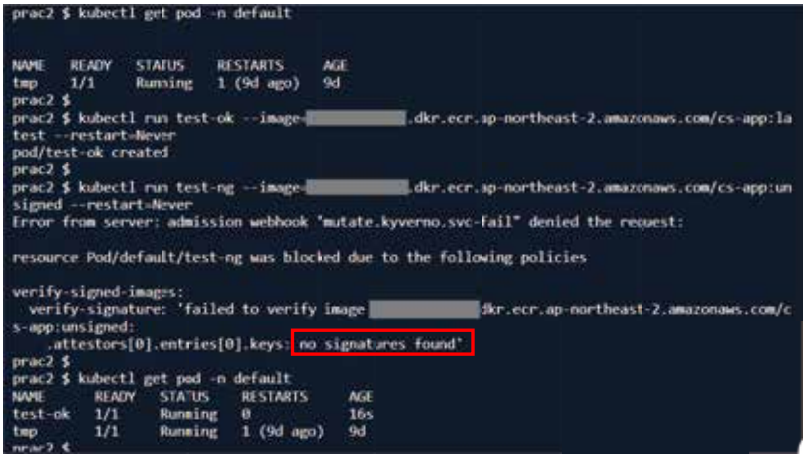
2-2：動作確認（署名と検証）

２　イメージのデプロイ確認

Cosignでの署名済のイメージはデプロイできましたが、未署名のイメージはデプロイに失敗。
Kyvernoの署名検証ポリシーが有効になっていることが分かります。

●署名済イメージ
　→デプロイ成功

●未署名イメージ
　→署名がないため、
　　デプロイ失敗



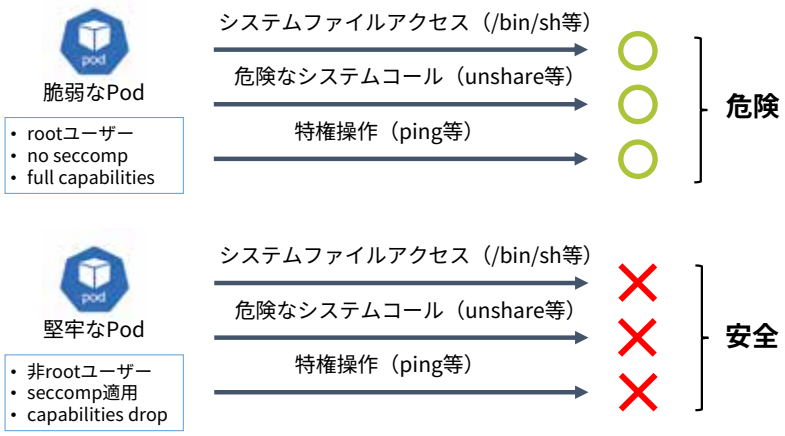
2-4：ランタイム保護による操作制御

解答編

演習２　安全なコンテナ構築とランタイム防御　解答編

2-4：期待される挙動

コンテナランタイムの保護の有無による挙動の違いを理解して、ランタイム保護の必要性を学習します。
本演習では、非root化、seccomp、capabilityを活用してコンテナランタイムを保護します。



ランタイムセキュリティ

seccomp
コンテナがOSに対して呼び出せる「システムコール」を制限します。

capability
Linuxのカーネル権限を分割し、rootでしかできない操作を個別に付与・削除します。

AppArmor
ファイル読み書き・ネットワーク・シグナル送信などをアクセス先を制限します。

演習２　安全なコンテナ構築とランタイム防御　解答編

2-4：ランタイム保護による操作制御

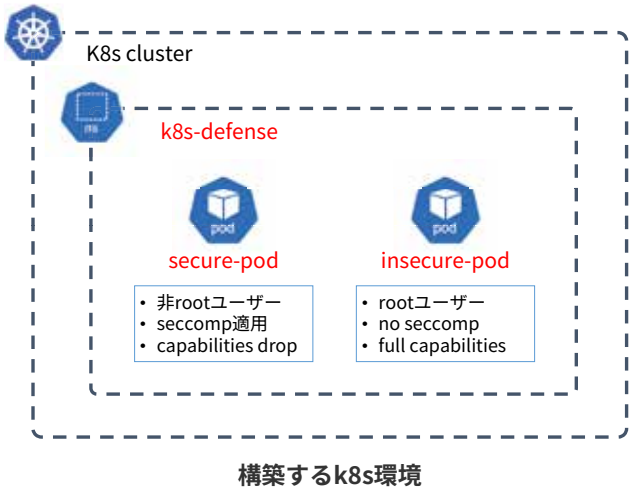
Pod セキュリティ構成（ユーザー権限・seccomp・capabilities）の違いによる挙動の差分を確認し、ランタイム保護の有効性を認識しましょう。

検証手順

1. 新規Namespace作成（未作成の場合）
   ns: k8s-defense

2. podの作成
   pod: secure-pod
   pod: insecure-pod

3. Podの動作確認
   ランタイム保護の影響確認



構築するk8s環境

演習２　安全なコンテナ構築とランタイム防御　解答編

2-4：secure-podとinsecure-podのコード

ランタイムを保護したPod（secure-pod）と脆弱なPod（insecure-pod）のマニフェストは以下の通りです。非rootユーザー起動、seccomp適用、capabilitiesのdropが差分です。

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
  namespace: k8s-defense
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000   # 非rootユーザー起動
    seccompProfile:   # seccompプロファイル適用
      type: RuntimeDefault
  containers:
  - name: demo
    image: alpine
    command: ["sleep", "infinity"]
    securityContext:
      allowPrivilegeEscalation: false # 特権昇格拒否
      capabilities:
        drop: ["ALL"]   # capabilityのドロップ
```

secure-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: insecure-pod
  namespace: k8s-defense
spec:
  containers:
  - name: demo
    image: alpine
    command: ["sleep", "infinity"]
    securityContext:
      allowPrivilegeEscalation: true   # 特権昇格許可
```

insecure-pod.yaml

演習２　安全なコンテナ構築とランタイム防御　解答編

2-4：動作確認

両方のPodに対するコマンドの実行結果の違いから、ランタイム保護の有効性を確認しましょう。

動作確認手順

1　堅牢なPodからのコマンド実行

```
$ kubectl exec -n k8s-defense secure-pod -- id
$ kubectl exec -n k8s-defense secure-pod -- unshare --user --mount
$ kubectl exec -n k8s-defense secure-pod -- ping -c 2 8.8.8.8
```

2　脆弱なPodからのコマンド実行

```
$ kubectl exec -n k8s-defense insecure-pod -- id
$ kubectl exec -n k8s-defense insecure-pod -- unshare --user --mount
$ kubectl exec -n k8s-defense insecure-pod -- ping -c 2 8.8.8.8
```

想定される挙動

Pod	id	unshare	ping
secure-pod	非root（1000）	失敗	失敗
insecure-pod	root（0）	成功	成功
備考	非root化でPSSに対応	危険なシステムコール →seccomp適用で抑止	root権限の操作 →capabilities dropで root操作を抑止

演習２　安全なコンテナ構築とランタイム防御　解答編

2-4：動作確認

1　堅牢なPodからのコマンド実行（secure-pod）

id→1000、　unshare→失敗、　ping→失敗

```
prac2 $ kubectl exec -n k8s-defense secure-pod -- id
uid=1000 gid=0(root) groups=0(root)
prac2 $ kubectl exec -n k8s-defense secure-pod -- unshare --user --mount
unshare: unshare(0x10020000): Operation not permitted
command terminated with exit code 1
prac2 $ kubectl exec -n k8s-defense secure-pod -- ping -c 2 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
ping: permission denied (are you root?)
command terminated with exit code 1
prac2 $
```

2　脆弱なPodからのコマンド実行（insecure-pod）

id→0、　unshare→成功、　ping→成功

```
prac2 $ kubectl exec -n k8s-defense insecure-pod -- id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),26(tape),27(video)
prac2 $ kubectl exec -n k8s-defense insecure-pod -- unshare --user --mount
prac2 $ kubectl exec -n k8s-defense insecure-pod -- ping -c 2 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=104 time=28.236 ms
64 bytes from 8.8.8.8: seq=1 ttl=104 time=28.195 ms

--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 28.195/28.215/28.236 ms
prac2 $
```

演習２　安全なコンテナ構築とランタイム防御　解答編

まとめ：対策効果とベストプラクティス

Trivy：脆弱性診断

対策効果：イメージの脆弱性を早期発見・可視化
- イメージ内の既知脆弱性を事前に検出し、攻撃リスクを削減

ベストプラクティス
- CI/CDパイプラインに組み込み、重大度閾値でビルド判断（例：Critical/High = 0）
- 軽量ベースイメージ選択（alpine/distroless）＋不要パッケージ排除＋非root化
- CVE DBの定期更新とキャッシュ運用、定期的にイメージの脆弱性を確認して随時対処するサイクル

> 脆弱性診断はイメージのセキュリティ対策の基本。デプロイするイメージは脆弱性が少なく軽量なのが理想です

ランタイム保護（seccomp / capabilities）

対策効果：コンテナ実行時の権限を適切に制限
- 権限昇格・危険なシステムコール・横展開攻撃を防止し、セキュリティを強化

ベストプラクティス
- runAsNonRoot + allowPrivilegeEscalation:false + capabilities: drop ALL（必要最小限のみadd）
- seccompProfile: RuntimeDefaultの適用を基本とし、必要に応じてカスタムプロファイルを作成
- PodSecurity(restricted)やRBACと組み合わせることで、より強固なセキュリティ環境を実現

> カーネルベースのシステム制御やKubernetesのセキュリティ技術を組み合わせることが大事です

演習２　安全なコンテナ構築とランタイム防御　解答編

まとめ：対策効果とベストプラクティス

Cosign：イメージ署名

対策効果：イメージの出所証明と改ざん検知
- サプライチェーン攻撃からの保護、信頼できるイメージのみ利用を保証

ベストプラクティス：
- さらにKMSやKeyless署名を活用し、鍵の安全な管理とローテーション計画を策定すると良い
- タグではなくダイジェストで署名・検証、SBOMも署名対象に含めることでセキュリティを強化
- レジストリ権限は最小限に設定、Rekor透明性ログでの監査証跡を強化

> 署名だけでなく、検証と組み合わせてイメージの正常性を確保しましょう

Kyverno（署名検証）

対策効果：クラスタへの未署名イメージデプロイを防止
- 未署名イメージを自動拒否し、クラスタ環境の信頼性を確保

ベストプラクティス
- ClusterPolicyで検証となるkindやnamespaceを正確に設定し、誤検知を防止
- 段階適用（audit→enforce）で影響を最小化、例外はPolicyExceptionで管理
- 公開鍵はConfigMap等で安全配布し、定期的なローテーションを実施するのが望ましい

> 署名検証の仕組みを導入することで、デプロイ前の改ざんを検知するセキュリティゲートの導入に繋がります

演習４　サービス間通信の暗号化とシークレット管理

解答編

演習４　サービス間通信の暗号化とシークレット管理　解答編

演習概要

「すべての通信・アクセスを検証する」というゼロトラストの基本原則に則ったセキュリティ対策を実施し、Kubernetes環境における暗号化・認証・認可を実践的に体得します。

Before：脆弱な通信環境

シークレット情報が平文で保存される
環境変数やConfigMapに直接格納され、取得・漏洩しやすい

サービス間が平文HTTP
通信が暗号化されておらず、盗聴・改ざんリスクがある

サービス識別なし（なりすまし可）
接続元の検証ができず、なりすましやサービス間の横展開攻撃の可能性

After：ゼロトラストな環境

Kubernetes Secret管理
シークレット情報を暗号化して保存し、必要なPodのみアクセス可能に制限

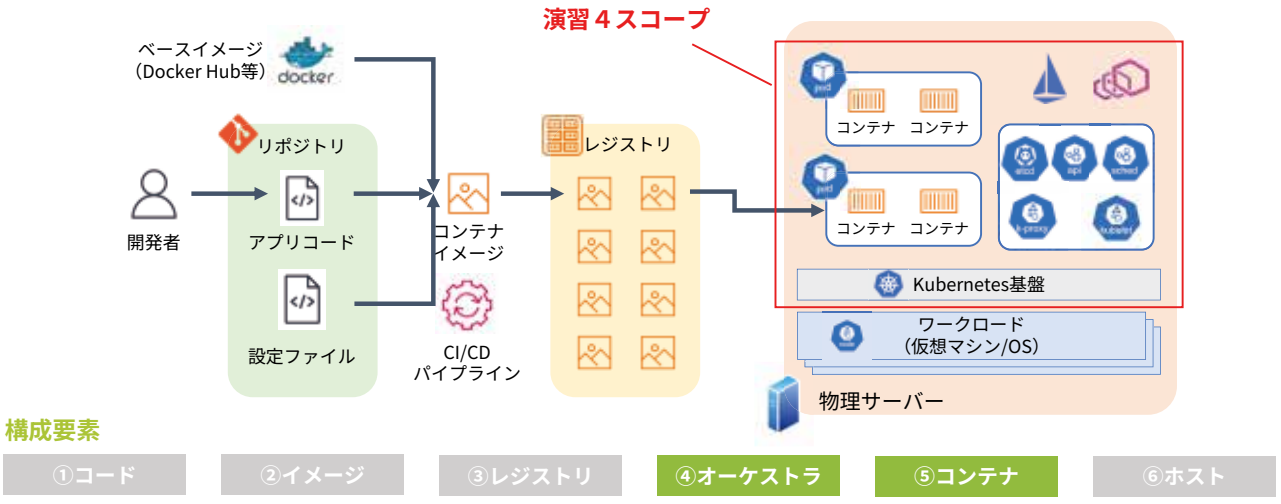
サービス間はIstioによりmTLS化
サイドカー（istio-proxy）が自動的に通信を暗号化し、盗聴・改ざんを防止

AuthorizationPolicyで通信元を厳格化
明示的に許可された通信元からのみ接続を受付

演習４　サービス間通信の暗号化とシークレット管理　解答編

演習４の位置づけ

演習４ではオーケストレーション、コンテナをターゲットにセキュリティ対策を行います。



演習４　サービス間通信の暗号化とシークレット管理　解答編

前提知識：Secret管理

アプリケーションで扱うAPIキー、パスワードなどのシークレット情報を含みます。シークレット情報は分離して管理・制御することで、システム全体のセキュリティを強化することができます。

シークレット情報の課題

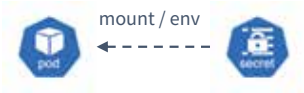
環境変数へのシークレット情報格納
環境変数はプロセス情報から簡単に参照可能なため、
Pod内の全てのコンテナから閲覧できてしまう

ConfigMapへのシークレット保存
ConfigMapは暗号化されずに保存され、
クラスター内で簡単に参照できるため機密情報の保管に不向き

マニフェストへのハードコーディング
Gitリポジトリに直接APIキーなどが記述され、
コード管理ツールで履歴も含めて流出するリスクがあります

基本的なSecretの管理課題
etcdに平文保存される基本的なKubernetes Secret自体も
適切なRBACやSecret暗号化なしでは安全とは言えない

Secretの適切な利用



- **Kubernetes Secretの作成**
  シークレット情報を管理。環境変数やファイルとして
  Podにマウント可能。etcd内では暗号化。
- **RBACによるアクセス制御**
  Secret参照権限を必要なServiceAccountのみに制限
  し、不正アクセスを防止。監査ログも活用

外部Secret管理サービスの活用（推奨）

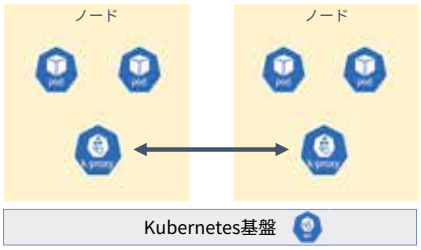
External Secrets Operatorなどを用いて、外部ツール
でSecretを管理することも考えましょう。

Sealed secrets　HashiCorp Vault　AWS Secrets Manager

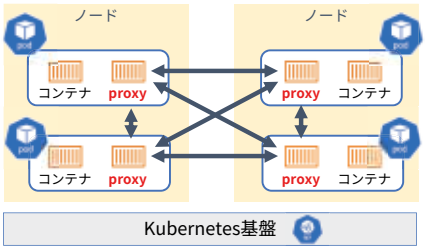
演習４　サービス間通信の暗号化とシークレット管理　解答編

前提知識：サービスメッシュ

マイクロサービス間の通信を制御・可視化します。各サービスに通信を制御するコンテナ（sidecar proxy）を置くことで、アプリケーションを変更せずにトラフィック制御やセキュリティを実現します。



通常のKubernetes



サービスメッシュ

サービスメッシュの背景

- マイクロサービス化によりサービス間通信が複雑化
- 再試行・暗号化・監視などを各サービスで個別に実装する運用が困難
- サービスメッシュの導入により、それらの機能を共通化し、通信を一元的に管理することが可能に

主な特徴

- **可観測性（Observability）**：メトリクス収集、トレース、ログ分析
- **トラフィック管理**：ルーティング制御、リトライ、F/O
- **セキュリティ**：mTLS通信、認証・認可制御
- **ポリシー管理**：通信ポリシーを一元的に設定・適用

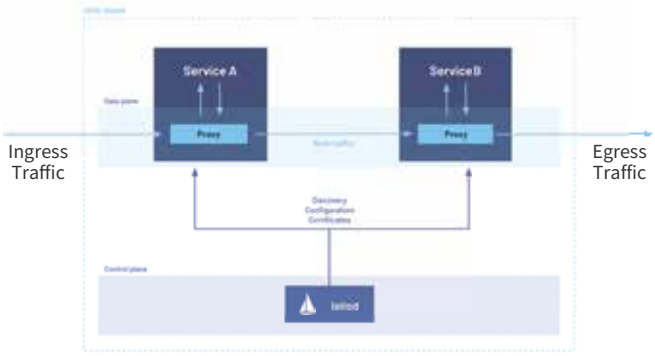
演習４　サービス間通信の暗号化とシークレット管理　解答編

前提知識：Istio

Istioは、制御プレーン（Istiod）とデータプレーン（Envoy）で構成されるサービスメッシュの代表的なツールです。トラフィック制御からセキュリティまで様々な機能を有しています。

主な特徴

- **トラフィック管理**
  （VirtualService / DestinationRule）
  カナリアリリース、Blue/Greenデプロイ、FailOver制御、
  外部通信制御（IngressGW / EgressGW）
- **ポリシー管理**　課題4-2
  （EnvoyFilter / Sidecar）
  通信ルール・外部連携の拡張
  ラベル付与のみでistio-proxy（Envoy）をPodに挿入
- **セキュリティ**　課題4-3、4-4
  （PeerAuthentication / AuthorizationPolicy）
  mTLSによる暗号化通信、認可制御（Zero Trust）
- **可観測性**　課題5
  （Prometheus / Grafana / Kiali）
  メトリクス・トレース・可視化ダッシュボード



Istioのアーキテクチャ
https://istio.io/latest/docs/ops/deployment/architecture/

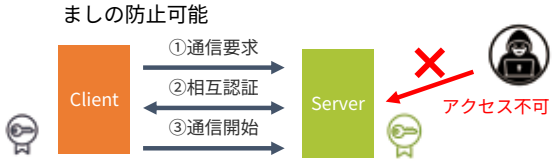
演習４　サービス間通信の暗号化とシークレット管理　解答編

前提知識：mTLSとAuthorizationPolicy（認証と認可）

Istioを利用してサービスメッシュ内の認証・認可を実装することで、ゼロトラストな環境を実現します。
mTLS：通信経路を暗号化し、通信相手を相互認証（AuthN：認証）
AuthorizationPolicy：通信相手・パス・メソッドなどに基づくアクセス制御（AuthO：認可）

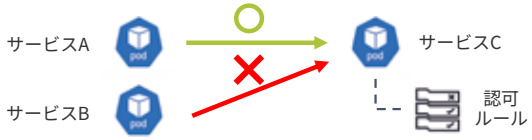
mTLS

- 通信時、クライアントとサーバーの双方が証明書を提示し、相互に認証する仕組み
- 暗号化に加え、サービス間の認証も行うため、なりすましの防止可能



AuthorizationPolicy

- 「どのサービス」が「どのサービス」にアクセス出来るかを制御する仕組み
- Ingress/Egressにも適用でき、サービスメッシュを出入りするトラフィックのアクセス制御も可能



Istioにおける設定方法

- **PeerAuthentication**のCRDで制御可能

  STRICT ：　mTLSのみ許可（推奨）
  PERMISSIVE：　mTLS / 平文も許可
  DISABLE：　mTLS無効

Istioにおける設定方法

- **AuthorizationPolicy**のCRDで制御可能

  Action ：　DENY / ALLOW
  Rules：　from / to

演習４　サービス間通信の暗号化とシークレット管理　解答編

演習４の進め方

「すべての通信・アクセスを検証する」というゼロトラストの基本原則に則ったセキュリティ対策を実施し、Kubernetes環境における暗号化・認証・認可を実践的に体得します。

4-1	対策1 **シークレット情報のSecret管理**	シークレット情報をSecretで管理し、configと分離
4-2	**Istioのインストール**	Istioおよびサンプルアプリケーションをインストールし、サービスメッシュを構築
4-3	対策2 **mTLS通信による認証**	mTLS通信の設定によりPod間通信を暗号化
4-4	対策3 **AuthorizationPolicyによる認可**	AuthorizationPolicyの設定によるアクセス制御

4-1：シークレット情報のSecret管理

解答編

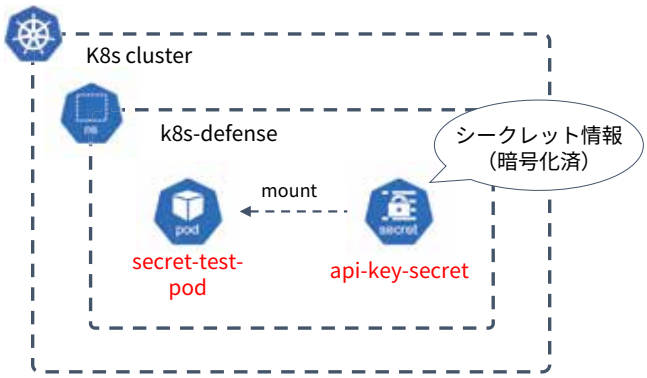
演習４　サービス間通信の暗号化とシークレット管理　解答編

4-1：シークレット情報のSecret管理

Secretリソースを作成し、Podにマウントして注入していきます。
この演習を通じて、基本的なシークレット情報の渡し方や分離管理の重要性を学習しましょう。

検証手順

1. 新規Namespace作成（未作成の場合）
   ns: k8s-defense
2. Secretの作成
   secret: api-key-secret
3. Podの作成
   pod: secure-test
4. Pod / Secretの確認



構築するk8s環境

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-1：Secret / Podのデプロイ

SecretとPodをデプロイしてください。
SecretリソースをPodにマウントできることを確認してください。

作業手順

0　Namespaceの作成

未作成の場合は、作成してください。

```
$ kubectl create secret namespace k8s-defense
```

1　Secretの作成

今回はコマンドラインでSecretを作成してください。

```
$ kubectl create secret generic api-key-secret  -n k8s-defense ¥
  --from-literal=API_KEY=1234-SECRET-5678
```

2　Podのデプロイ

Secretを読み込むPodを作成してください。
Pod内の環境変数およびボリュームに作成したシークレットをマウントしてください。

```
$ kubectl apply -f secret-test-pod.yaml
```

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-1：Podのサンプルマニフェスト

Secretを取り込むPodのサンプルマニフェストを記載します。
環境変数とボリュームマウントの2つの方法で、Secretからシークレット情報をPodに取り込んでいます。

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
  namespace: k8s-defense
spec:
  containers:
  - name: app
    image: busybox
    command: ["sh", "-c", "env | grep API_KEY && cat
/mnt/secrets/API_KEY && sleep 3600"]

# 右に続く
```

```
    env:
    - name: API_KEY
      valueFrom:
        secretKeyRef:
          name: api-key-secret
          key: API_KEY
    volumeMounts:
    - name: secret-volume
      mountPath: "/mnt/secrets" # Secretをvolumeにmount
  volumes:
  - name: secret-volume
    secret:
      secretName: api-key-secret
```

secret-test-pod.yaml

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-1：Pod / Secretの設定確認

SecretとPodのデプロイが完了したら、シークレット情報の見え方を確認してください。
シークレット情報のマウント方法および分離管理について理解を深めましょう。

Pod / Secretの設定確認

1　Podへのマウント確認

Podにシークレット情報を環境変数およびマウントが設定できているか確認します。

```
$ kubectl exec -n k8s-defense secret-test-pod -- env | grep API_KEY
$ kubectl exec -n k8s-defense secret-test-pod -- cat /mnt/secrets/API_KEY
```

2　Secretの確認

Secretのシークレット情報を確認し、暗号化されていることを確認します。
API_KEYは暗号化されているため、base64で復号化して値を確認します。

```
$ kubectl get secret api-key-secret -n k8s-defense -o yaml
$ echo “<暗号化されたキー>" | base64 --decode
```

演習 4　サービス間通信の暗号化とシークレット管理　解答編

4-1：動作確認

1 Podへのマウント確認

secretリソースに保存されたキーが、
Podの環境変数およびVolumeマウントに
反映されていることを確認できます。

```
prac4 $ kubectl get pod -n k8s-defense
NAME              READY   STATUS    RESTARTS      AGE
secret-test-pod   1/1     Running   2 (51m ago)   8h
prac4 $
prac4 $ kubectl get secret -n k8s-defense
NAME             TYPE     DATA   AGE
api-key-secret   Opaque   1      111m
prac4 $
prac4 $ kubectl exec -n k8s-defense secret-test-pod -- env | grep API_KEY
API_KEY=1234-SECRET-5678
prac4 $
prac4 $ kubectl exec -n k8s-defense secret-test-pod -- cat /mnt/secrets/API_KEY
1234-SECRET-5678prac4 $
```

2 Secretの確認

Secretのシークレット情報が暗号化され、
復号した値が一致することを確認できます。

```
prac4 $ kubectl get secret api-key-secret -n k8s-defense -o yaml
apiVersion: v1
data:
  API_KEY: MTIzNC1TRUNSRVQtNTY3OA==
kind: Secret
metadata:
  creationTimestamp: "2025-11-11T04:38:31Z"
  name: api-key-secret
  namespace: k8s-defense
  resourceVersion: "4529513"
  uid: 7ec2d549-c9bc-453c-82dd-2abac6ff21c2
type: Opaque
prac4 $
prac4 $ echo "MTIzNC1TRUNSRVQtNTY3OA==" | base64 --decode
1234-SECRET-5678prac4 $
```

4-2：Istioインストール

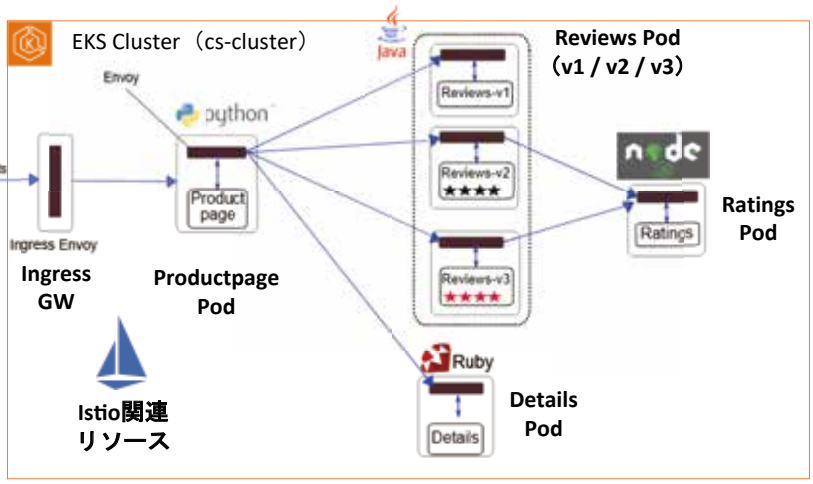
解答編

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-2：期待される挙動

EKSクラスターにIstioをインストールし、Istioのサンプルアプリ（book-info）を立ち上げましょう。
その際、各Podにistio-proxy（Envoy）が挿入された状態にします。

book-infoブラウザ画面



https://istio.io/latest/docs/examples/bookinfo/

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-2：不要リソースのアンインストール

余計なリソースが残っている場合、Istioのインストールに失敗する場合があります。
またリソース不足になる可能性があるため、Kyvernoや他リソースをアンインストールしてください。

1 Kyvernoのアンインストール

Kyvernoをアンインストールしてください。Namespaceも合わせて削除してください。

```
$ helm uninstall kyverno -n kyverno
$ kubectl delete namespace kyverno --ignore-not-found
```

2 Kyvernoの残リソース削除

KyvernoのWebhook設定（2種類）も合わせて削除してください。

```
$ kubectl get mutatingwebhookconfigurations
$ kubectl delete mutatingwebhookconfiguration <リソース名> --ignore-not-found

$ kubectl get validatingwebhookconfigurations
$ kubectl delete validatingwebhookconfiguration <リソース名>  --ignore-not-found
```

3 その他リソース削除

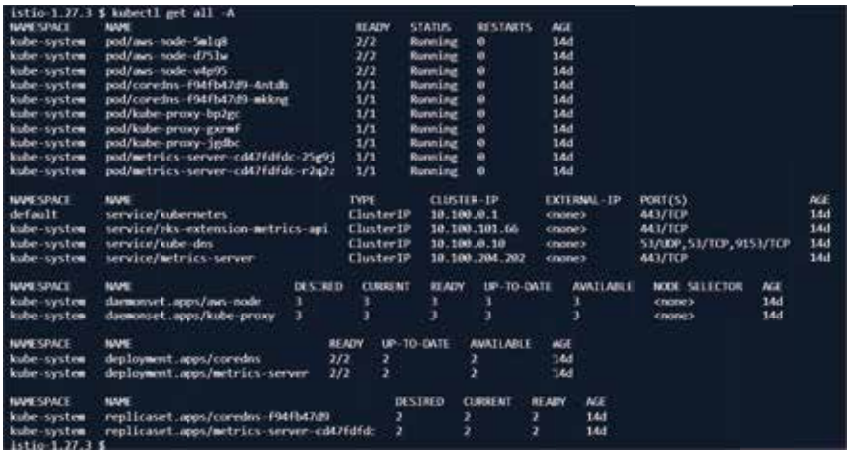
課題1～4-1までのリソースがあれば、kubectl delete等で削除してください。
その他、不要リソースがあれば、合わせて削除してください。

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-2：不要リソースのアンインストール

3 その他リソース削除

$ kubectl get all -A で全Namespaceのリソースを一覧で表示可能です。
これまでの課題で実装したリソースが残っているかを確認して、削除してください。



演習４　サービス間通信の暗号化とシークレット管理　解答編

4-2：Istioのインストール

IstioをKubernetesクラスターにインストールしてください。

インストール手順

1 Istioのインストール

Istioの公式サイトから、Istioをインストールしてください。
⚠本演習ではIstio v1.27.3を利用します（ディレクトリ名もistio-1.27.3で表記します）。

```
$ curl -L https://istio.io/downloadIstio | sh -
$ cd istio-1.27.3

istio-1.27.3 $ export PATH=$PWD/bin:$PATH
istio-1.27.3 $ istioctl install --set profile=demo -y
```

2 Istioの状態確認

Istioのリソースが正常に起動していることを確認してください。

```
istio-1.27.3 $ kubectl get namespace istio-system
istio-1.27.3 $ kubectl get all -n istio-system
```

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-2：book-infoアプリのデプロイ

Istioのサンプルアプリケーション（book-info）をデプロイしてください。
その際、デプロイした各Podにistio-proxyが挿入されていることを確認してください。

インストール手順

1 **Namespaceの設定**

book-infoアプリケーション用のNamespace（book-info）を作成してください。
合わせて、book-info Namespaceにistio-envoy挿入のラベルを付与します。

```
istio-1.27.3 $ kubectl create namespace book-info
istio-1.27.3 $ kubectl label namespace book-info istio-injection=enabled
```

2 **book-infoのアプライ**

ダウンロードしたistioのサンプルアプリケーション（book-info）をインストールしてください。

```
istio-1.27.3 $ kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml -n book-info
istio-1.27.3 $ kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml -n book-info
```

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-2：book-infoの動作確認①

book-infoの動作確認をしてください。
まず各リソースの起動確認とPod間の内部通信を確認します。

動作確認手順

1 **book-info関連リソースの起動確認**

book-info関連のリソースの起動状態を確認してください。
⚠ ローカルPCの場合はローカル、CloudShellの場合はECRに保存されたイメージに署名します。

```
istio-1.27. 3 $ kubectl get pods -n book-info -w   # 6つのPodが2/2で起動すること
istio-1.27. 3 $ kubectl get gateway -n book-info
istio-1.27. 3 $ kubectl get virtualservice -n book-info
```

2 **内部通信確認**

Ratings PodからProductpage Podにアクセスし、Pod間でHTTP通信できることを確認してください。
HTMLのタイトルが表示されればOKです。

```
istio-1.27.3 $ kubectl exec -n book-info "$(kubectl get pod -n book-info ¥
  -l app=ratings -o jsonpath='{.items[0].metadata.name}')" ¥
  -c ratings -- curl -s productpage:9080/productpage | grep -o "<title>.*</title>"
```

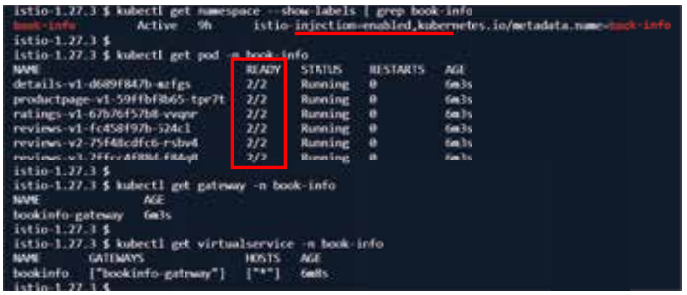
演習４　サービス間通信の暗号化とシークレット管理　解答編

4-2：book-infoの動作確認①

1 book-info関連リソースの起動確認

関連リソースが正常に起動しています。
全Pod、**READY 2/2**となっており、
istio-proxyの挿入を確認できました。

book-info Namespaceには、
istio-injection=enabled のラベルが
付与されているのも確認できます。



2 内部通信確認

Ratings PodからProductpage Podにアクセスし、
HTMLのタイトルを取得できました。

演習４　サービス間通信の暗号化とシークレット管理　解答編

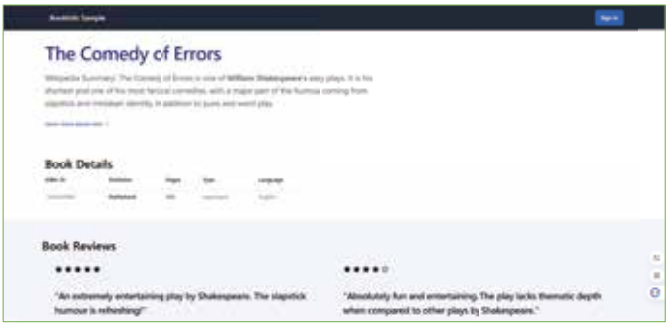
4-2：book-infoの動作確認②

ブラウザからbook-infoにアクセスしてください。Web画面が表示できれば、4-2は完了です。

動作確認手順

3 外部通信確認

ブラウザーからbook-infoを確認できれば、準備完了です。
EKSの場合、自動生成されるロードバランサー、CLBの<DNS名>/productpage、からアクセスできます。



book-infoアプリ（<DNS名>/productpage）

CLB

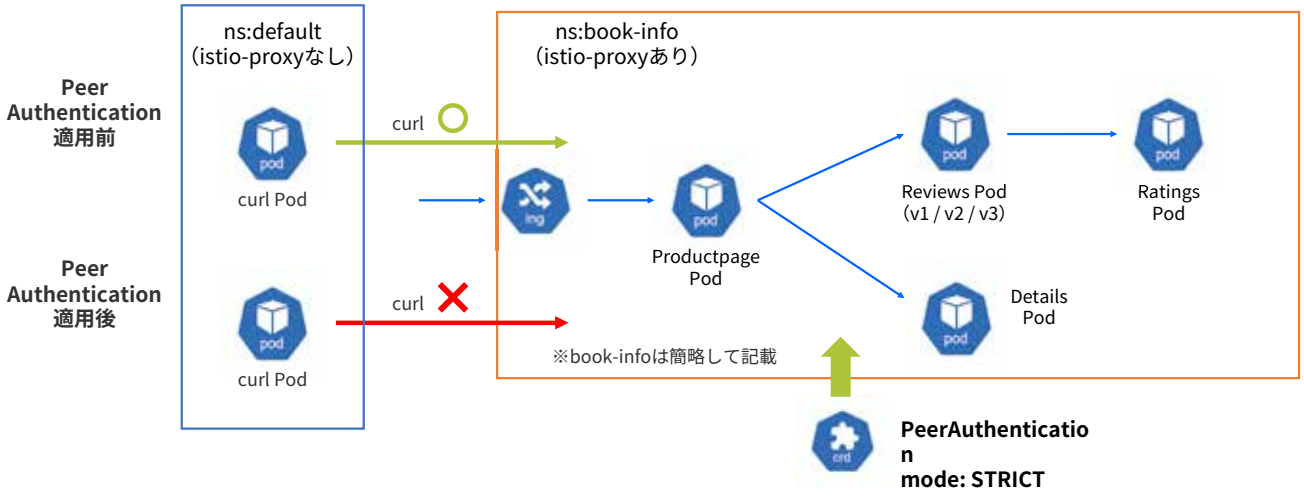
4-3：mTLS通信による認証

解答編

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-3：期待される挙動

PeerAuthenticationでmTLSを有効化します。前後でサービスメッシュ外からの通信が可能か、挙動の変化を確認し、mTLSの有効性を確認しましょう。



演習４　サービス間通信の暗号化とシークレット管理　解答編

4-3：mTLS有効化の適用と動作検証

PeerAuthenticationでmTLSを有効化します。前後でサービスメッシュ外からの通信が可能か、挙動の変化を確認し、mTLSの有効性を確認しましょう。

動作確認手順

1　curlを実行可能なPodの起動・動作検証

book-info以外のNamespaceに新しくcurl Podを起動し、
book-infoのPodにアクセス可能なこと（200OK）を確認してください。

```
$ kubectl run tmp --rm -it --image=curlimages/curl:8.6.0 -- sh
$ curl -v http://details.book-info:9080/details/0
```

2　PeerAuthentication / mTLSの適用

マニフェストを作成し、PeerAuthenticationを設定します。

```
$ nano peer-auth.yaml
$ kubectl apply -f  peer-auth.yaml
```

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: book-info
  namespace: book-info
spec:
  mtls:
    # mTLSを強制（平文通信を拒否）
    mode: STRICT
```

peer-auth.yaml

3　mTLS : STRICT化後　動作検証

curl Podからbook-infoのPodにアクセスできないこと（connection reset by peer）を確認してください。

```
$ kubectl run tmp --rm -it --image=curlimages/curl:8.6.0 -- sh
$ curl -v http://details.book-info:9080/details/0
```

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-3：book-infoの動作確認

1　curlを実行可能なPodの起動・動作検証

外部Namespaceに立てたPodにexecで入ってcurlすると、HTTP 200 OKの返却を確認できます。
kubectl run --rm でPodを一時的に起動しているため、exitしたタイミングでPodが削除されます。

```
istio-1.27.3 $ kubectl run tmp --rm -it --image=curlimages/curl:8.6.0 -- sh
If you don't see a command prompt, try pressing enter.
~ $ curl -v http://details.book-info:9080/details/0
* Host details.book-info:9080 was resolved.
* IPv6: (none)
* IPv4: 10.100.33.231
*   Trying 10.100.33.231:9080...
* Connected to details.book-info (10.100.33.231) port 9080
> GET /details/0 HTTP/1.1
> Host: details.book-info:9080
> User-Agent: curl/8.6.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-type: application/json
< server: istio-envoy
< date: Tue, 11 Nov 2025 07:30:22 GMT
< content-length: 178
< x-envoy-upstream-service-time: 2
< x-envoy-decorator-operation: details.book-info.svc.cluster.local:9080/*
<
```

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-3：book-infoの動作確認

③ mTLS : STRICT化後　動作検証

外部Namespaceに立てたPodにexecで入ってcurlすると、Connection reset by peerが返却されました。
mode: STRICTでmTLSが有効になり、istio-proxyを経由しない通信が拒否されることが分かります。

```
prac4 $ kubectl run tmp --rm -it --image=curlimages/curl:8.6.0 -- sh
If you don't see a command prompt, try pressing enter.
~ $
~ $ curl -v http://details.book-info:9080/details/0
* Host details.book-info:9080 was resolved.
* IPv6: (none)
* IPv4: 10.100.33.231
*   Trying 10.100.33.231:9080...
* Connected to details.book-info (10.100.33.231) port 9080
> GET /details/0 HTTP/1.1
> Host: details.book-info:9080
> User-Agent: curl/8.6.0
> Accept: */*
>
* Recv failure: Connection reset by peer
* Closing connection
curl: (56) Recv failure: Connection reset by peer
~ $
~ $ exit
Session ended, resume using 'kubectl attach tmp -c tmp -i -t' command when the pod is running
pod "tmp" deleted
```

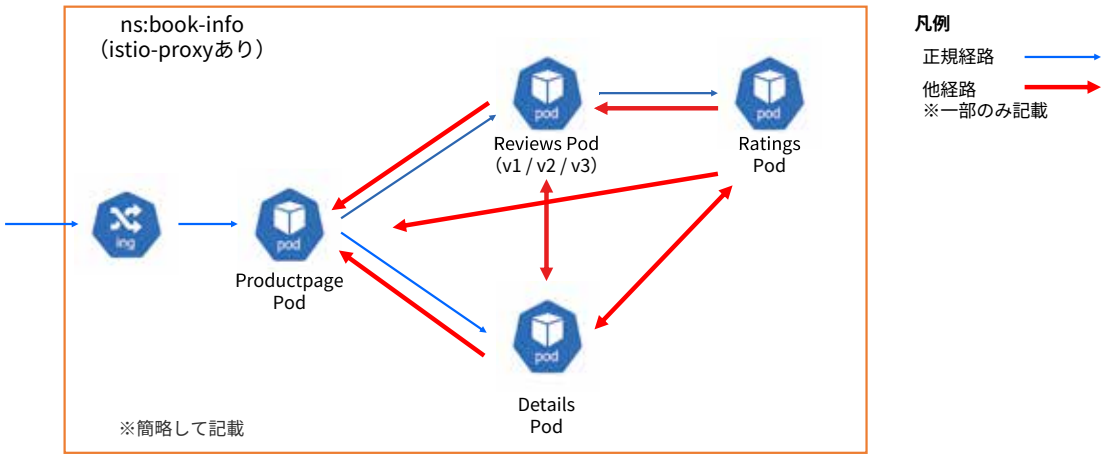
4-4：AuthorizationPolicyによる認可

解答編

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-4：期待される挙動

AuthorizationPolicyに正規の通信経路を設定します。
設定後は、それ以外の経路からの通信が拒否されることを確認します。



演習４　サービス間通信の暗号化とシークレット管理　解答編

4-4：動作検証

AuthorizationPolicy適用前に、正規通路および他経路でのPod間通信を確認しておきましょう。
適用前のため、どのPod間通信も制限されていないはずです。

動作検証手順

1　book-info　Pod間の通信確認

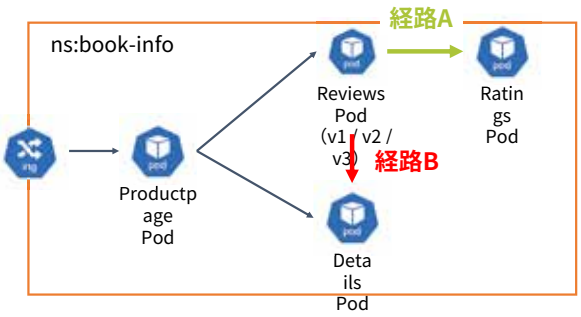
サンプルで、Reviews v2 Pod →Ratings / Details に向けてのPod間通信を確認します。
⚠ これ以外の経路でのPod間通信を確認しても構いません。

経路A（正規）：reviews v2 → ratings 通信（HTTP 200 OK）

```
$ kubectl exec -n book-info $(kubectl get pod -n book-info ¥
 -l app=reviews,version=v2 -o jsonpath='{.items[0].metadata.name}') ¥
 -c reviews -- curl -s -o /dev/null -w ¥
 "HTTP %{http_code}¥n" http://ratings:9080/ratings/0
```

経路B（他）：reviews v2 → details 通信（HTTP 200 OK）

```
$ kubectl exec -n book-info $(kubectl get pod -n book-info ¥
 -l app=reviews,version=v2 -o jsonpath='{.items[0].metadata.name}') ¥
 -c reviews -- curl -s -o /dev/null ¥
 -w "HTTP %{http_code}¥n" http://details:9080/details/0
```

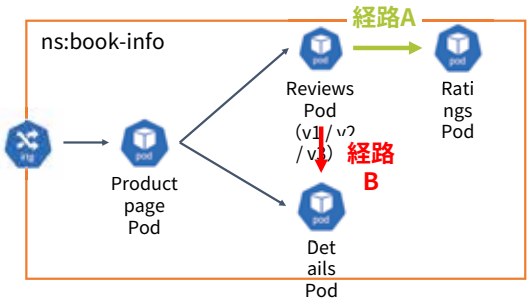


演習４　サービス間通信の暗号化とシークレット管理　解答編

4-4：動作検証

1 **book-info Pod間の通信確認**

Reviews-v2 Pod→Ratings / Details 共に
通信できることを確認しました。
PeerAuthenticationは適用済でPod間のmTLS通信は
できていますが、通信制御はできていません。



経路A（正規）
From: Reviews-v2
To: Ratings
200 OK

経路B（他）
From: Reviews-v2
To: Details
200 OK

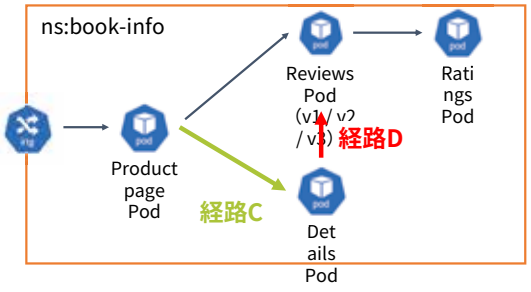
```
prac4 $ kubectl exec -n book-info $(kubectl get pod -n book-info \
>    -l app=reviews,version=v2 -o jsonpath='{.items[0].metadata.name}')  \
>    -c reviews -- curl -s -o /dev/null -w \
>    "HTTP %{http_code}\n" http://ratings:9080/ratings/0
HTTP 200
prac4 $
prac4 $ kubectl exec -n book-info $(kubectl get pod -n book-info \
>    -l app=reviews,version=v2 -o jsonpath='{.items[0].metadata.name}') \
>    -c reviews -- curl -s -o /dev/null \
>    -w "HTTP %{http_code}\n" http://details:9080/details/0
HTTP 200
```

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-4：動作検証

2 **book-info Pod間の通信確認（追加）**

他の正規経路 / 他経路についてもPod間通信が
できることを確認できます。
Podにより入っているパッケージが異なるため、
下記はcurlではなく、PythonやRubyで確認しています。



経路C（正規）
From: Productpage
To: Details
200 OK

経路D（他）
From: Details
To: Reviews
200 OK

```
prac4 $ kubectl exec -n book-info $(kubectl get pod -n book-info -l app=product
page -o jsonpath='{.items[0].metadata.name}') -c productpage -- python -c "impo
rt requests; print(requests.get('http://details:9080/details/0').status_code)"
200
prac4 $
prac4 $ kubectl exec -n book-info $(kubectl get pod -n book-info -l app=details
 -o jsonpath='{.items[0].metadata.name}') \>    -c details -- ruby -rnet/http -e
 "uri=URI('http://reviews:9080/reviews/1'); res=Net::HTTP.get_response(uri); pu
ts res.code"
200
prac4 $
```

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-4：AuthorizationPolicyの設計

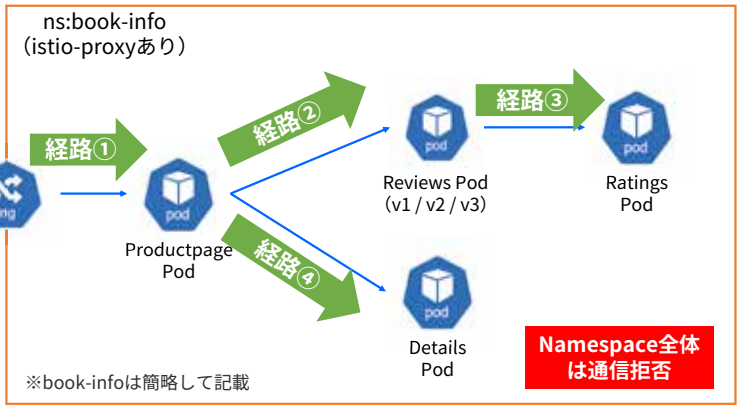
正規の通信経路を整理して、 AuthorizationPolicy（CRD）の設定に落としこみましょう。
Namespace内の通信を全て拒否し、必要な通信経路を許可していきます。

通信経路

以下をAuthorizationPolicyに設定

拒否
- Namespace全体を拒否（Deny All ）

許可
- 経路①：Ingress→Productpage Pod
  ※Ingressからの経路も許可が必要
- 経路②：Productpage Pod → Reviews Pod
- 経路③：Reviews Pod → Ratings Pod
- 経路④：Productpage Pod → Details Pod



演習４　サービス間通信の暗号化とシークレット管理　解答編

4-4：AuthorizationPolicyのサンプルマニフェスト（1/3）

AuthorizationPolicyのサンプルマニフェストを記載します。
今回は、CRDを適用するリソース単位でマニフェストを分割しましたが、まとめても大丈夫です。

```
# Deny all request
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-all
  namespace: book-info    # 対象：book-info全体
spec: {}                  # アクションはすべて拒否
```

Namespace全体の通信を拒否
（autho-deny-all.yaml）

```
# Allow from ingress to productpage
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-ingress-to-productpage
  namespace: book-info
spec:
  selector:
    matchLabels:
      app: productpage    # 対象：productpage
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account"]
```

経路①： Ingress→Productpage Pod
（autho-productpage.yaml）

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-4：AuthorizationPolicyのサンプルマニフェスト（2/3）

AuthorizationPolicyのサンプルマニフェストを記載します。
今回は、CRDを適用するリソース単位でマニフェストを分割しましたが、まとめても大丈夫です。

```
# Allow from prodcutpage to reviews
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-productpage-to-reviews
  namespace: book-info
spec:
  selector:
    matchLabels:
      app: reviews   # 対象：reviews
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/book-info/sa/bookinfo-
productpage"]
```

経路②：Productpage Pod → Reviews Pod
（autho-reviews.yaml）

```
# Allow-reviews-to-ratings
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-reviews-to-ratings
  namespace: book-info
spec:
  selector:
    matchLabels:
      app: ratings   # 適用対象：ratings
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/book-info/sa/bookinfo-
reviews"]
```

経路③：Reviews Pod → Ratings Pod
（autho-ratings.yaml）

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-4：AuthorizationPolicyのサンプルマニフェスト（3/3）

AuthorizationPolicyのサンプルマニフェストを記載します。
今回は、CRDを適用するリソース単位でマニフェストを分割しましたが、まとめても大丈夫です。

```
# Allow from prodcutpage to details
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-productpage-to-details
  namespace: book-info
spec:
  selector:
    matchLabels:
      app: details   # 対象：details
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/book-info/sa/bookinfo-
productpage"]
```

経路④：Productpage Pod → Details Pod
（autho-details.yaml）

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-4：AuthorizationPolicyの設定と動作検証

AuthorizationPolicyのマニフェストを準備して適用してください。
適用後に再度、Pod間通信を確認して、挙動の違いを確認してください。

検証手順

1　AuthorizationPolicyのマニフェスト準備

通信経路をマニフェストに落とし込んで適用してください。
⚠ マニフェストの分割単位は問いません。

2　内部通信確認

Reviews v2 Pod →Ratings / Details に向けてのPod間通信を確認し、挙動の違いを確認してください。

経路A（正規）：reviews v2 → ratings 通信（HTTP 200 OK）

```
$ kubectl exec -n book-info $(kubectl get pod -n book-info ¥
  -l app=reviews,version=v2 -o jsonpath='{.items[0].metadata.name}') ¥
  -c reviews -- curl -s -o /dev/null -w  "HTTP %{http_code}¥n" http://ratings:9080/ratings/0
```

経路B（他）：reviews v2 → details 通信（HTTP 403 NG）

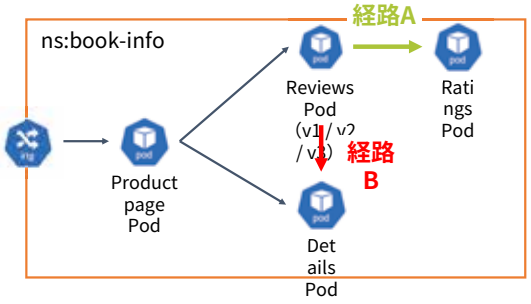
```
$ kubectl exec -n book-info $(kubectl get pod -n book-info ¥
  -l app=reviews,version=v2 -o jsonpath='{.items[0].metadata.name}') ¥
  -c reviews -- curl -s -o /dev/null  -w "HTTP %{http_code}¥n" http://details:9080/details/0
```

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-4：AuthorizationPolicyの設定と動作検証

2　book-info　Pod間の通信確認

Reviews-v2 Pod→Ratings は正規経路のため
通信できましたが、Detailsへの通信は拒否されました。
許可していない経路のため、Namespaceのdeny-allが
適用されることを確認できました。



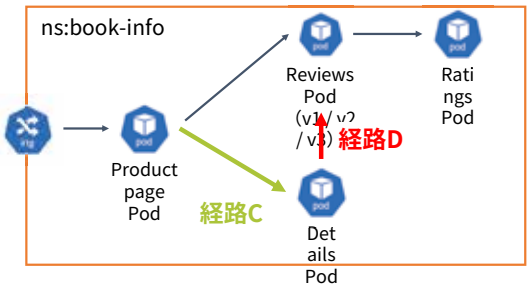
経路A（正規）
From: Reviews-v2
To: Ratings
200 OK

経路B（他）
From: Reviews-v2
To: Details
403 NG

```
prac4 $ kubectl exec -n book-info $(kubectl get pod -n book-info \
>    -l app=reviews,version=v2 -o jsonpath='{.items[0].metadata.name}')  \
>    -c reviews -- curl -s -o /dev/null -w  "HTTP %{http_code}\n" http://rati
ngs:9080/ratings/0
HTTP 200
prac4 $
prac4 $ kubectl exec -n book-info $(kubectl get pod -n book-info \
>    -l app=reviews,version=v2 -o jsonpath='{.items[0].metadata.name}') \
>    -c reviews -- curl -s -o /dev/null  -w "HTTP %{http_code}\n" http://deta
ils:9080/details/0
HTTP 403
```

演習４　サービス間通信の暗号化とシークレット管理　解答編

4-4：AuthorizationPolicyの設定と動作検証



3　book-info Pod間の通信確認（追加）

Productpage Pod→Details は通信できましたが、
Details→Reviewsへの通信は拒否されました。
検証試験の際は、他経路についても網羅的に
通信可能か検証すると良いでしょう。

経路C（正規）
From: Productpage
To: Details
200 OK

経路D（他）
From: Details
To: Reviews
403 NG

```
prac4 $ kubectl exec -n book-info $(kubectl get pod -n book-info -l app=product
page -o jsonpath='{.items[0].metadata.name}') -c productpage -- python -c "impo
rt requests; print(requests.get('http://details:9080/details/0').status_code)"
200
prac4 $
prac4 $
prac4 $ kubectl exec -n book-info $(kubectl get pod -n book-info -l app=details
 -o jsonpath='{.items[0].metadata.name}') \>   -c details -- ruby -rnet/http -e
 "uri=URI('http://reviews:9080/reviews/1'); res=Net::HTTP.get_response(uri); pu
ts res.code"
403
```


演習４　サービス間通信の暗号化とシークレット管理　解答編

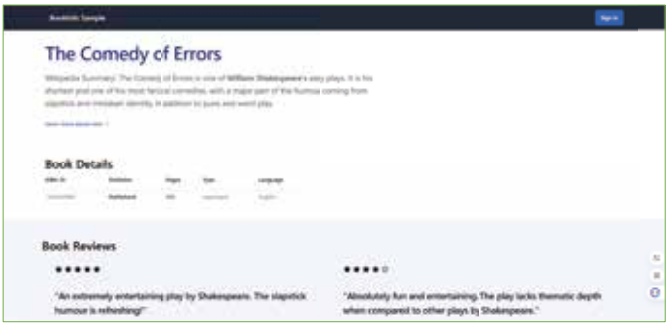
4-4：AuthorizationPolicyの設定と動作検証

ブラウザからbook-infoにアクセスしてください。変わらずWeb画面が表示されれば、4-4は完了です。

検証手順

4　外部通信確認

再度ブラウザからbook-infoを確認してみましょう。
mTLS、Authorizationを適用しても、ブラウザから確認できていればOKです。



book-infoアプリ（<DNS名>/productpage）

CLB

演習４　サービス間通信の暗号化とシークレット管理　解答編

まとめ：対策効果とベストプラクティス

Secret管理

機密情報（APIキー、パスワード等）の安全な管理と提供を実現し、環境変数などへの直接記述による漏洩リスクを軽減

ベストプラクティス

- ボリュームマウントによるファイルとしての提供（env直接注入より安全）
- RBACによる最小権限でのSecret読み取り権限設定
- 定期的なSecret更新とローテーションの自動化
- 外部Secret管理サービスを活用することで、より安全なSecret管理が可能（推奨）

Istio / サービスメッシュ

マイクロサービス間通信の一元管理と制御を実現し、アプリコードを変更せずにセキュリティ機能を導入可能

ベストプラクティス

- リリース戦略やトラフィック制御を活用し、サービスの継続と安定運用
- アプリケーションからネットワーク制御を分割し、サイドカー（istio-proxy）に集約
- 特定のnamespaceから段階的に導入し、影響範囲を制御
- RBACやPSSなどと組み合わせることで、システムをさらに堅牢化

演習４　サービス間通信の暗号化とシークレット管理　解答編

まとめ：対策効果とベストプラクティス

mTLS（相互TLS認証）

クライアント/サーバ双方の証明書検証による相互認証の実現。サービス間通信の暗号化により、盗聴・改ざんを防止

ベストプラクティス

- PERMISSIVE→STRICTへ段階的に適用することで、既存システムへの影響を最小化
- namespace単位でのPeerAuthentication制御で柔軟な運用
- istioctl authn tls-checkで定期的な暗号化状態の確認
- サービス間の通信プロトコルはHTTP/HTTPS/grpcなどが推奨される、必要に応じて改修

AuthorizationPolicy（通信制御）

サービス間の通信経路を明示的に定義。サービス間の不要な通信を遮断し、攻撃対象領域を縮小

ベストプラクティス

- 最小権限の原則に基づく明示的な許可ルールの設定
- Rule（通信元）、Selector（通信先）、Action（操作）を細かく指定
- NetworkPollicyよりも簡易に制御が可能。

演習５　Kialiによる可視化と通信分析

解答編

演習５　Kialiによる可視化と通信分析　解答編

演習概要

サービスメッシュ環境（Istio + book-infoアプリケーション）の可観測性（Observability）を確立し、異常の早期検知と迅速な原因特定により安定稼働を実現するための手法を学習します。

Before：可観測性が不十分な環境

- **マイクロサービス間通信がブラックボックス化**
  サービス間のトポロジーと通信の把握が困難で、依存関係が不透明
- **エラー検知と原因特定の遅延**
  障害発生から対応までの時間が長く、影響範囲の把握が困難
- **セキュリティインシデントの検知困難**
  異常な通信パターンの発見が遅れ、対応が後手に回る

After：可観測性を確立した環境

- **Kialiによるメッシュ可視化**
  サービス間の依存関係と通信状態をリアルタイム表示
- **PrometheusとGrafanaによる監視**
  メトリクス収集とダッシュボード化
- **エラー率と遅延の分析**
  異常の早期検知と根本原因の迅速な特定が可能に

演習５　Kialiによる可視化と通信分析　解答編

演習内容

サービスメッシュ環境（Istio＋book-infoアプリケーション）の可観測性（Observability）を確立し、異常の早期検知と迅速な原因特定により安定稼働を実現するための手法を学習します。

利用するツール

Prometheus
メトリクス収集

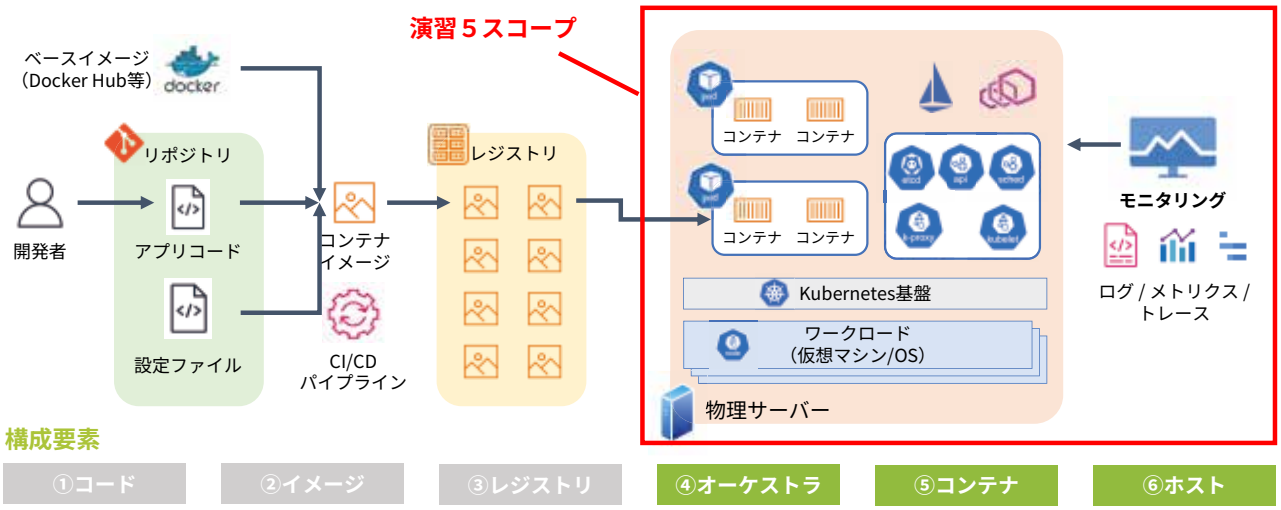
Grafana
メトリクス可視化・分析

Kiali
サービストポロジー可視化

演習５　Kialiによる可視化と通信分析　解答編

演習５の位置づけ

演習５ではオーケストレーション、コンテナ、ホストをターゲットに可観測性を実現します。



構成要素

①コード　②イメージ　③レジストリ　④オーケストラ　⑤コンテナ　⑥ホスト

演習５　Kialiによる可視化と通信分析　解答編

前提知識：可観測性（Observability）

可観測性とはシステムの内部状態を外部から理解できる状態にすることです。従来の閾値やログによる監視から、システム全体を把握して予兆検知、障害対応、性能最適化を可能にします。

Metrics（メトリクス）

- 数値化された計測値（CPU使用率、レイテンシ等）
- 時系列データベースに格納され、集計・可視化

メトリクス種類

インフラ系
CPU、メモリ、ディスクIO
AP系
リクエスト数、エラー率、レイテンシ
Kubernetes/EKS
Pod CPU、メモリ、HPAスケールメトリクス

Logs（ログ）

- イベント発生時の詳細な記録
- エラー内容、リクエスト情報、スタックトレース

ログ種類

- アプリログ
- Kubernetesログ
- セキュリティログ
- インフラログ

Traces（トレース）

- マイクロサービス間の1リクエストの流れを追跡
- 分散システムにおける処理のボトルネック特定

構成

トレース
1つのリクエストの全体像

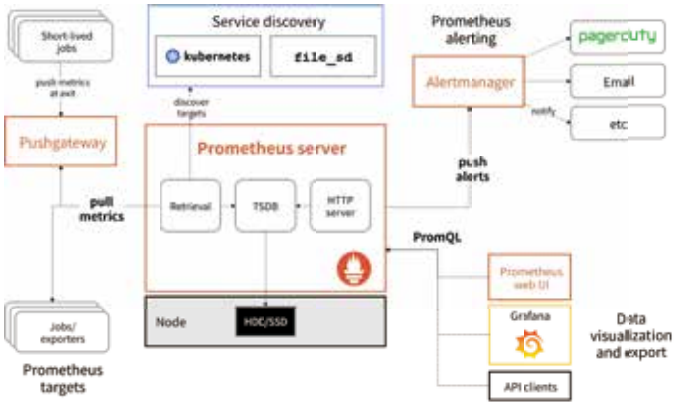
スパン
各サービスでの処理のまとまり
開始〜終了時間、タグ、ログを持つ

「メトリクス」×「ログ」×「トレース」で複合的にシステムを分析して原因を特定します

演習５　Kialiによる可視化と通信分析　解答編

前提知識：Prometheus（メトリクス収集）

Prometheusは、アプリやインフラの状態（CPU使用率・メモリ・リクエスト数など）を自動で収集し、監視や可視化に利用できるオープンソースのモニタリングツールです。



https://prometheus.io/docs/introduction/overview/

Prometheusの特徴

オープンソースの監視ツール
コンテナ環境の監視に最適なOSS

時系列データベース
メトリクスの収集・保存・クエリに特化した高性能データベースを内蔵

Pullモデル
ターゲットからメトリクスを収集
エクスポーターの種類が豊富

サービスディスカバリ
動的環境でも自動的にモニタリングターゲットを検出

強力なクエリ言語
PromQLによる柔軟なメトリクスの検索・集計・分析

演習５　Kialiによる可視化と通信分析　解答編

前提知識：Grafana（メトリクス可視化）

Grafanaは可視化と監視のためのプラットフォームです。サーバーやコンテナ、アプリケーションなどから収集したメトリクスを、グラフや表などで表現し、アラート通知します。Prometheusと連動して動かすことが多いです。

Grafana - Node Exporterダッシュボード
https://grafana.com/grafana/dashboards/

Grafanaの特徴

リアルタイム可視化
CPUやメモリなどのリソース使用状況をリアルタイムで表示

カスタムダッシュボード
必要なメトリクスのみをグループ化して一覧表示可能。正常/警告/危険状態を視覚的に色分けすることも可能。

インタラクティブなグラフ
ズームイン・アウトや時間範囲変更が可能

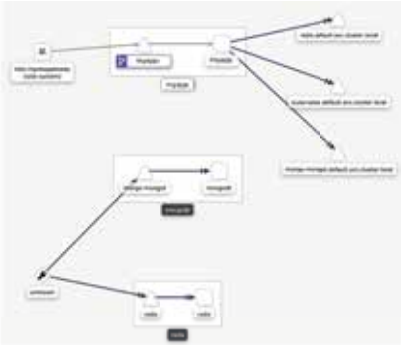
多様なグラフタイプ
折れ線グラフ、ゲージ、ヒートマップ、テーブルなど

演習５　Kialiによる可視化と通信分析　解答編

前提知識：Kiali（サービストポロジー可視化）

Kialiはサービスメッシュ可視化ツールです。サービスメッシュ内の通信、遅延、エラー率をリアルタイムにグラフ表示します。mTLS状態・トレース・サービス依存関係を自動マップ化することも可能です。

Kiali アーキテクチャ
https://kiali.io/docs/architecture/architecture/



Kiali トラフィックグラフ
https://kiali.io/docs/faq/graph/

Kialiの特徴

サービス間通信のリアルタイム可視化
マイクロサービス同士の通信経路やリクエスト数、エラー率、レイテンシをリアルタイムで表示

サービスマップ（依存関係の自動生成）
サービス間のつながりをトラフィックグラフとして表示。サービス間の通信方向を可視化

mTLS状態の可視化
サービス間の通信が mTLS で保護されているかをアイコンで表示

トラフィック分析（詳細レベル）
成功/失敗リクエストの割合、p90/p99レイテンシ、RPSなどを細かく表示。ボトルネック分析や性能チューニングに役立つ

演習５　Kialiによる可視化と通信分析　解答編

演習５の進め方

マイクロサービス環境（Istio + book-infoアプリケーション）の可観測性（Observability）を確立し、異常の早期検知と迅速な原因特定により安定稼働を実現するための手法を学習します。

5-1	**Prometheus / Grafana / Kialiのインストール**	Helm経由でPrometheus / Grafana / Kialiをインストール
5-2	**Kialiトラフィックグラフによる可視化**	Kialiのトラフィックグラフで、book-infoアプリケーションのトポロジーを可視化
5-3	**Kialiを用いた通信分析**	複数パターンでPod間通信を発生させ、Kialiによる通信状況やエラーを分析
5-4	**Grafanaダッシュボード作成**	Grafanaのダッシュボードを作成し、Kubernetes環境の状況をメトリクスで可視化

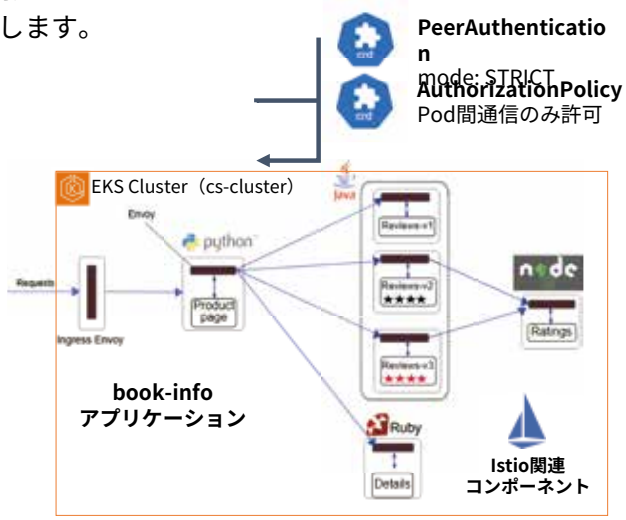
演習５　Kialiによる可視化と通信分析　解答編

演習前提

演習5は、EKS、Istio、book-infoアプリケーションを利用して進めます。
EKSクラスターを作成の上、演習4実施後の環境を用意します。

EKSクラスター（演習4-4実施後）

- **Istio**
  istio-system Namespaceにインストール済
- **book-infoアプリケーション**
  book-info Namespaceに、
  Istioのサンプルアプリ「book-info」をデプロイ済
- **PeerAuthentication**
  mode: STRICTを適用済
- **AuthorizationPolicy**
  以下ルールを適用済
  - Namespace全体の通信は拒否
  - book-infoのPod間通信のみ許可



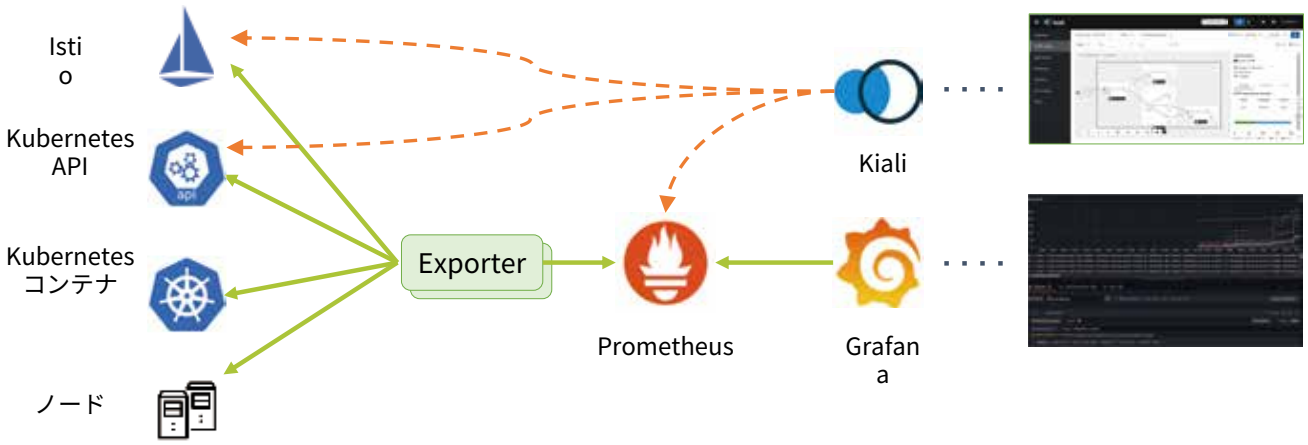
5-1：Prometheus / Grafana / Kialiのインストール

解答編

演習５　Kialiによる可視化と通信分析　解答編

5-1：期待される挙動

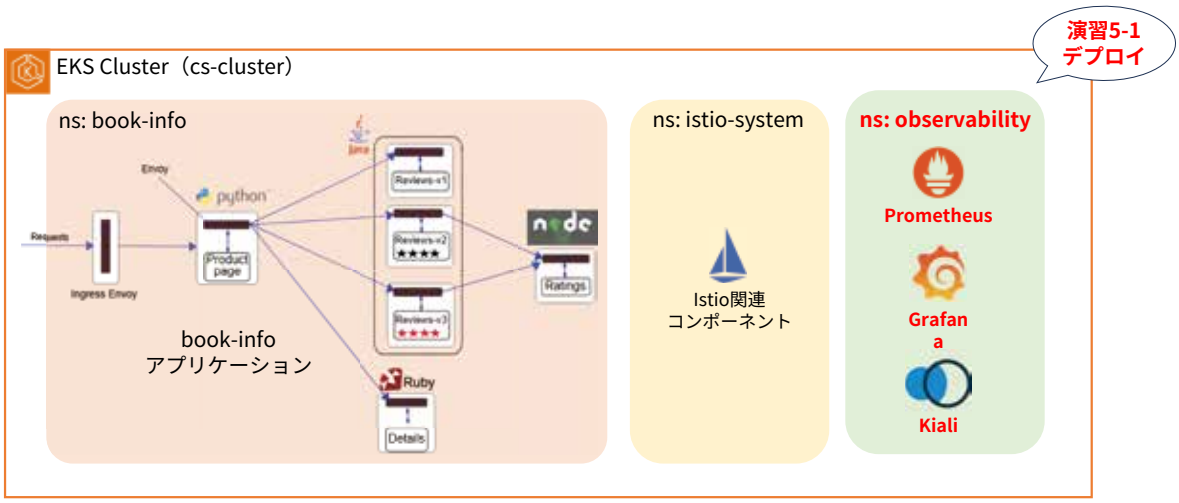
EKSクラスターにPrometheus / Grafana / Kialiをインストールし、可観測性の準備をします。



演習５　Kialiによる可視化と通信分析　解答編

5-1： Prometheus / Grafana / Kialiのインストール

EKSクラスターにPrometheus / Grafana / Kialiをインストールします。
これらのツールは新規Namespaceにインストールします。



演習５　Kialiによる可視化と通信分析　解答編

5-1： Prometheus / Grafana / Kialiのインストール

EKSクラスターにPrometheus / Grafana / Kialiをインストールします。
これらのツールは新規Namespaceにインストールします。

演習5-1手順

1. **Namespaceの作成**
   新規Namespace「observability」を作成します。
   ```
   $ kubectl create namespace observability
   ```

2. **Helmインストール**
   Helmがない場合は、先にHelmをインストールします。
   ```
   $ curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
   $ helm version
   ```

3. **Helmリポジトリの追加**
   Prometheus / Grafana / KialiのHelmリポジトリを追加します。
   ```
   $ helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
   $ helm repo add grafana https://grafana.github.io/helm-charts
   $ helm repo add kiali https://kiali.org/helm-charts
   $ helm repo update
   ```

演習５　Kialiによる可視化と通信分析　解答編

5-1： Prometheus / Grafana / Kialiのインストール

EKSクラスターにPrometheus / Grafana / Kialiをインストールします。
これらのツールは新規Namespaceにインストールします。

演習5-1手順

4　Prometheusのインストール

⚠ 動作軽量化のため、今回はPrometheus ServerはPVCを利用せず、Alertmanagerは無効化します。

```
$ helm install prometheus prometheus-community/prometheus -n observability ¥
  --set alertmanager.enabled=false  --set server.persistentVolume.enabled=false ¥
  --set server.statefulSet.enabled=false
```

5　Grafanaのインストール

```
$ helm install grafana grafana/grafana -n observability ¥
    --set adminPassword='admin'  --set service.type=LoadBalancer
```

6　Kialiのインストール

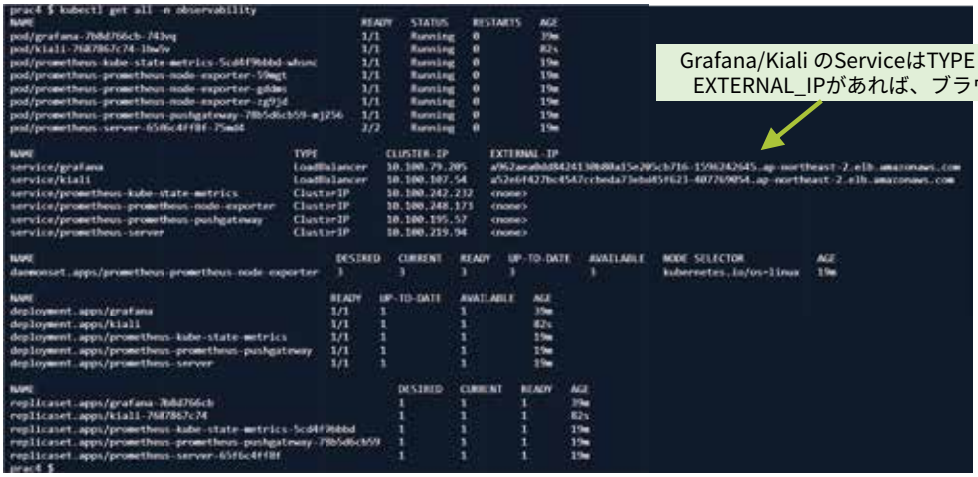
```
$ helm install kiali kiali/kiali-server -n observability  --set auth.strategy=anonymous
  --set external_services.prometheus.url=http://prometheus-server.observability.svc.cluster.local ¥
  --set deployment.service_type=LoadBalancer
```

⚠ Grafana / Kialiは外部ブラウザからアクセスできるよう、Serviceのタイプを「LoadBalancer」にしています

演習５　Kialiによる可視化と通信分析　解答編

5-1：動作確認

インストールが完了したら、各ツールの起動状態を確認します。
observability Namespaceにリソースが正常起動していれば、演習5-1は完了です。



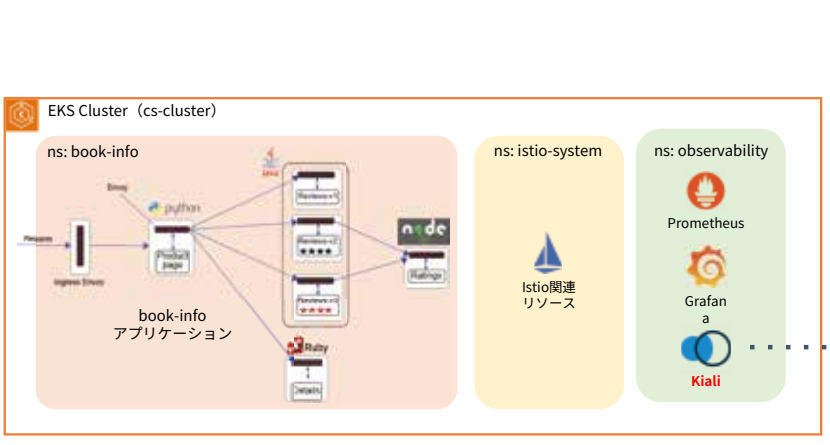
5-2：Kialiトラフィックグラフによる可視化

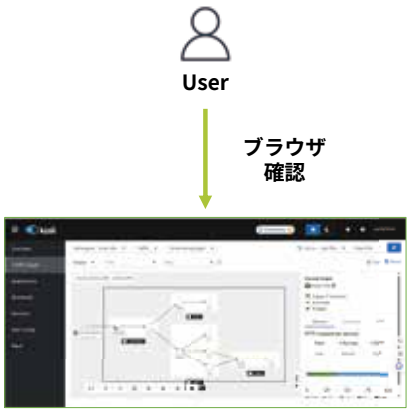
解答編

演習５　Kialiによる可視化と通信分析　解答編

5-2：期待される挙動

Kialiのトラフィックグラフにアクセスし、book-infoアプリケーションのトポロジーが表示されることを確認します。サービスメッシュの可視化を実現します。





Kiali
トラフィックグラフ

演習５　Kialiによる可視化と通信分析　解答編

5-2： Kialiトラフィックグラフによる可視化

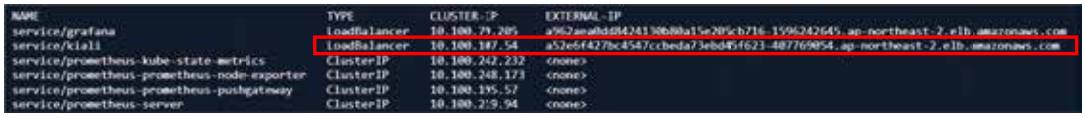
Kialiにアクセスして、トラフィックグラフを表示し、book-infoアプリケーションのトポロジーが可視化されることを確認します。サービスのトポロジーを確認できたら、演習5-2は完了です。

演習5-2 手順

1　Kialiの接続先を確認

KialiのService（Type：LoadBalancer）を確認し、EXTERNAL-IPを確認します。

$ kubectl get service -n observability



⚠ ServiceのTypeがClusterIPの場合は、kubectl patchでLoadBalancerに変更することができます。

$ kubectl patch svc kiali -n observability -p '{"spec":{"type":"LoadBalancer"}}'

2　ブラウザからKialiにアクセス

ブラウザから以下にアクセスし、Kialiが表示されることを確認します。

http://<EXTERNAL-IP>:20001

演習５　Kialiによる可視化と通信分析　解答編

5-2： Kialiトラフィックグラフによる可視化

Kialiにアクセスして、トラフィックグラフを表示し、book-infoアプリケーションのトポロジーが可視化されることを確認します。サービスのトポロジーを確認できたら、演習5-2は完了です。

3　トラフィックグラフの確認

Kialiのトラフィックグラフ（Traffic Graph）を操作して、book-infoアプリケーションのトポロジーが表示されることを確認します。またその通信が暗号化されていることを確認します。

⚠ 事前にブラウザからbook-infoに何度かアクセスし、トラフィックを発生させます。

トラフィックグラフ設定

トラフィックグラフの設定を変更することで、可視化する項目や時間帯を変更できます。

設定例

- Namespace：book-info
- Traffic：Httpに✅
- App graph / Versioned app graph
- Display：以下項目に✅
  Response Time / Traffic Rate / Security
- Last 10m / Every 1m

演習５　Kialiによる可視化と通信分析　解答編

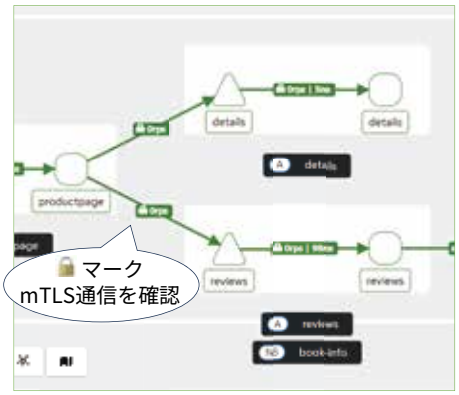
5-2： 動作確認

3 **トラフィックグラフの確認**

Versioned app graphを選択することで、
Pod単位の通信を表示可能になります。

Versioned app graph

鍵マークから Pod間通信が暗号化されて
いることが分かります。（mTLS通信）



Display：Securityに✅

5-3：Kialiを用いた通信分析

解答編

演習５　Kialiによる可視化と通信分析　解答編

5-3：期待される挙動

book-infoアプリケーションに対して、様々なトラフィックを発生させ、Kialiのトラフィックグラフで検知できることを確認します。今回は5つのパターンで通信状態を確認します。

パターン１　定常負荷
正常時のベースライン取得
通常運用時のリクエスト数、レイテンシ、エラー率の基準値を取得します。

パターン２　高負荷スパイク
急激なトラフィック増加の観測
高負荷ツールを使用して、急激なリクエスト増加時のシステム挙動を観測します。

パターン３　非認証Pod 通信
istio-proxyなしPodからのアクセス
サイドカーを持たないPodからサービスメッシュ内サービスへのアクセスに失敗することを確認します。

パターン４　非認可経路の通信
PeerAuthenticationで許可されていない経路
許可されていない経路で通信し、アクセスに失敗することを確認します。

パターン５　Fault Injection（遅延）
VirtualServiceによる遅延注入
特定のサービスに意図的な遅延を注入し、システムの挙動を観測します。

演習５　Kialiによる可視化と通信分析　解答編

5-3： Kialiを用いた通信分析

book-infoアプリケーションに対して、幾つかのパターンでトラフィックを発生させます。
各通信パターンにおけるトポロジー、レスポンス、リクエスト量などを確認します。

作業内容

1. **パターン１　定常負荷**
   book-infoアプリケーションのproductpageに定常負荷を掛け、Kialiで通信状態を確認します。
   サンプルコマンド　$ hey -z 60s -q 5 -c 10 http://<EXTERNAL-IP>/productpage

2. **パターン２　高負荷**
   book-infoアプリケーションのproductpageに高負荷を掛け、Kialiで通信状態を確認します。
   サンプルコマンド　$ hey -z 60s -q 40 -c 40 http://<EXTERNAL-IP>/productpage

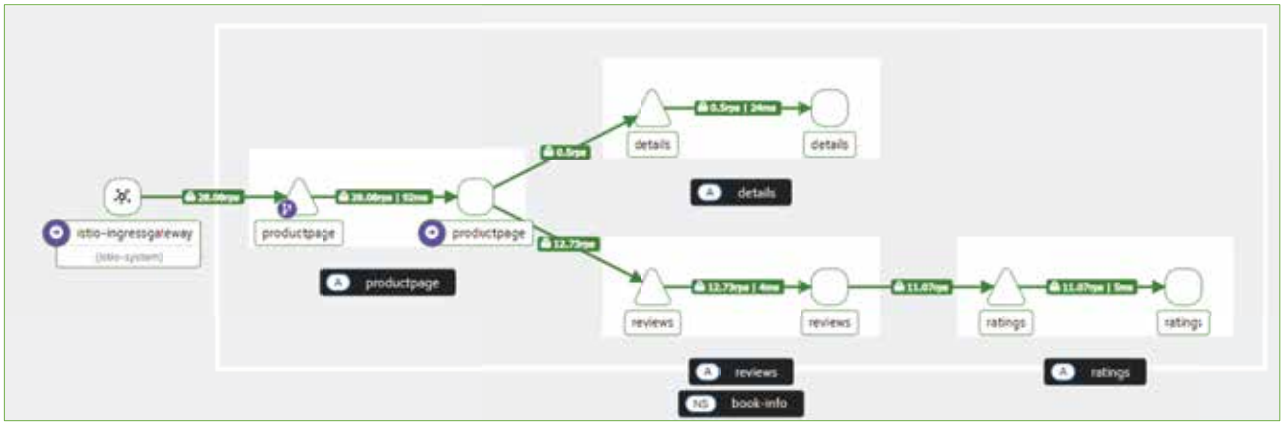
3. **動作確認**
   定常負荷時、高負荷時を掛けた際の変化を確認します。
   - サービストポロジー
   - レスポンスタイム（p95 Percentile）
   - リクエスト量（Traffic Rate）

演習５　Kialiによる可視化と通信分析　解答編

5-3：動作確認（パターン１：通常負荷）

1

パターン１　定常負荷
サービストポロジーがbook-infoアプリケーションの構成通りに表示されました。
定常的なレスポンス、リクエストの値を確認できました。エラーも発生していません。
　IngressGW→productpage Pod（レスポンス：92ms、リクエスト量：28.07rps）



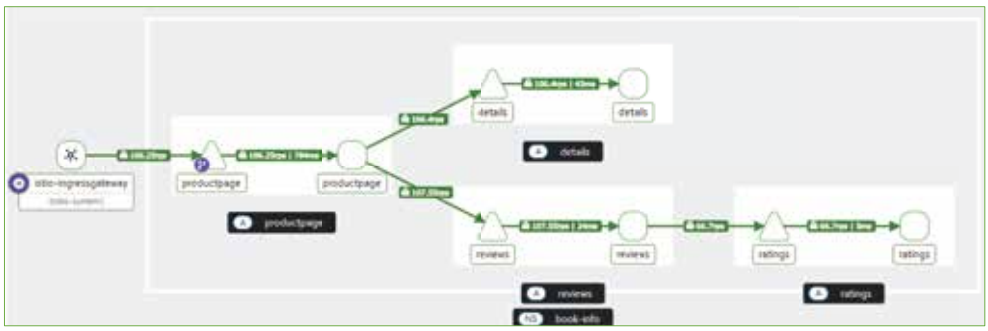
サービストポロジー（通常負荷）

演習５　Kialiによる可視化と通信分析　解答編

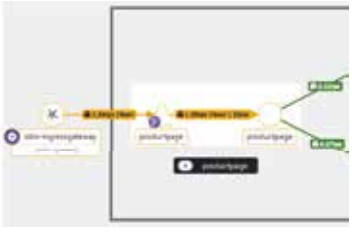
5-3：動作確認（パターン２：高負荷）

2

パターン２　高負荷
サービストポロジーがbook-infoアプリケーションの構成通りに表示されました。高負荷が掛かったことにより、
レスポンス、リクエストの急増を確認できました。場合によってはエラーが発生している可能性があります。
　IngressGW→productpage Pod（レスポンス：782ms、リクエスト量：106.25rps）



サービストポロジー（高負荷）



エラー発生時のトポロジー
（エラー率：1%）

演習５　Kialiによる可視化と通信分析　解答編

5-3：動作確認（パターン１、２）

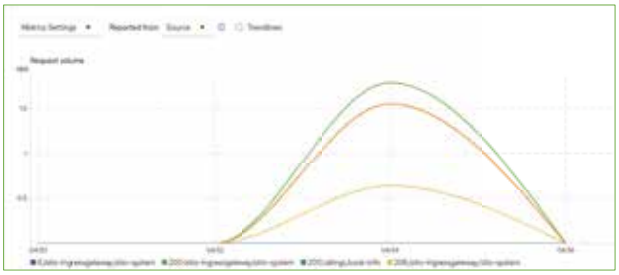
3 **動作確認**
パターン１、２における、
IngressGWからproductpageに対する
パフォーマンスを比較した結果です。
高負荷になると性能劣化することを確認できます。

項目	パターン１	パターン２
レスポンス	92ms	782ms
リクエスト	28.07rps	106.25rps

サービスに関連するメトリクスをグラフ表示することも可能です

インバウンドメトリクス

例えば、レスポンスコード別のリクエスト分析可能です



リクエスト - レスポンスコード

演習５　Kialiによる可視化と通信分析　解答編

5-3： Kialiを用いた通信分析

book-infoアプリケーションに対して、幾つかのパターンでトラフィックを発生させます。
各通信パターンにおけるトポロジー、レスポンス、リクエストなどを確認します。

作業内容

4 **パターン３　非認証Pod通信**
一時的にout-mesh Pod（Istio-proxyなし）を立て、book-infoアプリ（ratings）にアクセスします。
Kialiで一時的なPodからratingsへのトラフィックが100%エラーになっていることを確認します。

サンプルコマンド
$ kubectl run out-mesh --rm -it -n book-info --image=curlimages/curl -- sh
~ $ curl -v http://ratings.book-info.svc.cluster.local:9080/ratings/1

5 **パターン４　非認可経路の通信**
AuthorizationPolicyで許可されていないPod間通信を発生させます。
Kialiでreivews v2からratingsへのトラフィックが100%エラーになっていることを確認します。

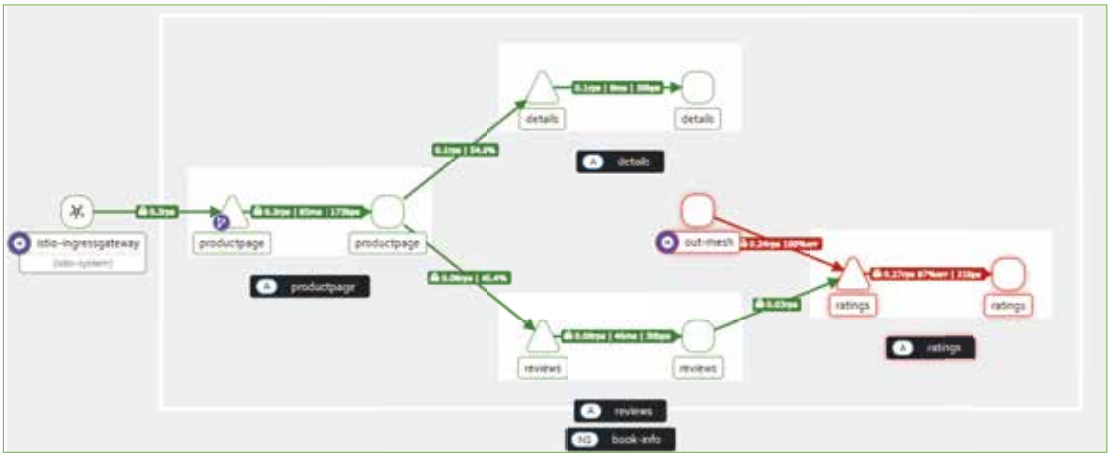
サンプルコマンド
$ kubectl exec deploy/reviews-v2 -n book-info -it -- curl -v http://details:9080/details/1

演習５　Kialiによる可視化と通信分析　解答編

5-3：動作確認　パターン３　非認証Pod通信

4　パターン３　非認証Pod通信

一時的なPod（out-mesh）からratings Podへの通信がエラーになったことを確認できました。
Istio-proxyを経由したアクセスは継続して正常に通信できています。

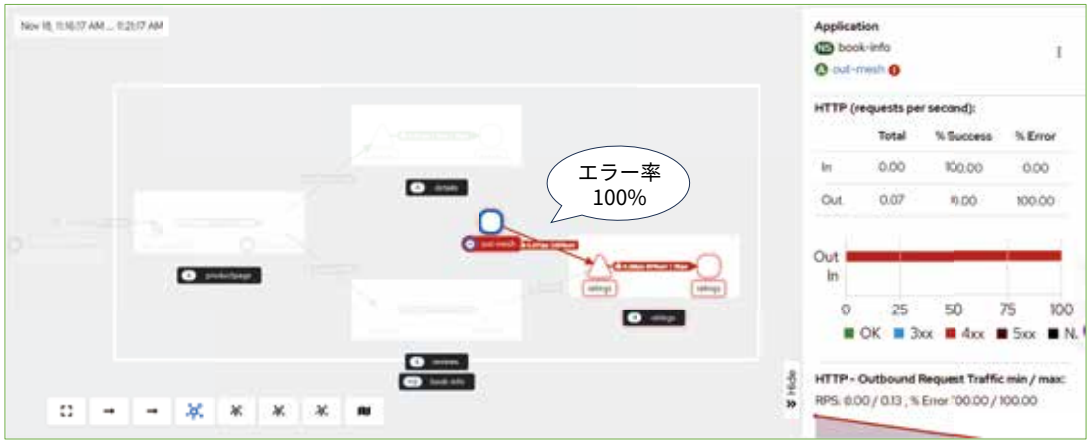


演習５　Kialiによる可視化と通信分析　解答編

5-3：動作確認　パターン３　非認証Pod通信

4　パターン３　非認証Pod通信

経路を選択することで、絞って状況を可視化することができます。一時的なPod（out-mesh）から
ratings Podへの通信経路を選択し、エラー率100%であることを確認できました。

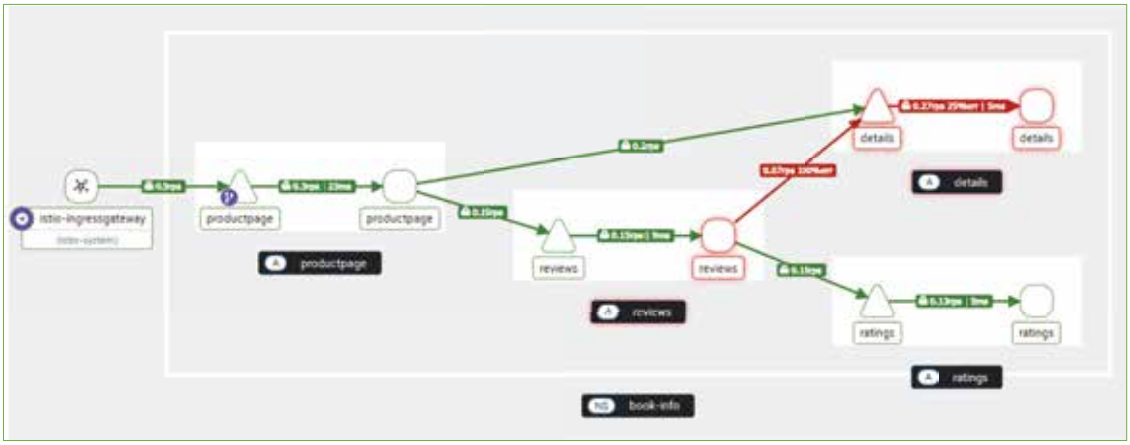


演習５　Kialiによる可視化と通信分析　解答編

5-3：動作確認　パターン４　非認可経路の通信

5　パターン４　非認可経路の通信

AuthorizationPolicyで許可されていないreviews Podからratings Podへの通信がエラーになったことを確認できました。許可された経路は正常に通信できています。

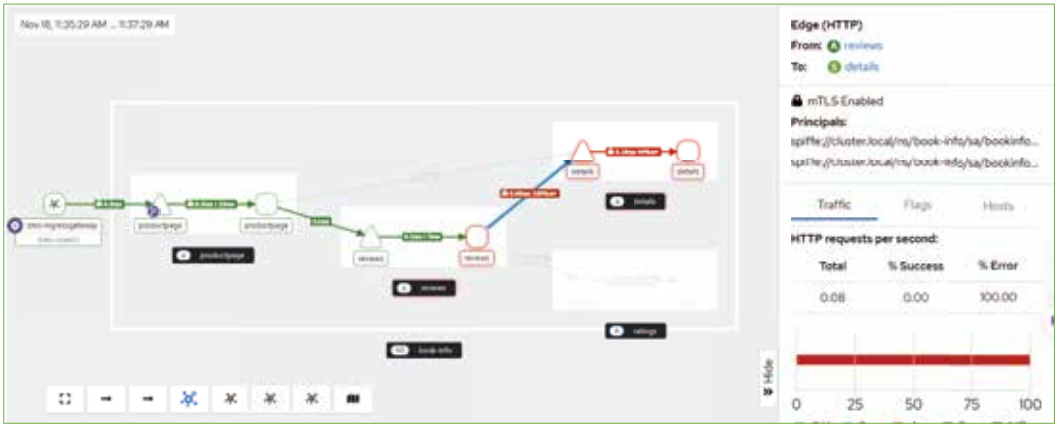


演習５　Kialiによる可視化と通信分析　解答編

5-3：動作確認　パターン４　非認可経路の通信

5　パターン４　非認可経路の通信

reviews Podからratings Podへの通信がエラー率100%であることを確認できました。



演習５　Kialiによる可視化と通信分析　解答編

5-3： Kialiを用いた通信分析

book-infoアプリケーションに対して、幾つかのパターンでトラフィックを発生させます。
各通信パターンにおけるトポロジー、レスポンス、リクエストなどを確認します。

作業内容

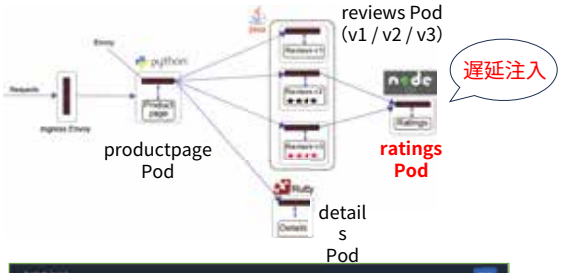
6　パターン５　Fault Injection

Fault injection（VirtualService）で意図的に5秒の処理遅延を注入し、reviews v2/v3からratingsへの通信をタイムアウトさせてください。
またKialiで該当トラフィックがエラーになっていることを確認してください。

⚠ ratingsへの通信はproductpage側の設定により、3秒でタイムアウトします。

⚠ VirtualServiceのサンプルマニフェストは次ページ。

```
$ nano fault-injection.yaml
$ kubectl apply -f fault-injection.yaml
```



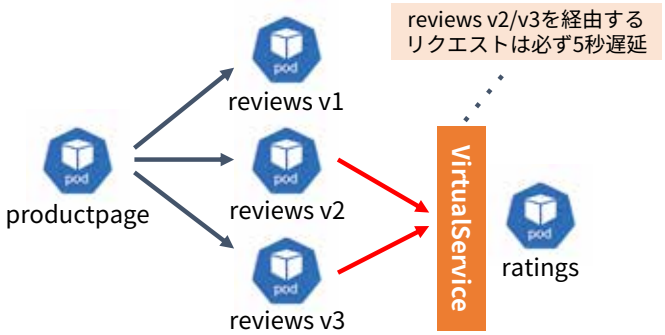
演習５　Kialiによる可視化と通信分析　解答編

5-3：Virtual Serviceのサンプルマニフェスト

Fault Injection（遅延注入）を実現するVirtualServiceのサンプルマニフェストです。
このマニフェストを適用することで、reviews v2/v3 からratingsへの通信が一律5秒遅延します。

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: ratings
  namespace: book-info
spec:
  hosts:
  - ratings
  http:
  - fault:
      delay:
        fixedDelay: 5s
        percentage:
          value: 100
    route:
    - destination:
        host: ratings
```

fault-injection-delay.yaml

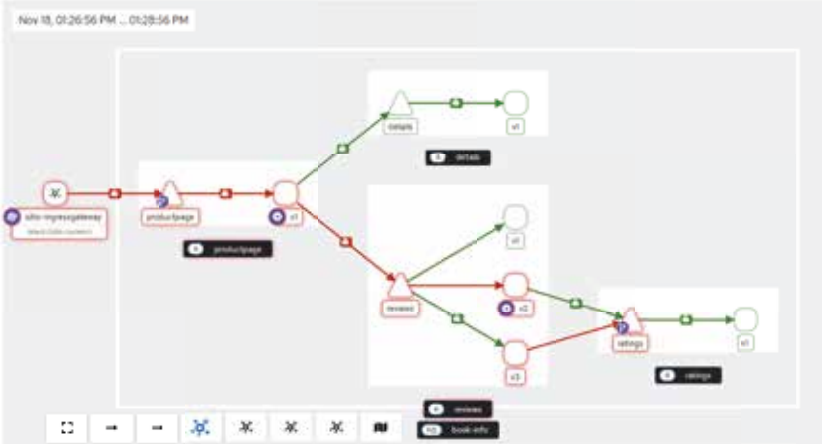


演習５　Kialiによる可視化と通信分析　解答編

5-3：動作確認　パターン５　Fault Injection

5　**パターン５　Fault Injection**

Fault Injectionによるratingsへの5秒遅延の注入により、reviews v2/v3からratingsへの通信が100%エラーになっていることを確認できました。reviews v1経路の通信は正常です。



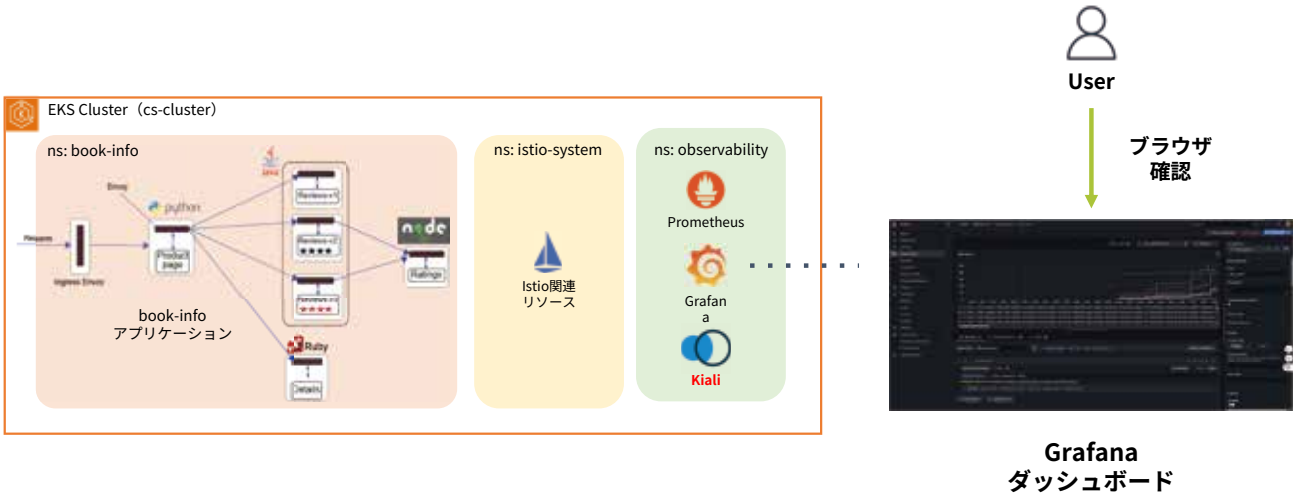
5-4：Grafanaダッシュボード作成

解答編

演習５　Kialiによる可視化と通信分析　解答編

5-4：期待される挙動

Grafanaにアクセスし、EKSクラスターのメトリクスを使ったダッシュボードを作成します。
Prometheusでメトリクスを収集し、Grafanaで参照することで実現します。



演習５　Kialiによる可視化と通信分析　解答編

5-4：Grafanaダッシュボード作成

Grafanaにアクセスし、EKSクラスターのメトリクスを使ったダッシュボードを作成します。
Prometheusでメトリクスを収集し、Grafanaで参照することで実現します。

演習5-4 手順

1. **Grafanaの接続先を確認**

   GrafanaのService（Type：LoadBalancer）を確認し、EXTERNAL-IPを確認します。

   ```
   $ kubectl get service -n observability
   ```

2. **ブラウザからGrafanaにアクセス**

   ブラウザから以下にアクセスし、
   Grafanaが表示されることを確認します。
   http://<EXTERNAL-IP>

3. **Grafanaにログイン**

   ID/PW：admin/adminでログインします。

Grafana ログイン後トップページ

演習５　Kialiによる可視化と通信分析　解答編

5-4：Grafanaダッシュボード作成

GrafanaにアクセスしEKSクラスターのメトリクスを使ったダッシュボードを作成します。
Prometheusでメトリクスを収集し、Grafanaで参照することで実現します。

演習5-4 手順

4 **PrometheusとGrafanaの統合**

左ペイン > Connections > Data Source > Add data source > Prometheusを選択
Connectionに、プロメテウスサーバーのURLを入力して「Save & Test」を押下
　URL：http://prometheus-server.observability.svc.cluster.local

5 **Grafanaダッシュボード作成**

左ペイン > Dashboards > New Dashboardを選択
K8sクラスター、Istio、アプリケーション関連のメトリクスを使って複数パネルを作成します。

メトリクス例
- node_cpu_seconds_total
- istio_request_duration_milliseconds_bucket
- istio_requests_total

演習５　Kialiによる可視化と通信分析　解答編

5-4：Grafanaダッシュボード作成

5 **Grafanaダッシュボード作成**

パネル１：ノード単位のCPU使用率（%）
Data source：　Prometheus
Metrics：　node_cpu_seconds_total（ノード単位の各コア、各モードのCPU使用時間の累計値 - 秒）
PromQL：　100 - (avg(irate(node_cpu_seconds_total{mode="idle"}[5m])) by (instance) * 100)
Visualization：　Gauge

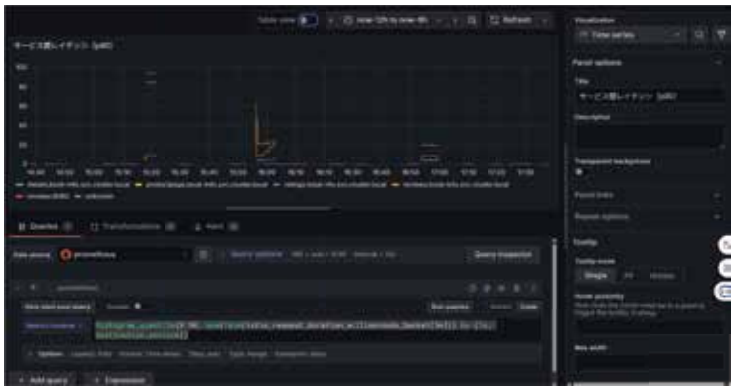
演習５　Kialiによる可視化と通信分析　解答編

5-4：Grafanaダッシュボード作成

5 Grafanaダッシュボード作成

パネル２：istio の各サービスに対するリクエスト遅延のp90（ms）

Data source：　Prometheus
Metrics：　istio_request_duration_milliseconds_bucket （istio-proxyが計測したリクエスト遅延のヒストグラム）
PromQL：　histogram_quantile(0.90, sum(rate(istio_request_duration_milliseconds_bucket[5m])) by (le, destination_service))
Visualization：　Time series



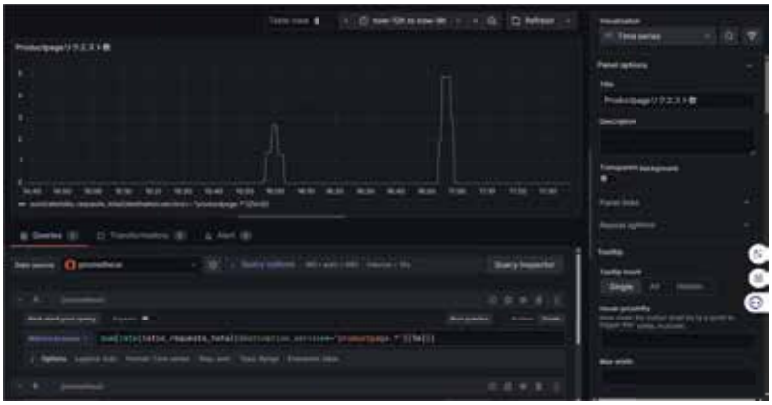
演習５　Kialiによる可視化と通信分析　解答編

5-4：Grafanaダッシュボード作成

5 Grafanaダッシュボード作成

パネル３：productpageサービスへのリクエスト数/秒（rps）

Data source：　Prometheus
Metrics：　istio_requests_total（Istio-proxyが処理したリクエスト数－個）
PromQL：　sum(rate(istio_requests_total{destination_service="productpage.book-info.svc.cluster.local"}[5m]))
Visualization：　Time series



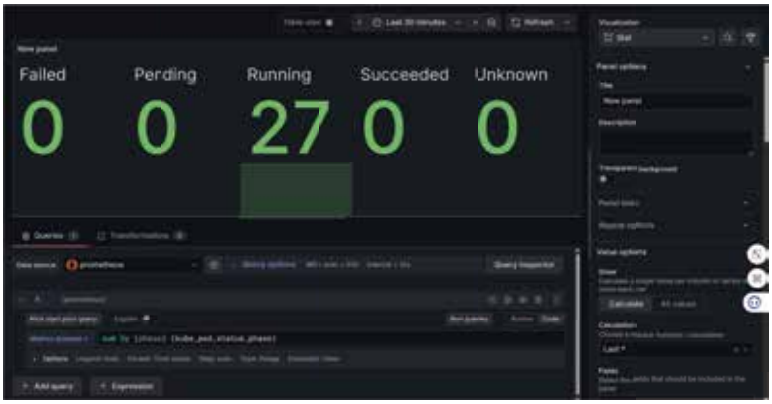
演習５　Kialiによる可視化と通信分析　解答編

5-4：Grafanaダッシュボード作成

5 Grafanaダッシュボード作成

パネル４：クラスター全体のフェーズ別Pod起動状態（個）

Data source：　Prometheus
Metrics：　kube_pod_status_phase（Podの起動状態）
PromQL：　sum by (phase) (kube_pod_status_phase)
Visualization：　Stat



演習５　Kialiによる可視化と通信分析　解答編

5-4：Grafanaダッシュボード作成

5 Grafanaダッシュボード作成

最終的にはパネルを組み合わせてダッシュボードを作成します。
用途別に複数のダッシュボードを用意したり、グラフの表示形式やサイズなど調整することができます。

演習５　Kialiによる可視化と通信分析　解答編

5-4：（参考）Grafanaダッシュボードのテンプレート

Grafanaにはダッシュボードやパネルのテンプレートも用意されています。
Dashboards > New > New Dashboard > 「Import dashboard」「Add library panel」から利用可能です。

Node Exporter Full
（ID：1860）

Kubernetes cluster monitoring (via Prometheus)
（ID：315）

演習５　Kialiによる可視化と通信分析

まとめ

演習５　Kialiによる可視化と通信分析　解答編

まとめ：対策効果とベストプラクティス

可観測性（Prometheus/Grafana/Kiali）

サービスメッシュを可視化。メトリクス・ログ・トレースの統合的な監視により異常の早期検知と迅速な原因特定が可能

ベストプラクティス

- Prometheus標準メトリクスの活用とカスタムメトリクスの追加
- GrafanaダッシュボードでSLI/SLOを可視化し、チーム内で共有
- Kialiによるリアルタイムトポロジ監視とトラフィック分析
- アラートルールの設定と運用プロセスへの統合

運用とセキュリティ

可観測性をセキュリティ運用に活用し、異常なトラフィックやエラースパイクを検知。継続的な改善サイクルを確立

ベストプラクティス

- 定常負荷時のベースラインを定期的に取得・更新
- エラー率・遅延の閾値を設定し、自動アラート化
- Fault Injectionによる障害訓練とレジリエンス向上
- mTLS状態とAuthorizationPolicyの適用状況を定期監査

令和７年度「専門職業人材の最新技能アップデートのための専修学校リカレント教育（リ・スキリング）推進事業」
情報技術者の技能アップデートのためのリカレント教育推進事業

コンテナサーバーセキュリティ教材資料
解答編

令和８年２月

一般社団法人全国専門学校情報教育協会

〒164-0003 東京都中野区東中野 1-57-8 辻沢ビル３F

電話：03-5332-5081 FAX.03-5332-5083