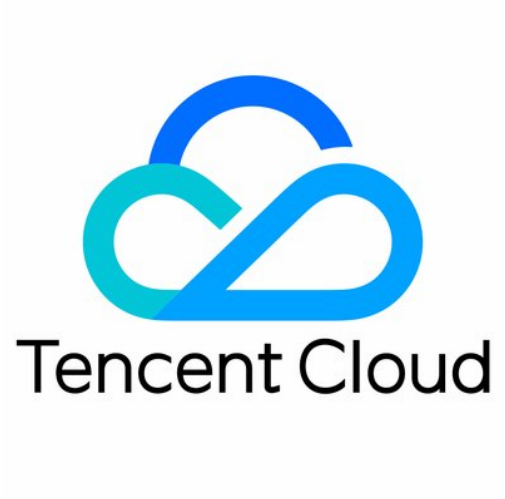
Tencent Kubernetes Engine

ベストプラクティス

製品ドキュメント



カタログ：

ベストプラクティス
Serverlessクラスター
Elastic IPを介してパブリックネットワークにアクセスします
Serverlessクラスターでディープラーニングを堪能する
ディープラーニングコンテナイメージをビルドする
よくあるご質問
パブリックネットワークアクセス関連
ネットワーク
DNS関連
TKE DNSベストプラクティス
TKEでのカスタムドメイン名解決の実現
Nginx Ingressベストプラクティス
ログ
NginxIngressカスタムログ
監視
Prometheusを使用してMySQLとMariaDBをモニタリングします
運用・保守
クラスター審査を使用したトラブルシューティング
DevOps
TKEベースのJenkins外部ネットワークアーキテクチャアプリケーションのビルドとデプロイ
ステップ1：TKEクラスター側およびJenkins側の設定
ステップ2：Slave podのビルド設定
TKEでJenkinsをデプロイする
オートスケーリング
クラスターオートスケーリング実践
TKE上でカスタム指標を使用して自動スケーリングします
TKEでHPAを使用してサービスのAuto Scalingを実装します
ストレージ
CFS-Turboクラスのファイルシステムを静的にマウントします
TKE ServerlessでCFS-Turboを静的にマウントします

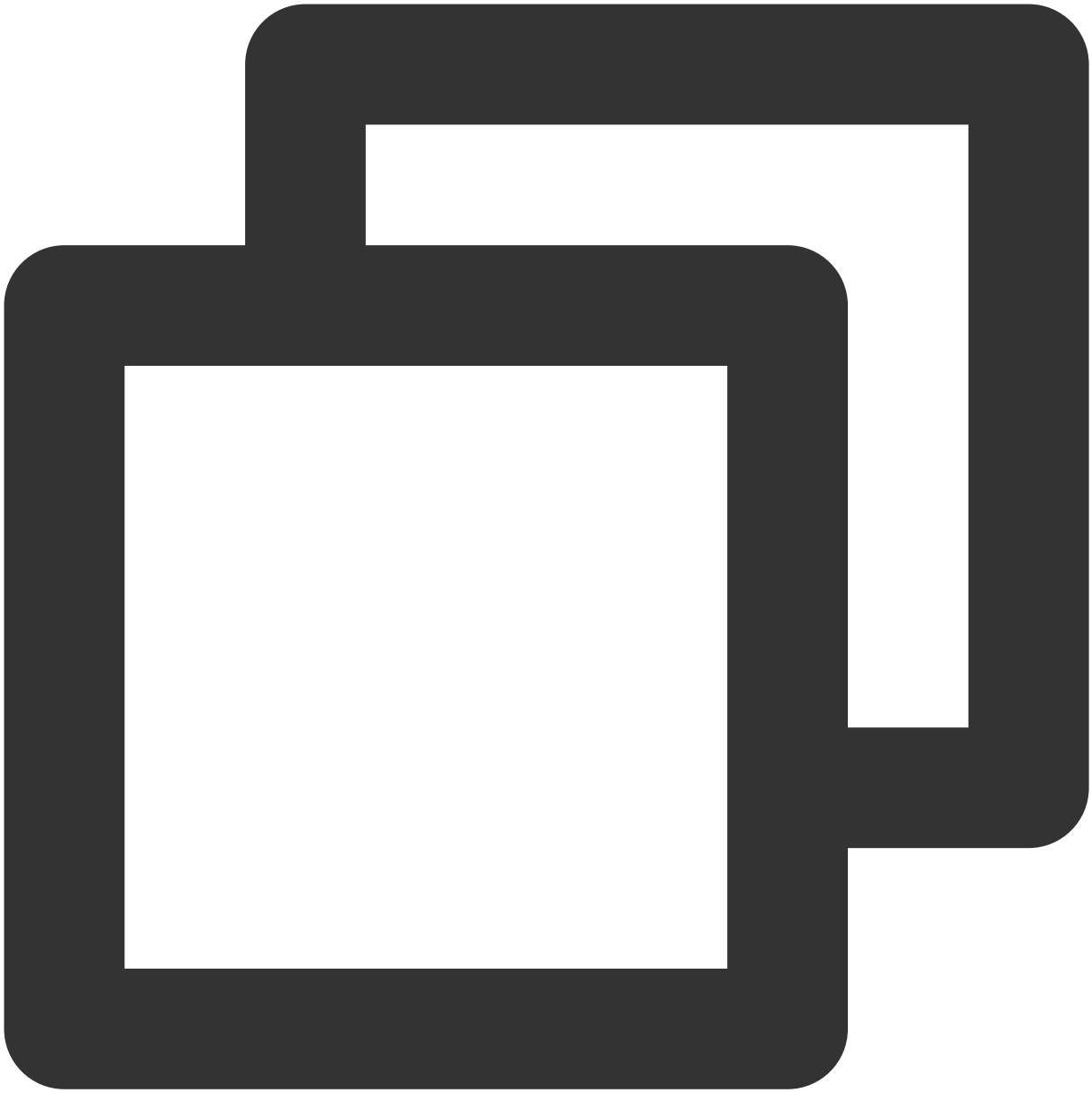
ベストプラクティス
Serverlessクラスター
Elastic IPを介してパブリックネットワークにアクセスします

最終更新日：：2023-04-26 18:43:22

現在、TKE Serverlessでは、PodでのEIPのバインドをサポートしており、これについてはtemplate annotationでのみ説明するだけです。詳細については、Annotationの説明ドキュメントをご参照ください。
EIPに関連するAnnotationマーカーは、次のリストを参照できます。

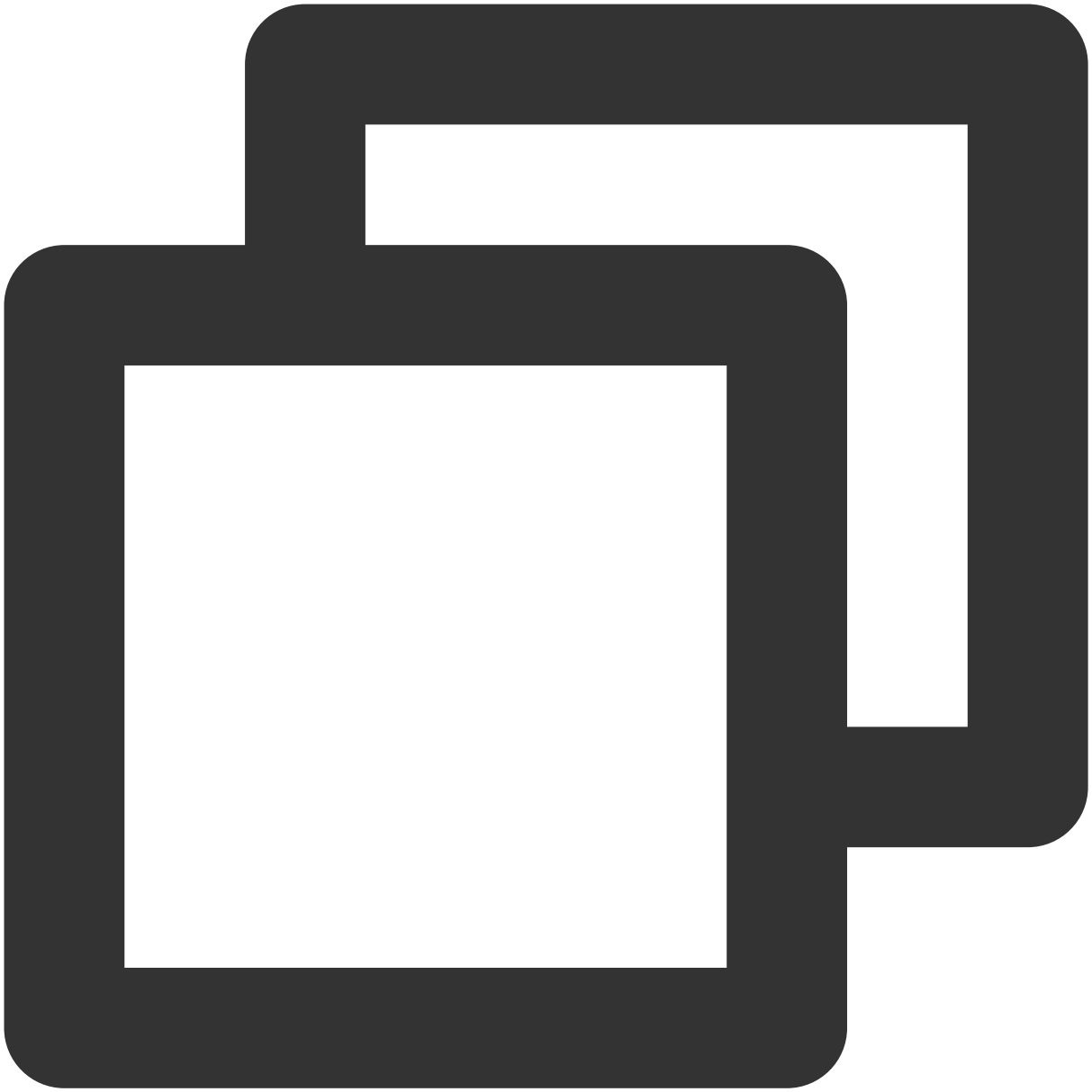
Annotation Key	Annotation Valueおよび説明	入力必須かどうか
eks.tke.cloud.tencent.com/eip-attributes	このWorkloadのPodは、EIPとバインドする必要があることを示しています。値が `""` の場合は、デフォルトのEIP設定を使用して作成されることを示しています。 `""` 内にEIP Tencent Cloud APIパラメータjsonを入力すると、カスタム設定を実装できます。	EIPをバインドする必要がある場合、この項目は入力必須になります
eks.tke.cloud.tencent.com/eip-claim-delete-policy	Podが削除された後、EIPが自動的にリサイクルされるかどうかについては、Neverの場合はリサイクルされず、デフォルトではリサイクルされます。	いいえ
eks.tke.cloud.tencent.com/eip-id-list	ストックEIPが使用され、statefulsetのみがサポートされていることを示しています。デフォルトでは、Podを破棄してもEIPはリサイクルされません。なお、statefulset podの数は、このAnnotationで指定されたeipIdの数までしか指定できないので、ご注意ください。	いいえ

1. WorkloadまたはPodがEIPにバインドしてパブリックネットワークにアクセスする必要がある場合、対応するWorkloadまたはPodの `annotation` の下に `eks.tke.cloud.tencent.com/eip-attributes: ""` というマーカーを追加するのが最も簡単な方法です。以下に例を示します。



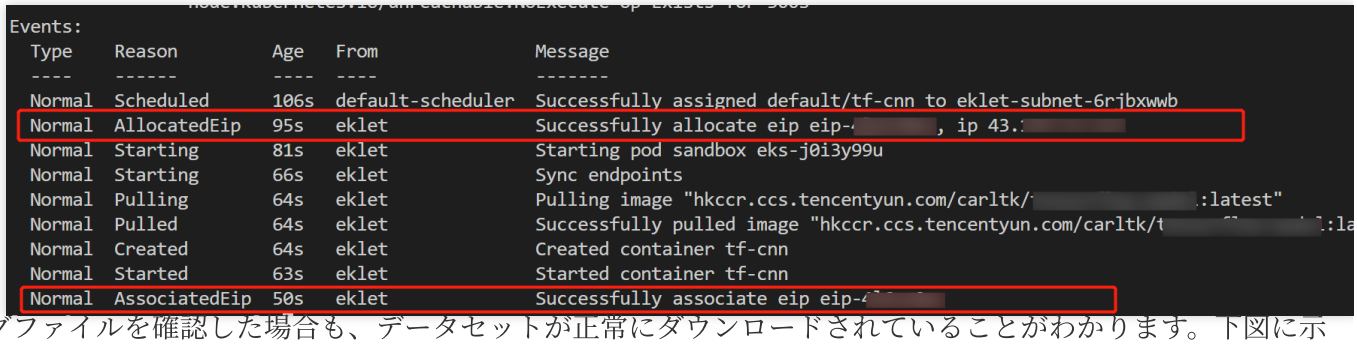
```
metadata:
  name: tf-cnn
  annotations:
    eks.tke.cloud.tencent.com/eip-attributes: ""   #EIPが必要な場合、すべてデフォルトに設
```

2. 以下のコマンドを実行すると、イベントが表示されます。

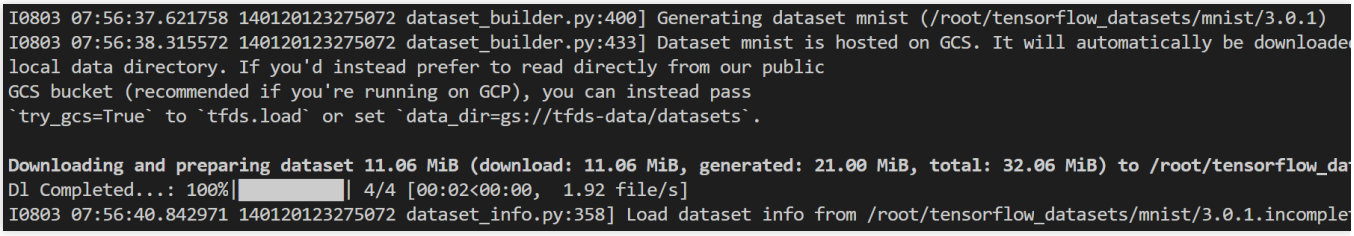


```
kubectl describe pod [name]
```

下の図のように、EIPに関連するイベントが2行追加された場合、実行が成功したことを示しています。



3. ログファイルを確認した場合も、データセットが正常にダウンロードされていることがわかります。下図に示すとおりです。



注意

EIPの申請は1日あたりの上限があり、複数回の実行が必要なタスクには適しません。

Serverlessクラスターでディープラーニングを堪能するディープラーニングコンテナイメージをビルドする

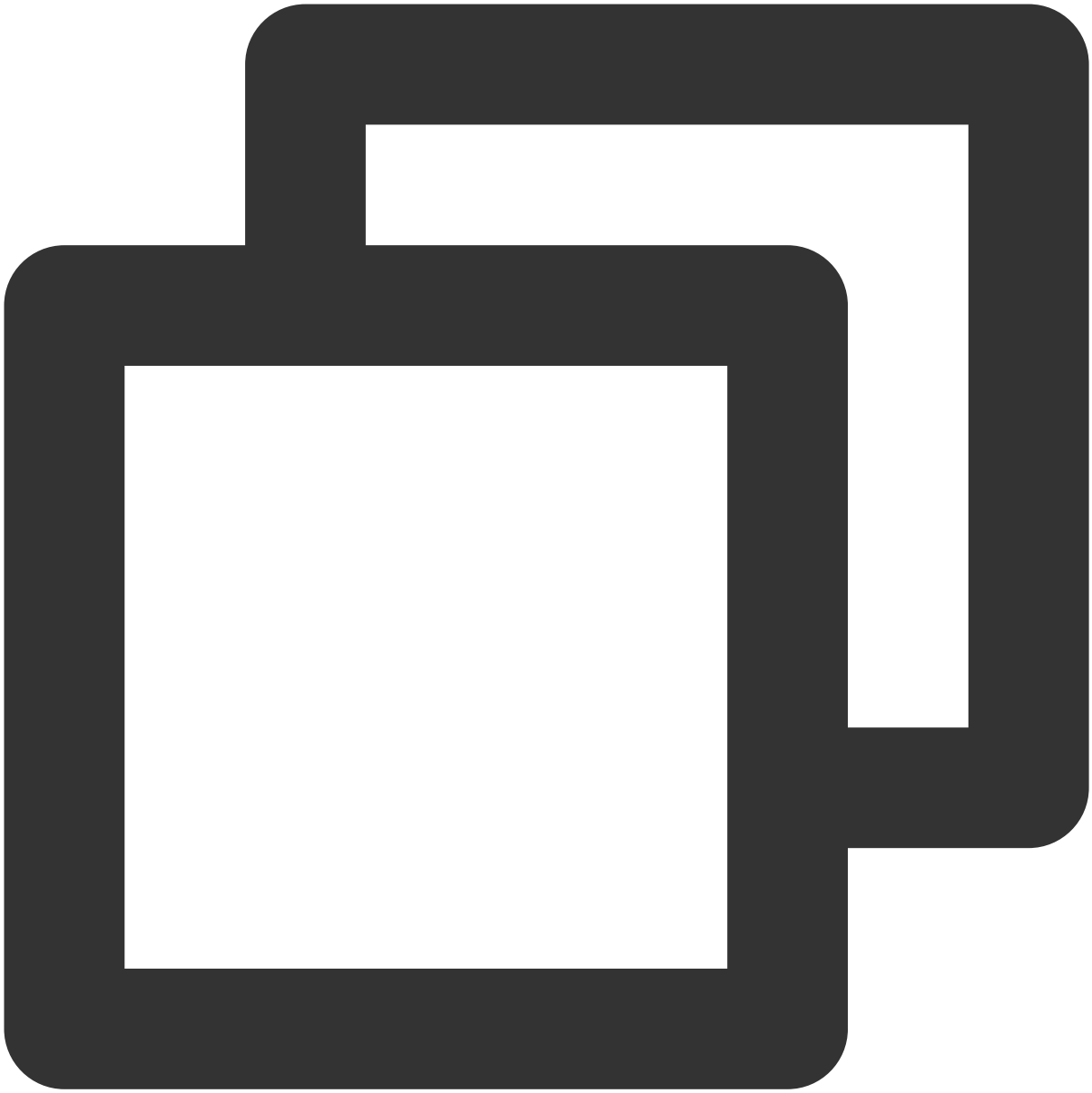
最終更新日：：2023-04-28 15:30:11

概要

このドキュメントではTKE Serverlessクラスターにディープラーニングをデプロイする一連の実践を記録し、TensorFlowをデプロイすることから後続のKubeflowを実現するデプロイまで、完全なコンテナのディープラーニング実践方法を提供することを目的としています。このドキュメントでは自作ディープラーニングのコンテナイメージの構築を重点的に紹介し、後のディープラーニングのデプロイタスクにより手軽でスピーディーな完了方法を提供します。
この実践タスクの要件では、パブリックイメージがディープラーニングをデプロイするニーズを満たすことができないため、この実践では自作イメージを選択します。
ディープラーニングフレームワークTensorFlow-gpu以外に、このイメージはさらにGPU トレーニングに必要なcuda、cudnnを含み、TensorFlow公式が提供するディープラーニングモデルを統合します。現在CV、NLP、RSなどの分野のSOTAモデルが含まれています。モデルの詳細については、Model Garden for TensorFlowをご参照ください。

操作手順

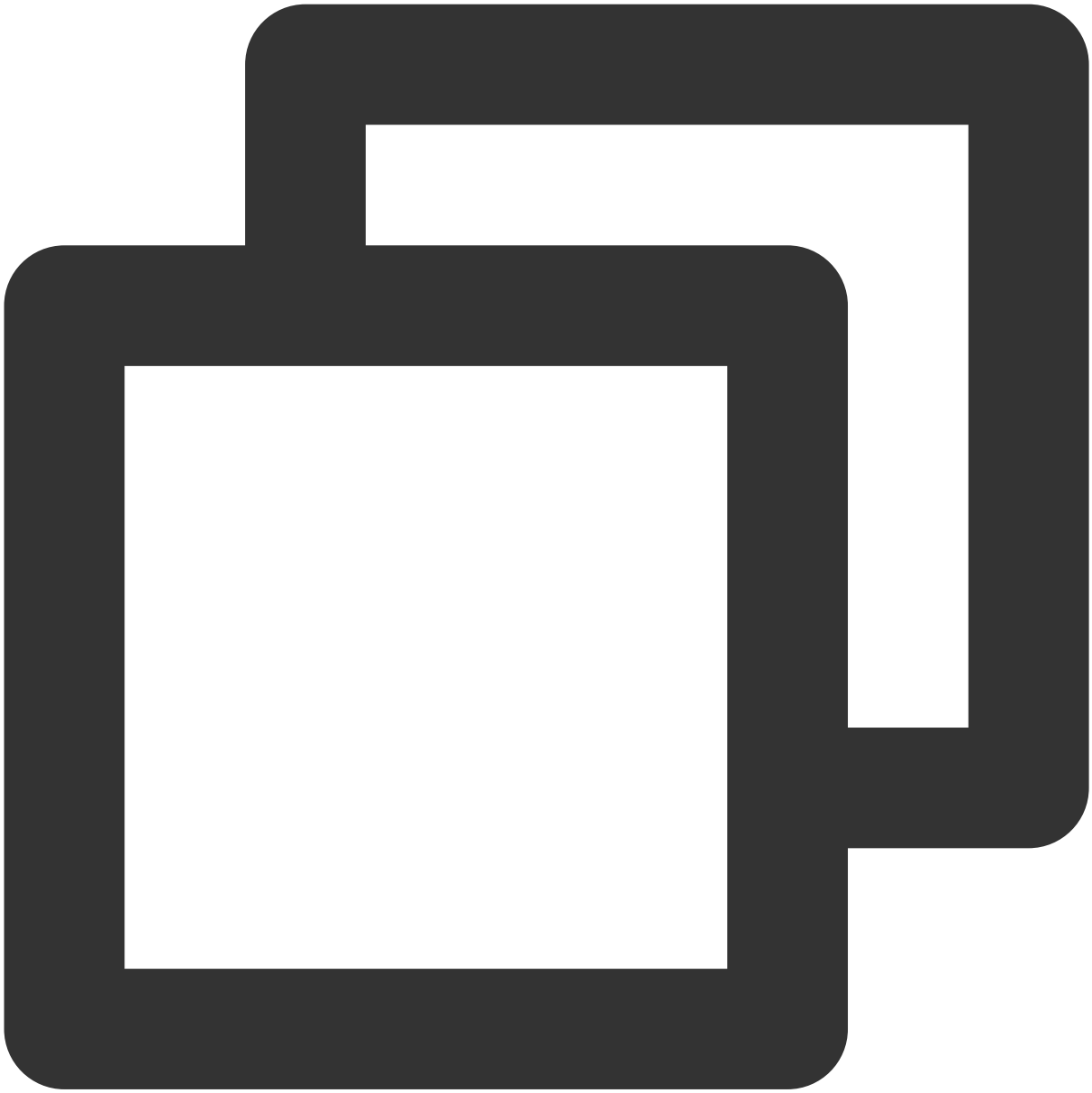
1. この例ではDocker コンテナによってイメージを構築します。Dockerfileファイルを準備し、例は次のとおりです。



```
FROM nvidia/cuda:11.3.1-cudnn8-runtime-ubuntu20.04
RUN apt-get update -y \\
 && apt-get install -y python3 \\
    python3-pip \\
    git \\
 && git clone git://github.com/tensorflow/models.git \\
 && apt-get --purge remove -y git \\    #不要なコンポーネントは適時アンインストールする（オフ
 && rm -rf /var/lib/apt/lists/*         #aptインストール用のインストールパッケージを削除する
 && mkdir /tf /tf/models /tf/data       #保存モデルおよびデータを作成するパス。マウントポイン
ENV PYTHONPATH $PYTHONPATH:/models
ENV LD_LIBRARY_PATH $LD_LIBRARY_PATH:/usr/local/cuda-11.3/lib64:/usr/lib/x86_64-lin
```

```
RUN pip3 install --user -r models/official/requirements.txt \\
 && pip3 install tensorflow
```

2. 次のコマンドを実行してデプロイを行います。



```
docker build -t [name]:[tag] .
```

説明

例えばPython、TensorFlow、cuda、cudnnおよびモデルライブラリなどの必要なコンポーネントのインストールステップは、ここでは説明を省略します。

関連説明

イメージ関連

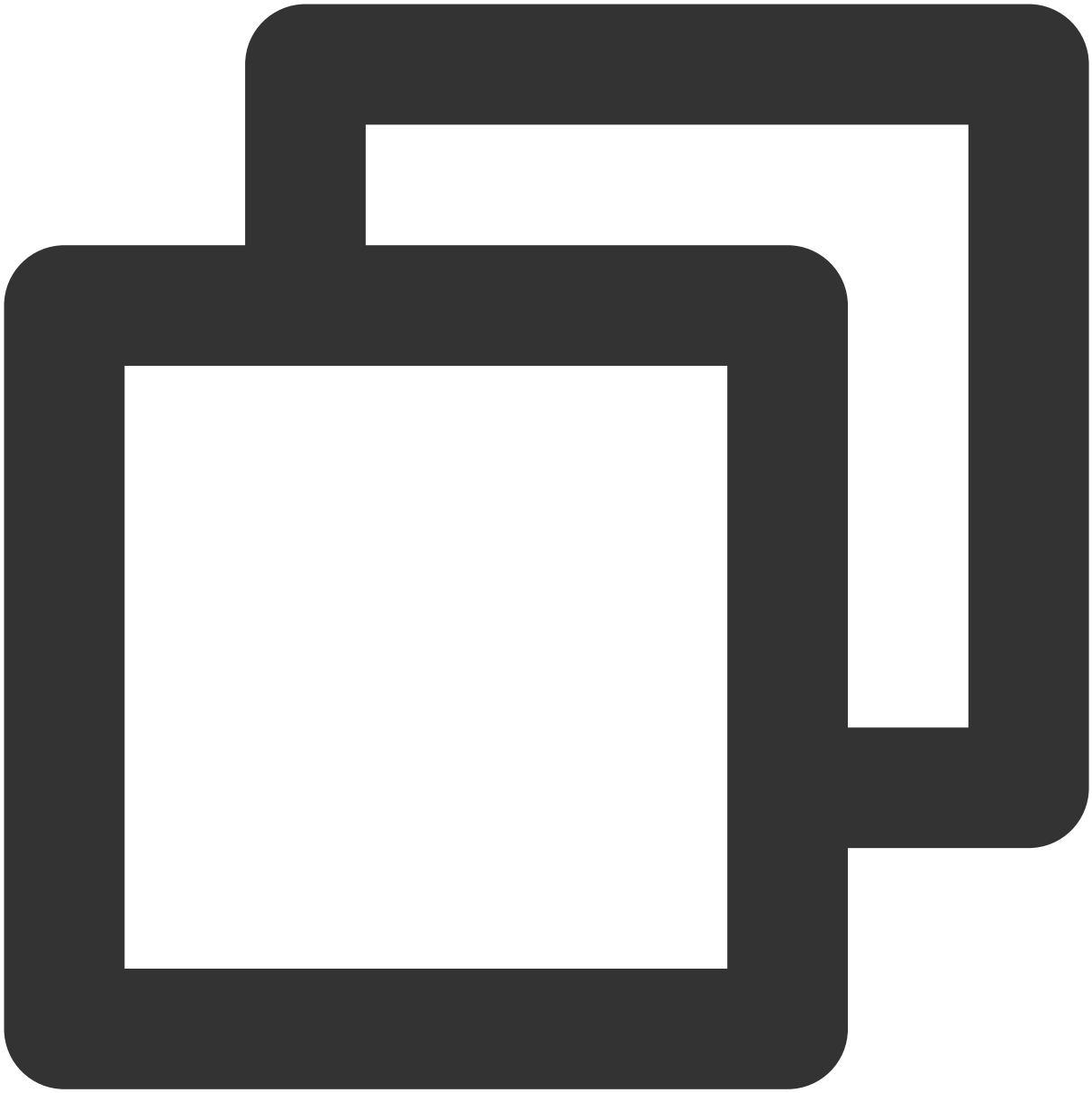
基本イメージ nvidia/cudaについて、CUDAコンテナイメージはCUDAがサポートするプラットフォームおよびアーキテクチャは使用しやすい配信版を提供します。この選択はcuda 11.3.1、cudnn 8の組み合わせです。その他のバージョン選択については、Supported tagsをご参照ください。

環境変数

このドキュメントのベストプラクティスを行う場合、重点的に環境変数 `LD_LIBRARY_PATH` に注目する必要があります。
`LD_LIBRARY_PATH` はダイナミックリンクライブラリのインストールパスで、通常libxxxx.soの形式です。ここでは主にcudaおよびcudnnをリンクします。例えばlibcudart.so.[version]、ibcusolver.so.[version]、libcudnn.so.[version] などです。 `ll` コマンドを実行して確認することができ、下図に示すとおりです。

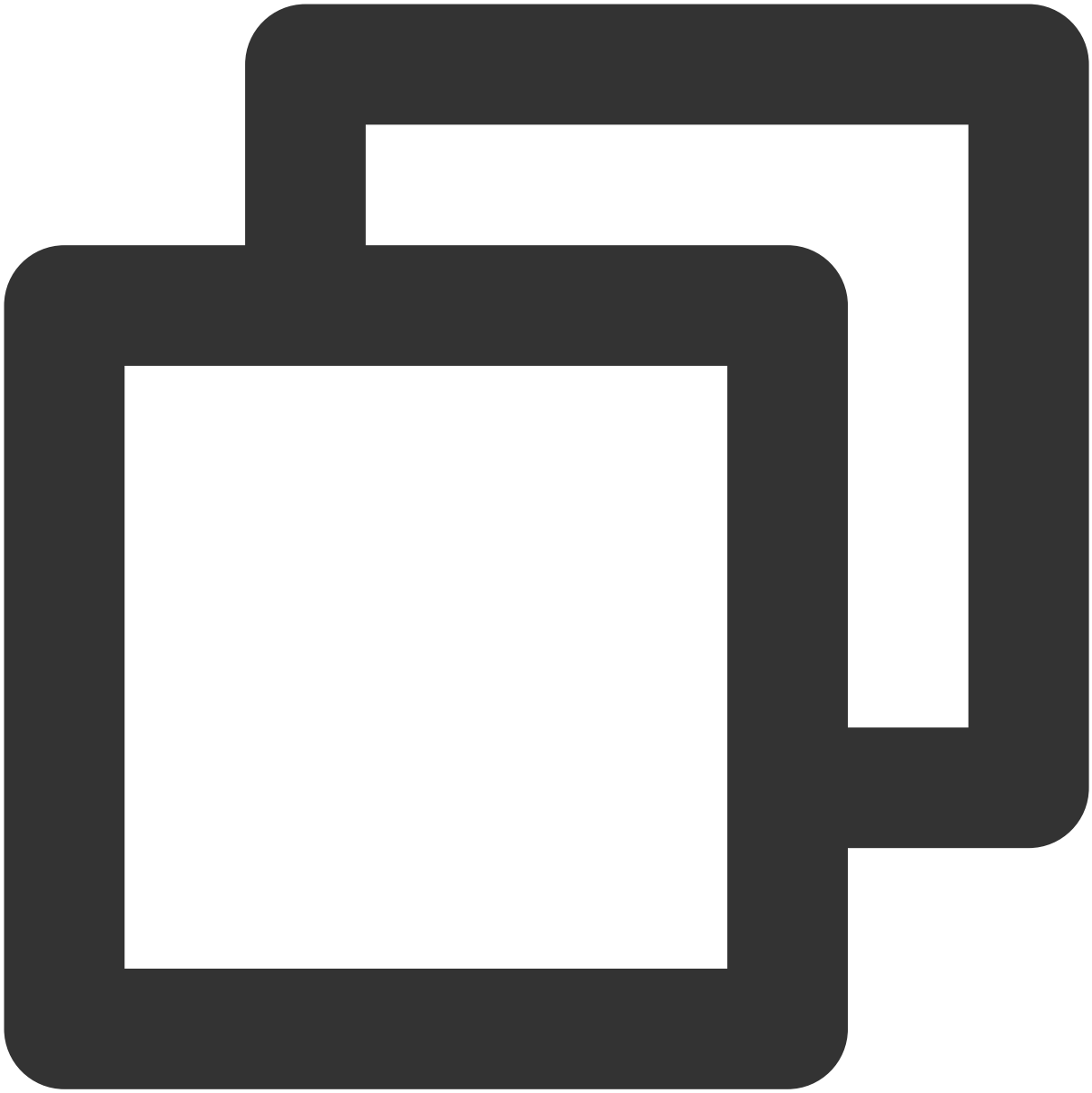
```
root@5a949761c669:/usr/local/cuda-11.3/lib64# ll
total 1534336
drwxr-xr-x 1 root root      4096 Jul  2 03:57 ./
drwxr-xr-x 1 root root      4096 Jul  2 03:57 ../
lrwxrwxrwx 1 root root        16 May  4 02:30 libOpenCL.so.1 -> libOpenCL.so.1.0
lrwxrwxrwx 1 root root        18 May  4 02:30 libOpenCL.so.1.0 -> libOpenCL.so.1.0.0
-rw-r--r-- 1 root root     30856 May  4 02:30 libOpenCL.so.1.0.0
lrwxrwxrwx 1 root root        23 May 13 23:26 libcublas.so.11 -> libcublas.so.11.5.1
-rw-r--r-- 1 root root 121866104 May 13 23:26 libcublas.so.11.5.1.109
lrwxrwxrwx 1 root root        25 May 13 23:26 libcublasLt.so.11 -> libcublasLt.so.11
-rw-r--r-- 1 root root 263770264 May 13 23:26 libcublasLt.so.11.5.1.109
lrwxrwxrwx 1 root root        21 May  4 02:30 libcudart.so.11.0 -> libcudart.so.11.3
-rw-r--r-- 1 root root    619192 May  4 02:30 libcudart.so.11.3.109
lrwxrwxrwx 1 root root        22 May 13 23:30 libcufft.so.10 -> libcufft.so.10.4.2.1
-rw-r--r-- 1 root root 190417864 May 13 23:30 libcufft.so.10.4.2.109
lrwxrwxrwx 1 root root        23 May 13 23:30 libcufftw.so.10 -> libcufftw.so.10.4.2
-rw-r--r-- 1 root root    631888 May 13 23:30 libcufftw.so.10.4.2.109
```

公式イメージのDockerfileソースコードに基づいて以下のコマンドを実行します



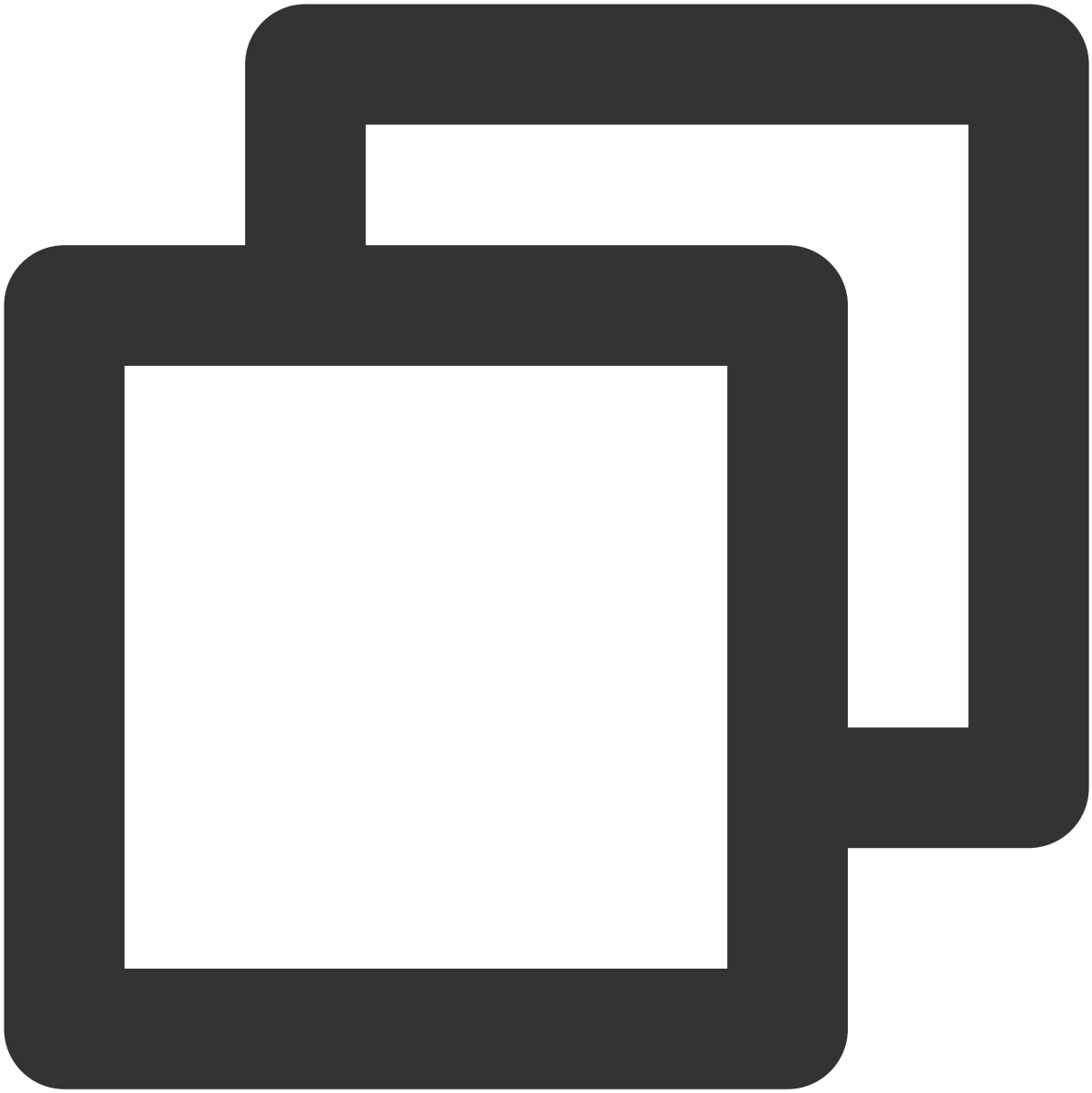
```
ENV LD_LIBRARY_PATH /usr/local/nvidia/lib:/usr/local/nvidia/lib64
```

このうち、`/usr/local/nvidia/lib` はcudaパスのソフトリンクを指し、cudaに用意されています。cudnnを付帯するバージョンはcudnnのインストールのみを行い、cudnnには `LD_LIBRARY_PATH` が指定されていないため、Warningエラーが発生し、GPU リソースが使用できなくなり、エラーは次のように表示されます。



```
Could not load dynamic library 'libcudnn.so.8'; dlerror: libcudnn.so.8: cannot open
Cannot dlopen some GPU libraries. Please make sure the missing libraries mentioned
```

このようなエラーが発生した場合、手動でcudnnパスの追加を試みることができます。ここで以下のコマンドを実行してイメージを実行し、libcudnn.soが所在するパスを確認できます。



```
docker run -it nvidia/cuda:[tag] /bin/bash
```

ソースコードからわかるように、cudnnは `apt-get install` コマンドによってインストールされ、デフォルトで `/usr/lib` にインストールされます。この例ではlibcudnn.so.8の実際のパスは `/usr/lib/x86_64-linux-gnu#` 下で、コロンで後方に補足されます。
バージョンのシステムが異なるなどの原因により、実際のパスにばらつきが生じ、ソースコードおよび実際の観察が難しくなる可能性があります。

後続の操作

後続の操作はTKE Serverlessでディープラーニングを行うというドキュメントをご参照ください。

よくあるご質問

この実践過程で発生した問題は、よくあるご質問というドキュメントを参照してトラブルシューティングを行って解決してください。

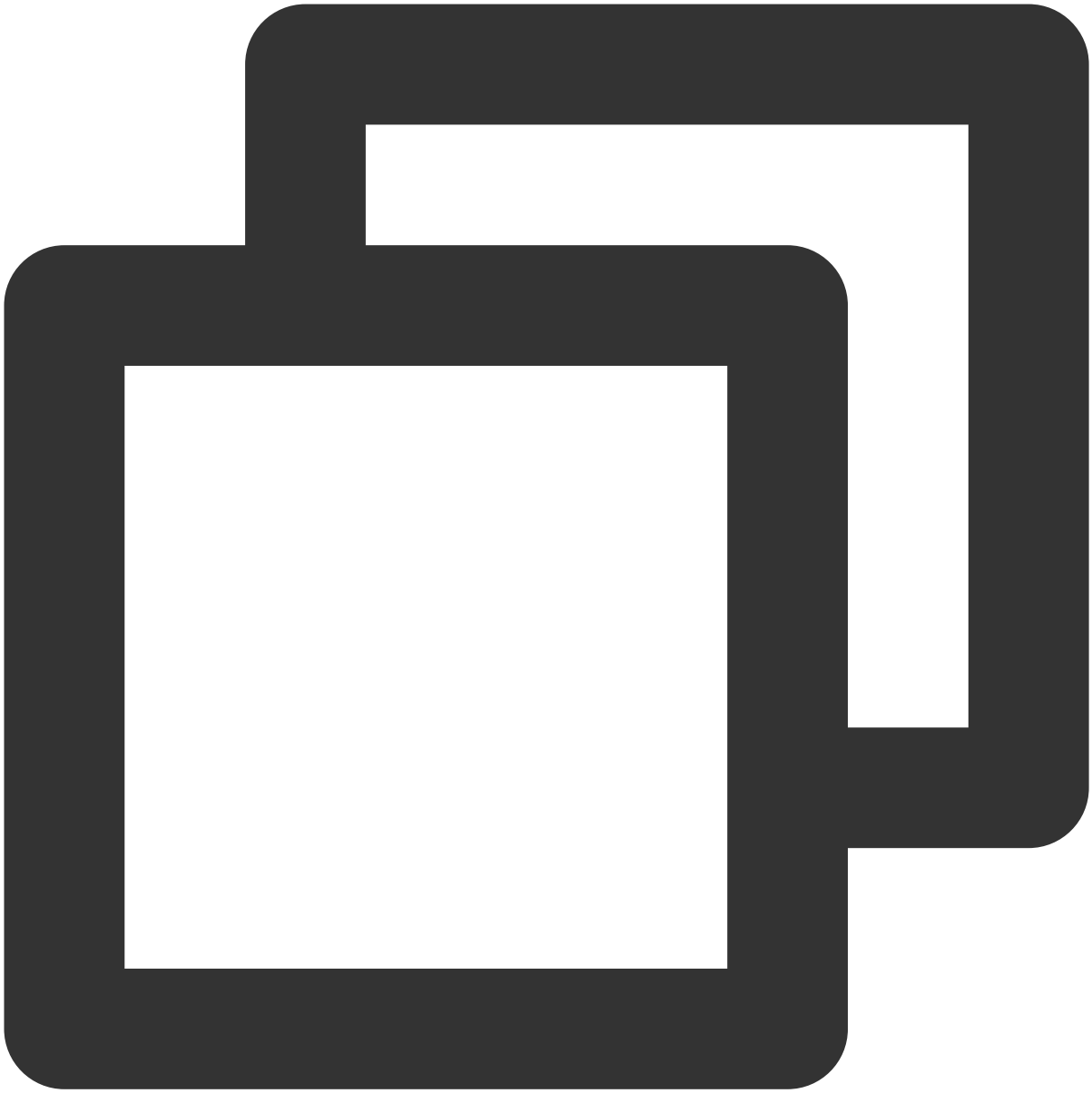
よくあるご質問
パブリックネットワークアクセス関連

最終更新日：：2023-04-28 15:30:11

ここではディープラーニングのコンテナイメージの構築およびTKE Serverlessでディープラーニングを行うときに発生する可能性があるよくあるご質問に解答します。

コンテナはどのようにパブリックネットワークにアクセスするのですか。

タスクプロセスでトレーニング用データセットをダウンロードする必要があるため、パブリックネットワークにアクセスする操作が必要な場合があります。コンテナの初期状態ではパブリックネットワークにアクセスすることはできず、直接データセットをダウンロードするコマンドを実行すると、次のようなエラーが報告されます。



```
W tensorflow/core/platform/cloud/google_auth_provider.cc:184] All attempts to get a
E tensorflow/core/platform/cloud/curl_http_request.cc:614] The transmission  of req
```

上記の問題に対し、2種類のパブリックネットワークにアクセスする方法を提供しています。

NAT Gatewayを使用する：あるVPC下の複数のインスタンスがパブリックネットワークと通信する必要がある場合に適用されます。NAT Gatewayによるパブリックネットワークへのアクセス というドキュメントに従って操作してください。

注意

作成されたNAT GatewayおよびルートテーブルはTKE Serverlessクラスターと同じリージョン、同じプライベートネットワークVPCである必要があります。
Elastic IP（EIP）を使用する：単一または少数のインスタンスがパブリックネットワーク相互接続を実現する必要がある場合に適用されます。EIPを使用してパブリックネットワークにアクセスするというドキュメントに従って操作してください。

ネットワーク
DNS関連
TKE DNSベストプラクティス

最終更新日：：2023-04-28 15:30:11

まとめ

DNSはKubernetesクラスター内のサービスアクセスの第一部分として、その安定性およびパフォーマンスが極めて重要です。どのようにしてより優れた方法でDNSを設定および使用するかは、さまざまな側面があります。このドキュメントではこれらのベストプラクティスをまとめます。

最適なCoreDNSバージョンを選択する

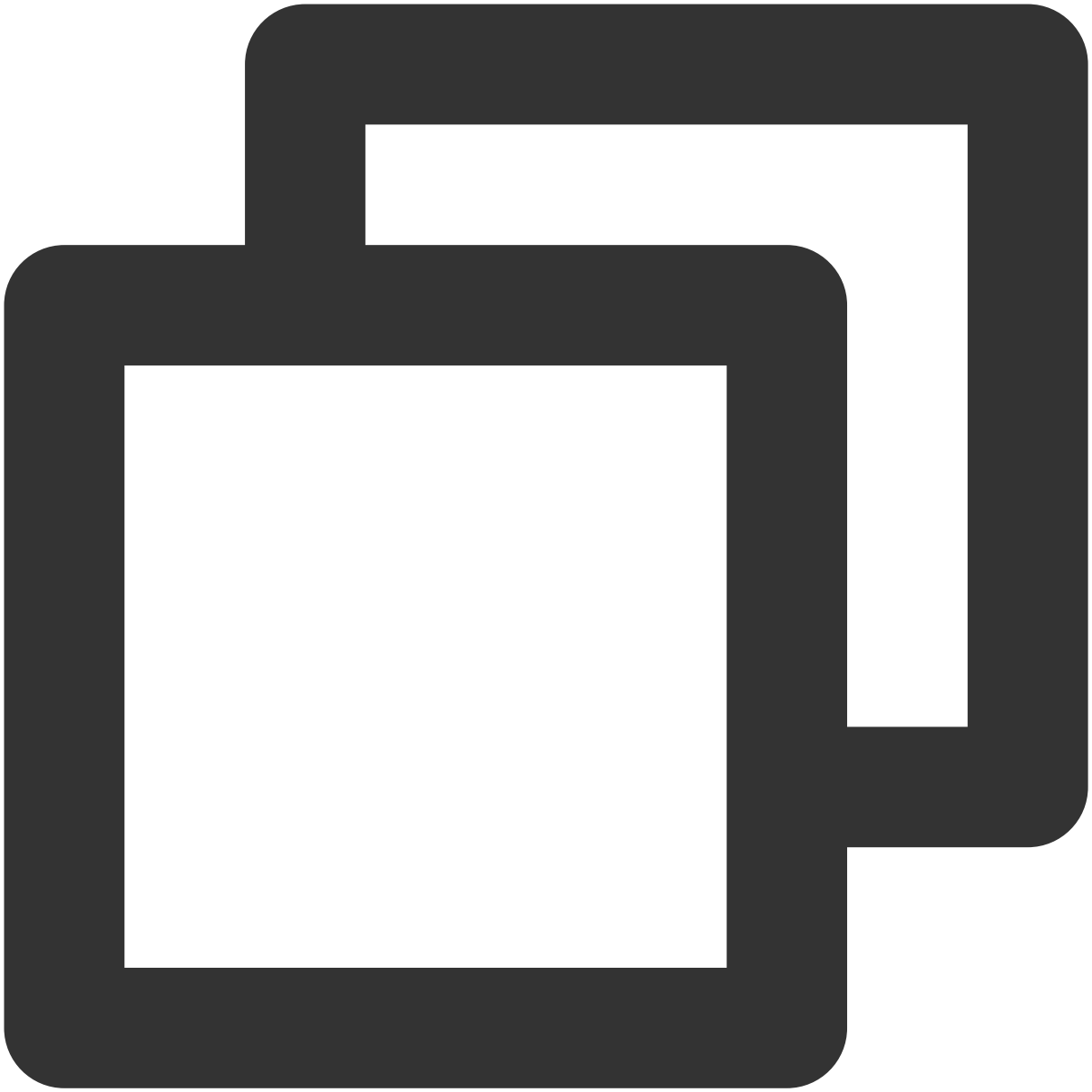
以下の表では各バージョンのTKEクラスターと共にデフォルトでデプロイされるCoreDNSバージョンを示しています。

TKE Version	CoreDNS version
v1.22	v1.8.4
v1.20	v1.8.4
v1.18	v1.7.0
v1.16	v1.6.2
v1.14	v1.6.2

履歴的原因により、バージョンv1.18以上のクラスターにv1.6.2バージョンのCoreDNSがデプロイされている可能性があります。現在のCoreDNS バージョンがニーズを満たさない場合、以下のガイドに従って手動でアップグレードできます。
1.7.0にアップグレード
1.8.4にアップグレード

適切なCoreDNSレプリカ数を設定する

1. TKEはデフォルトでCoreDNSレプリカ数が2に設定され、かつ `podAntiAffinity` を設定して2つのレプリカが異なるノードにデプロイされます。
2. ノード数が80以上のクラスターに対して、NodeLocal DNSCacheをインストールすることをお勧めします。詳細はTKEクラスター内でNodeLocal DNS Cacheを使用するをご参照ください
3. 一般的にクラスター内の業務がDNSにアクセスするQPSによってCoreDNSの合理的なレプリカ数を確定します。ノード数および総コア数によって確定することもできます。NodeLocal DNSCacheをインストールした後、CoreDNSの最大レプリカ数を10にすることをお勧めします。以下の方法に従って設定することができます。
レプリカ数=min(max(ceil(QPS/10000)、ceil(クラスターノード数/8))、10)
事例：
クラスターノード数が10、DNSサービスリクエストQPSが 22000の場合、レプリカ数は3です
クラスターノード数が30、DNSサービスリクエストQPSが15000の場合、レプリカ数は4です
クラスターノード数が100、DNSサービスリクエストQPSが50000の場合、レプリカ数は10です（NodeLocal DNSCacheをデプロイ済み）
4. コンソールでDNSAutoScalerコンポーネントをインストールすることによって、CoreDNSレプリカ数（事前にスムーズアップグレードを設定するように注意してください）の自動調整を実現することができます。コンポーネントのデフォルト設定は次のとおりです。



```
data:
  ladder: |-
 {
   "coresToReplicas":
   [
     [ 1, 1 ],
     [ 128, 3 ],
     [ 512,4 ],
   ],
   "nodesToReplicas":
   [
```

```
      [ 1, 1 ],
      [ 2, 2 ]
    ]
  }
```

NodeLocal DNSCacheを使用する

TKEクラスター内にNodeLocal DNSCacheをデプロイすると、サービスが検出する安定性およびパフォーマンスを向上させることができます。それはノード上でDaemonSetとしてDNSキャッシュプロキシを実行することによってクラスターDNSのパフォーマンスを向上させます。
その他のNodeLocal DNSCacheの紹介およびTKEクラスター内でNodeLocal DNSCacheをデプロイする具体的なステップについては、以下をご参照ください。TKEクラスター内でNodeLocal DNS Cacheを使用する

CoreDNSのスムーズアップグレードを設定する

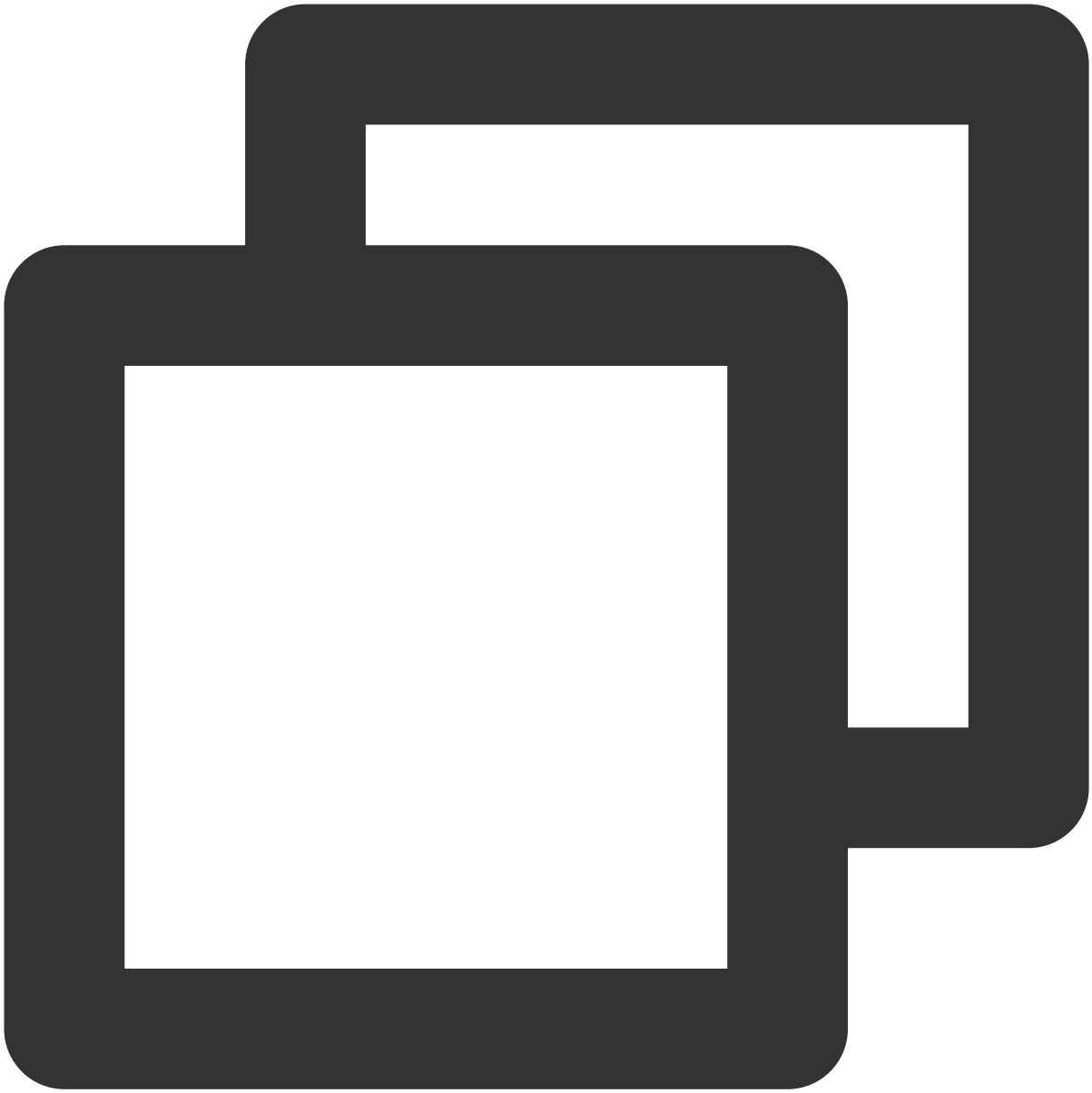
ノードの再起動またはCoreDNSをアップグレードする時に、CoreDNSの一部のレプリカが一定時間使用できなくなる可能性があります。以下の設定によって、DNSサービスの可用性を最大限保証し、スムーズなアップグレードを実現します。

kube-proxyはiptablesモードです。設定する必要はありません

iptablesモードでは、kube-proxyはiptablesルールを同期した後タイムリーに保持されているconntrackエントリーをクリーンアップするため、セッション維持の問題は存在せず、設定の必要がありません。

kube-proxyはIPVSモードです。IPVS UDPプロトコルのセッション維持タイムアウト時間を設定します

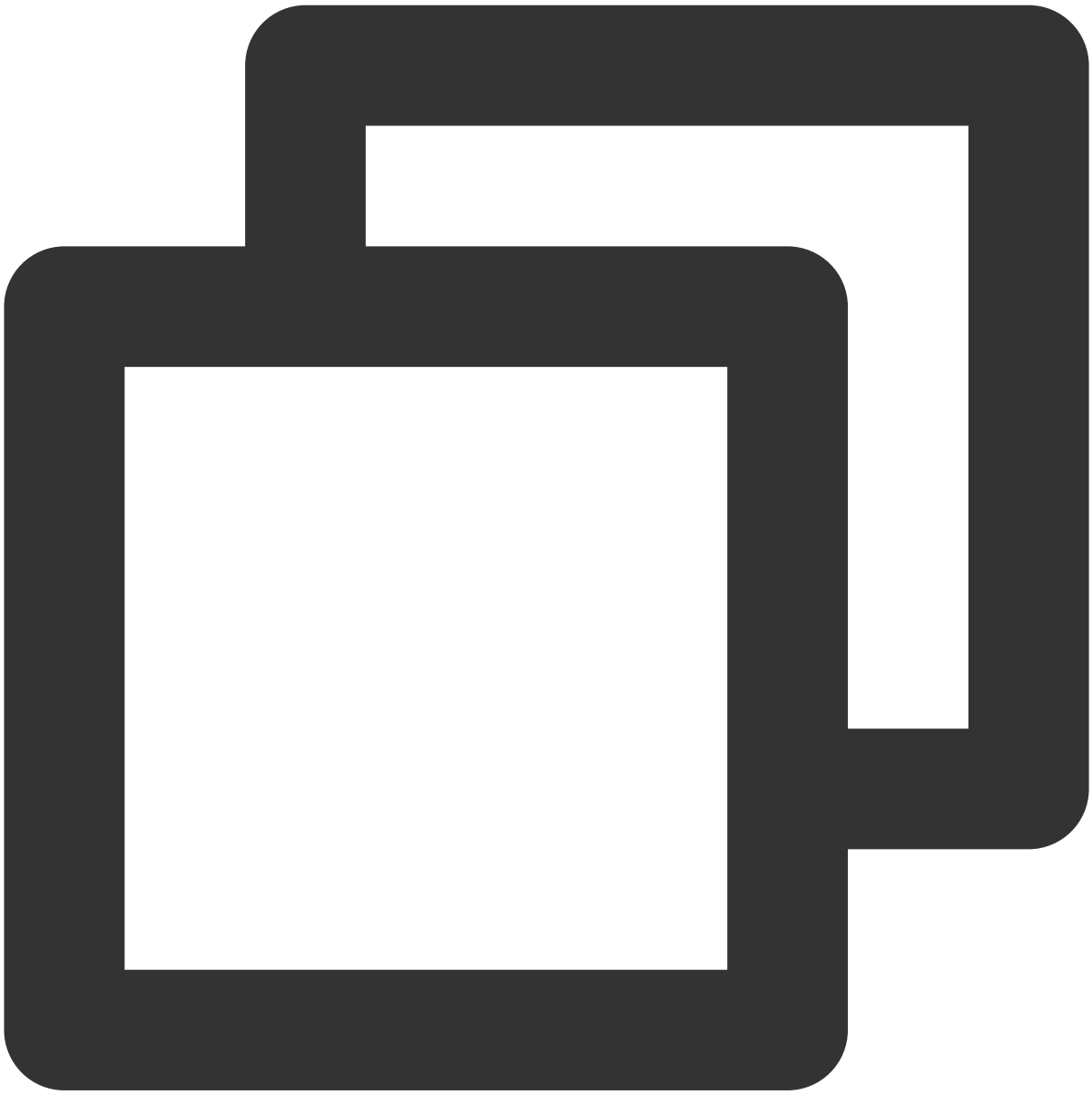
IPVSモードでは、業務自身にUDPサービスがない場合、IPVS UDPプロトコルのセッション維持タイムアウト時間を低下させることによってサービスが利用できなくなる時間を可能な限り減少させます。
1. クラスターバージョンが1.18以上の場合、kube-proxyはパラメータ `--ipvs-udp-timeout` を提供します。デフォルトは0sですが、システムのデフォルト値：300sを使用し、 `--ipvs-udp-timeout=10s` に設定することを推奨します。以下の方法に従ってkube-proxy DaemonSetを設定します。



```
spec:
  containers:
  - args:
    - --kubeconfig=/var/lib/kube-proxy/config
    - --hostname-override=$(NODE_NAME)
    - --v=2
    - --proxy-mode=ipvs
    - --ipvs-scheduler=rr
    - --nodeport-addresses=$(HOST_IP)/32
    - --ipvs-udp-timeout=10s
    command:
```

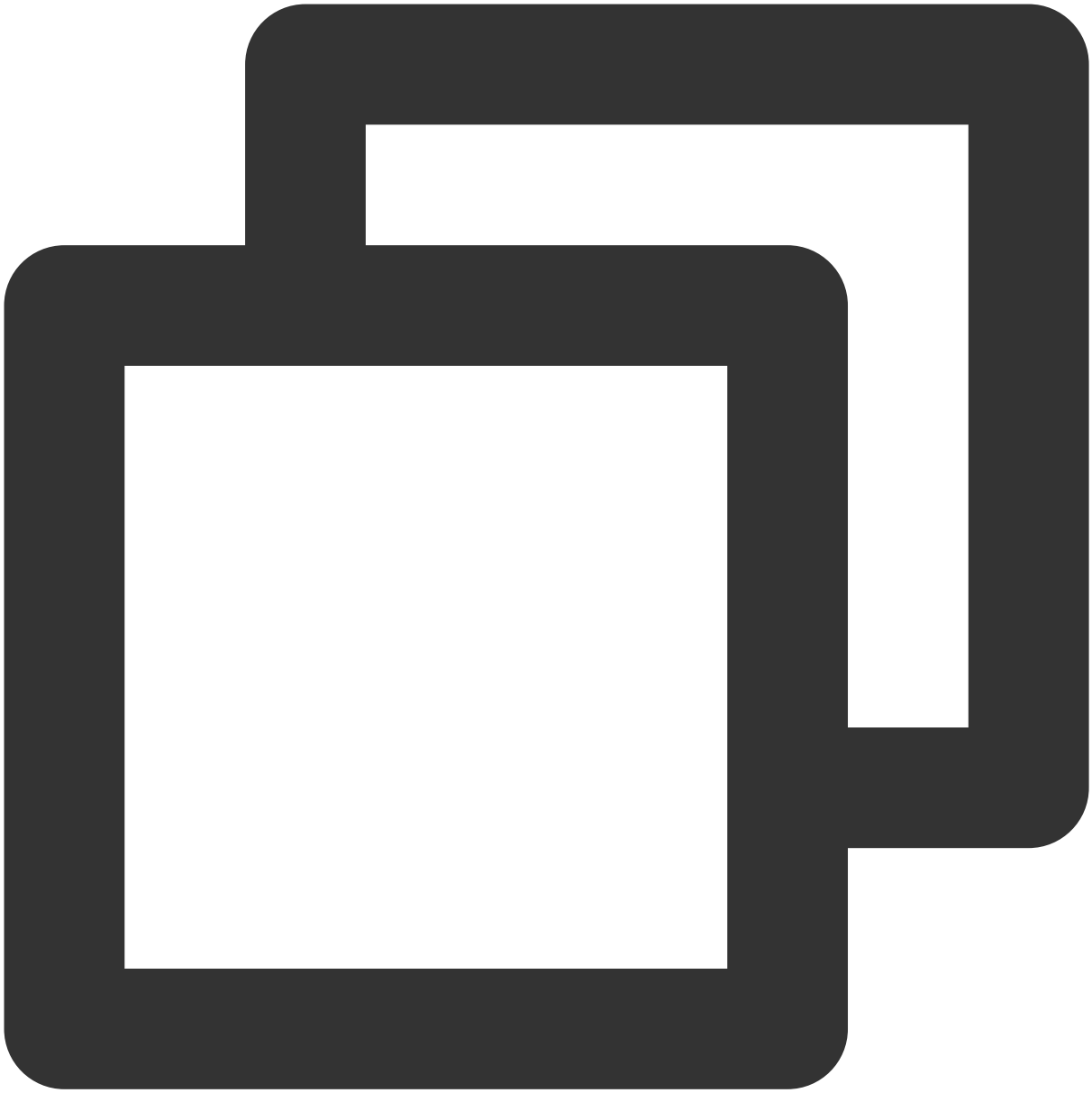
```
      - kube-proxy
      name: kube-proxy
```

2. クラスターバージョンが1.16以下の場合、kube-proxyはこのパラメータをサポートしません。 `ipvsadm` ツールを使用してノード側で一括変更することができます：



```
yum install -y ipvsadm
ipvsadm --set 900 120 10
```

3. 設定が完了したら、以下の方法で検証します。



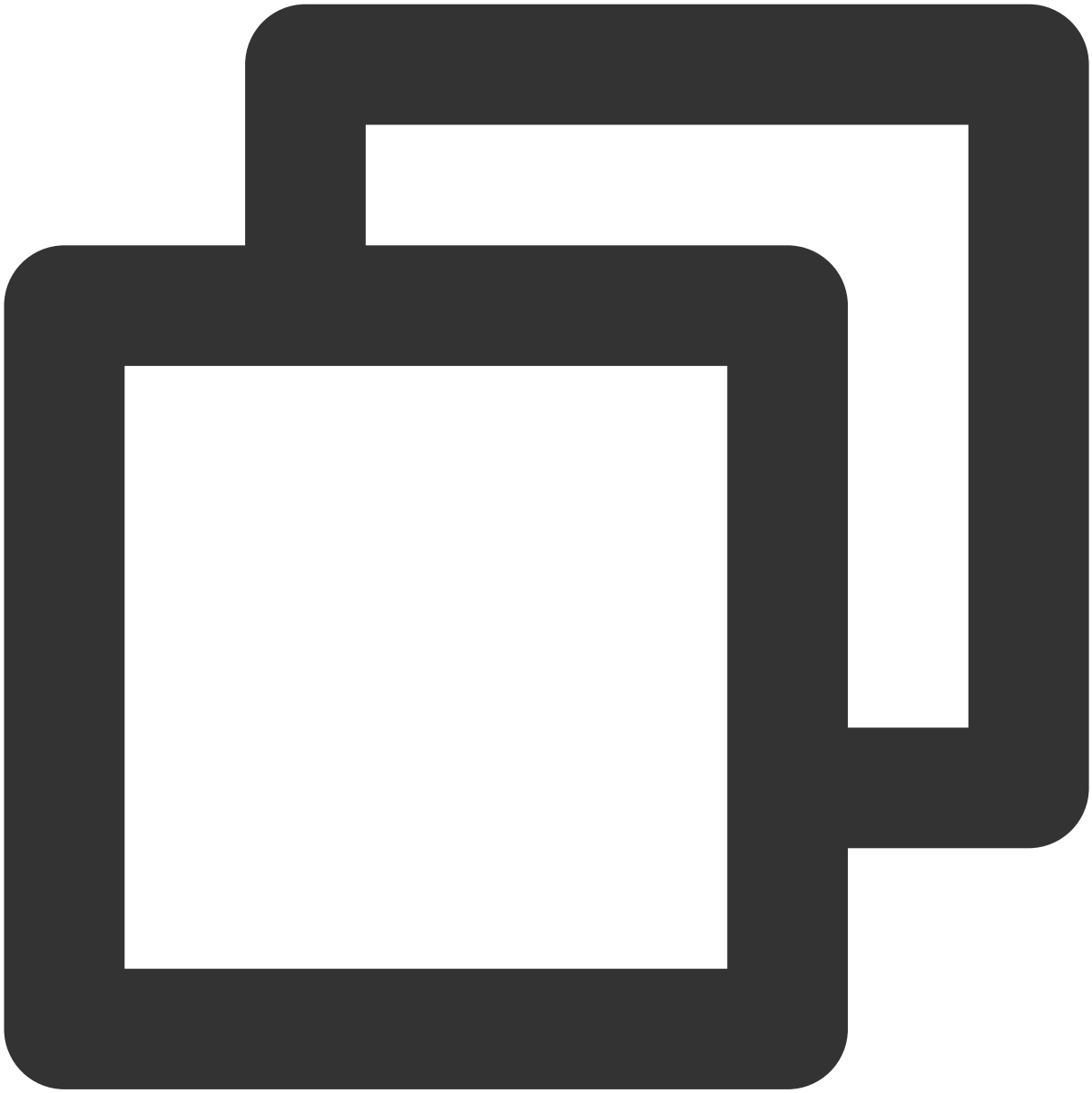
```
ipvsadm -L --timeout
Timeout (tcp tcpfin udp): 900 120 10
```

注意

設定が完了したら、5分待ち、その後のステップを継続する必要があります。業務にUDPサービスを使用している場合は、チケットを提出してサポートを受けることができます。

CoreDNSのグレースフルシャットダウンを設定する

終了信号を受信したレプリカは、lameduckを設定することにより一定時間内でサービスの提供を継続します。以下の方法に従ってCoreDNSのconfigmapを設定します（CoreDNSバージョン1.6.2の一部の設定のみを表示します。その他のバージョンの設定はCoreDNSの手動アップグレード をご参照ください）。



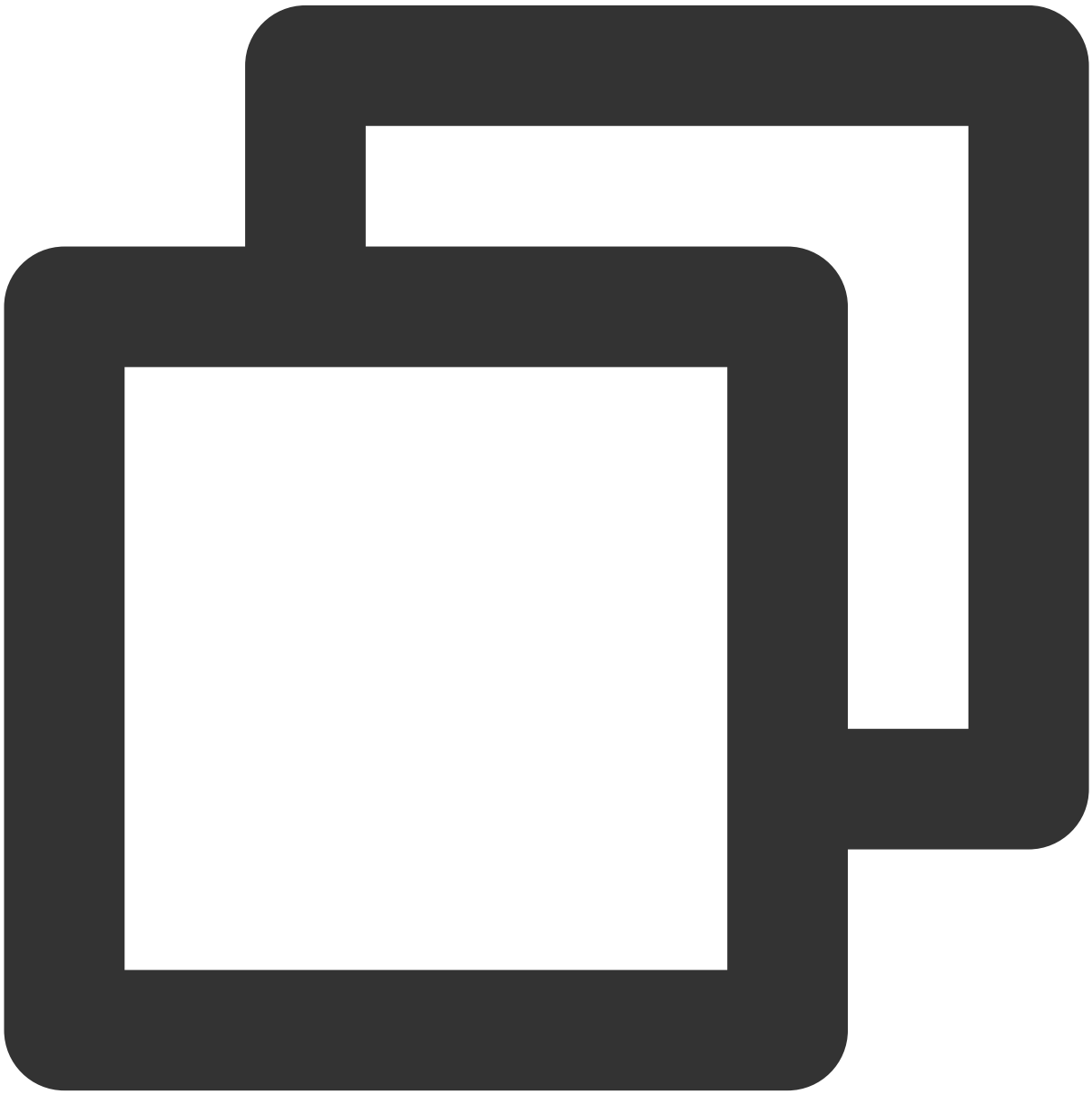
```
        .:53 {
            health {
                lameduck 30s
            }
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                upstream
```

```
                fallthrough in-addr.arpa ip6.arpa
            }
        }
```

CoreDNS サービス準備の確認を設定する

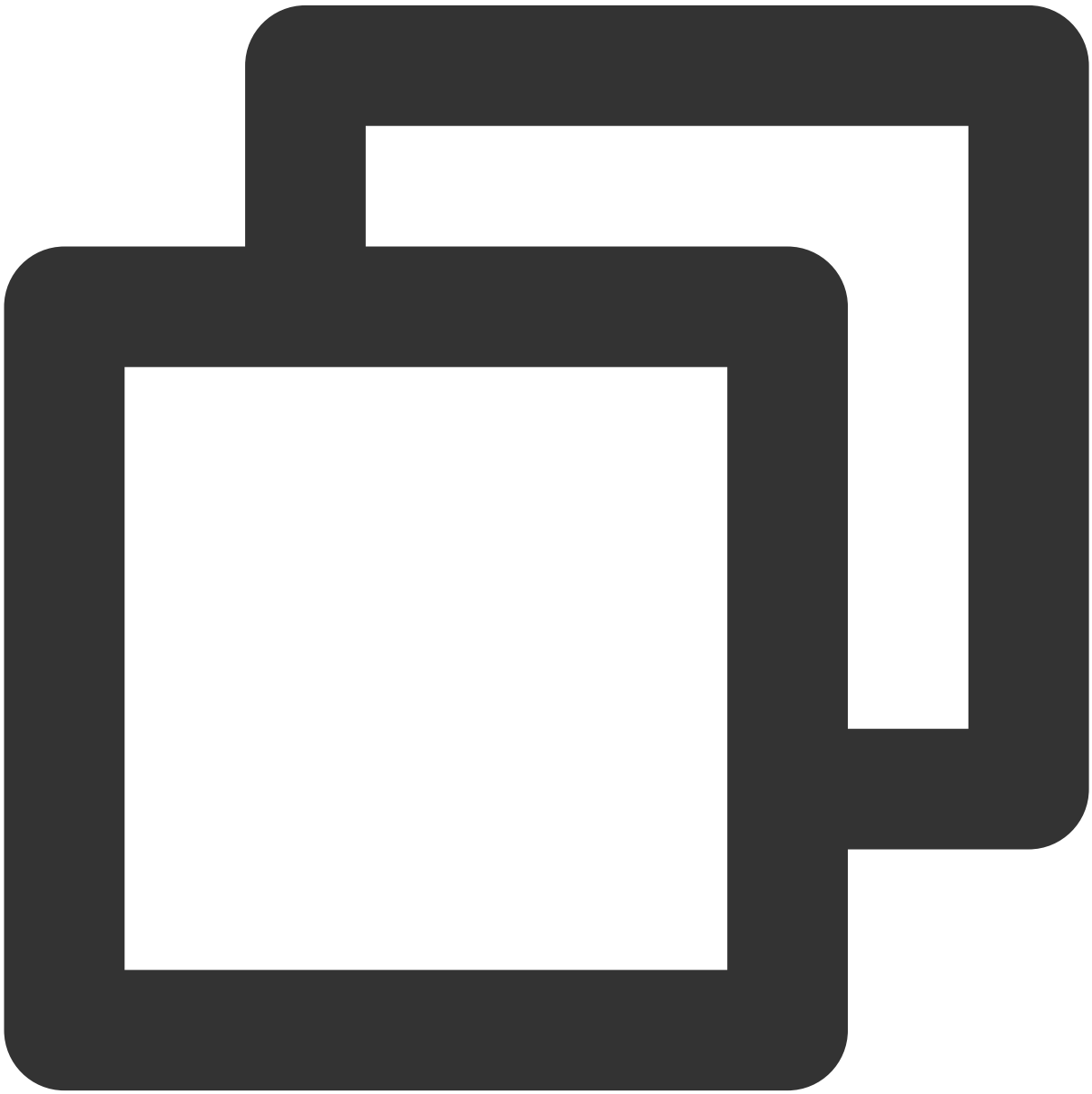
新しいレプリカを起動した後、そのサービスの準備を確認し、さらにDNSサービスのバックエンドリストを追加する必要があります。

1. readyプラグインを開き、以下の方法に従ってCoreDNSのconfigmapを設定します（CoreDNS 1.6.2バージョンの一部の設定のみを表示します。その他のバージョン設定はCoreDNSの手動アップグレードをご参照ください）。



```
        .:53 {
            ready
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                upstream
                fallthrough in-addr.arpa ip6.arpa
            }
        }
```

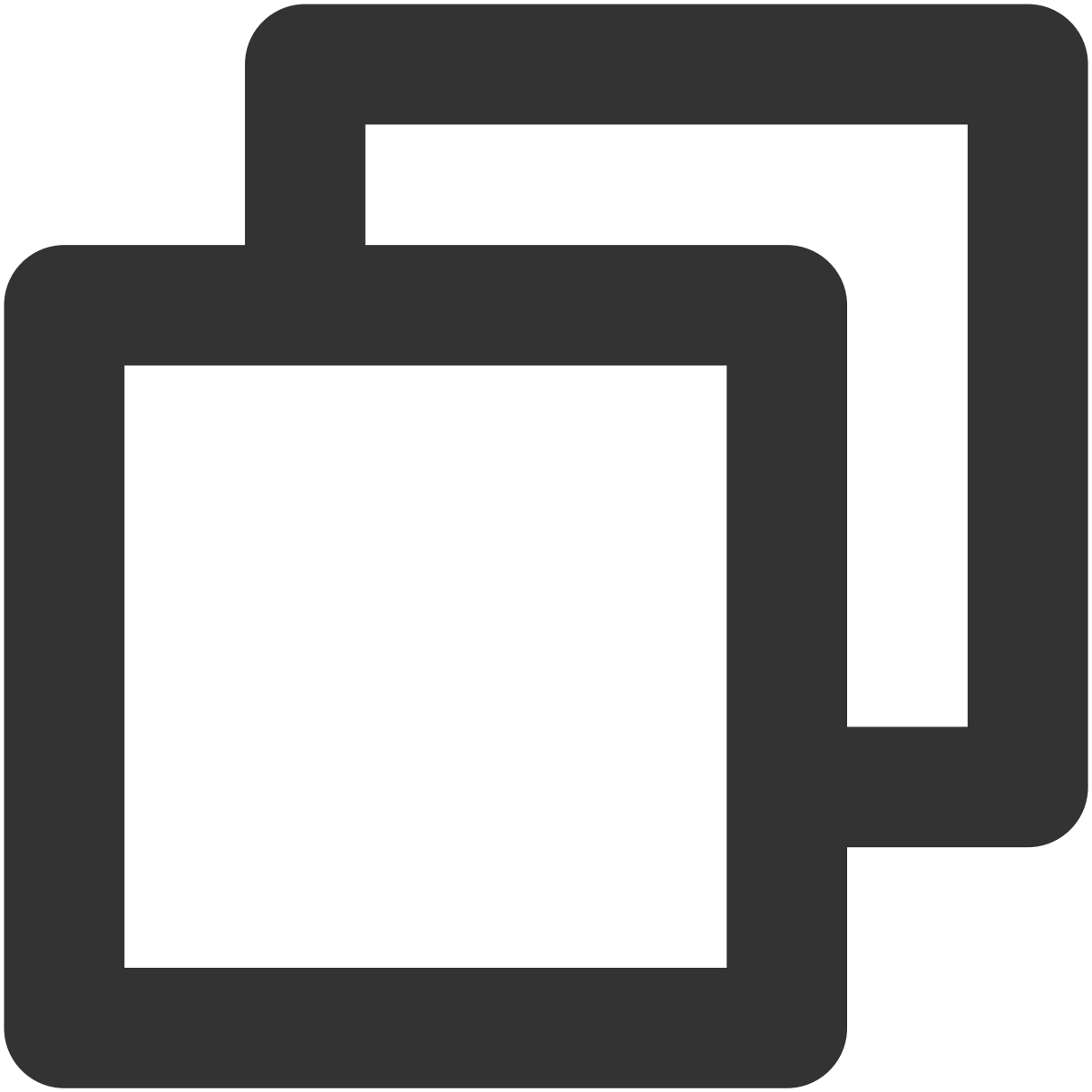
2. CoreDNSにReadinessProbeを追加設定する：



```
readinessProbe:
  failureThreshold: 5
  httpGet:
    path: /ready
    port: 8181
    scheme: HTTP
  initialDelaySeconds: 30
  periodSeconds: 10
  successThreshold: 1
  timeoutSeconds: 5
```

CoreDNSがUDPを使用してアップストリームDNSにアクセスする設定

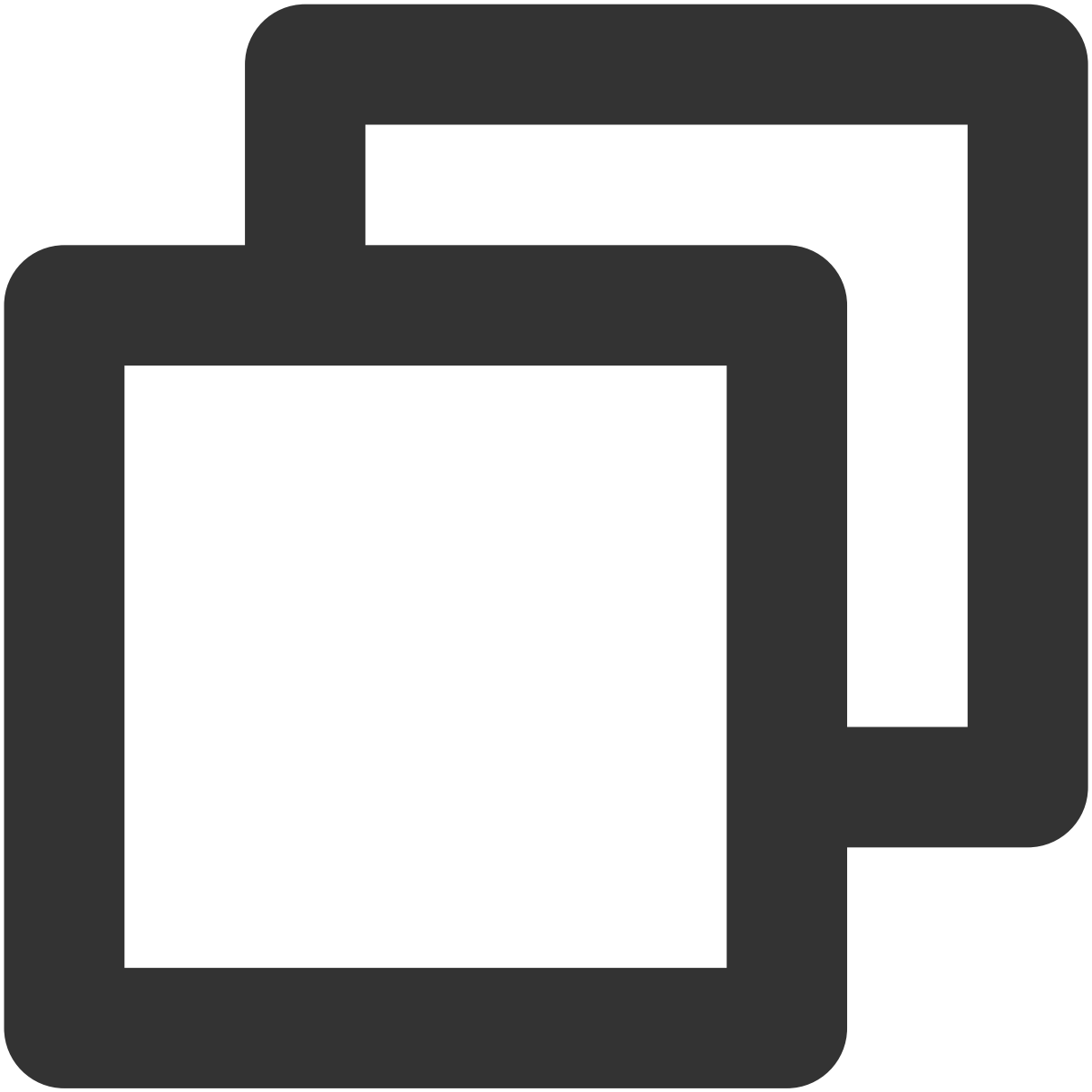
CoreDNSがアップストリームDNS Serverと通信する必要がある場合、デフォルトでクライアントがリクエストしたプロトコル（UDPまたはTCP）を使用します。TKE内のCoreDNSのアップストリームはデフォルトではVPC内のDNSサービスです。このサービスのTCPに対するサポートはパフォーマンスの制限があるため、以下のように設定し、指定UDPを表示することを推奨します（特にNodeLocal DNSCacheをインストールした場合）。



```
.:53 {
    forward . /etc/resolv.conf {
        prefer_udp
    }
}
```

CoreDNSを設定してINFOリクエストをフィルタリングする

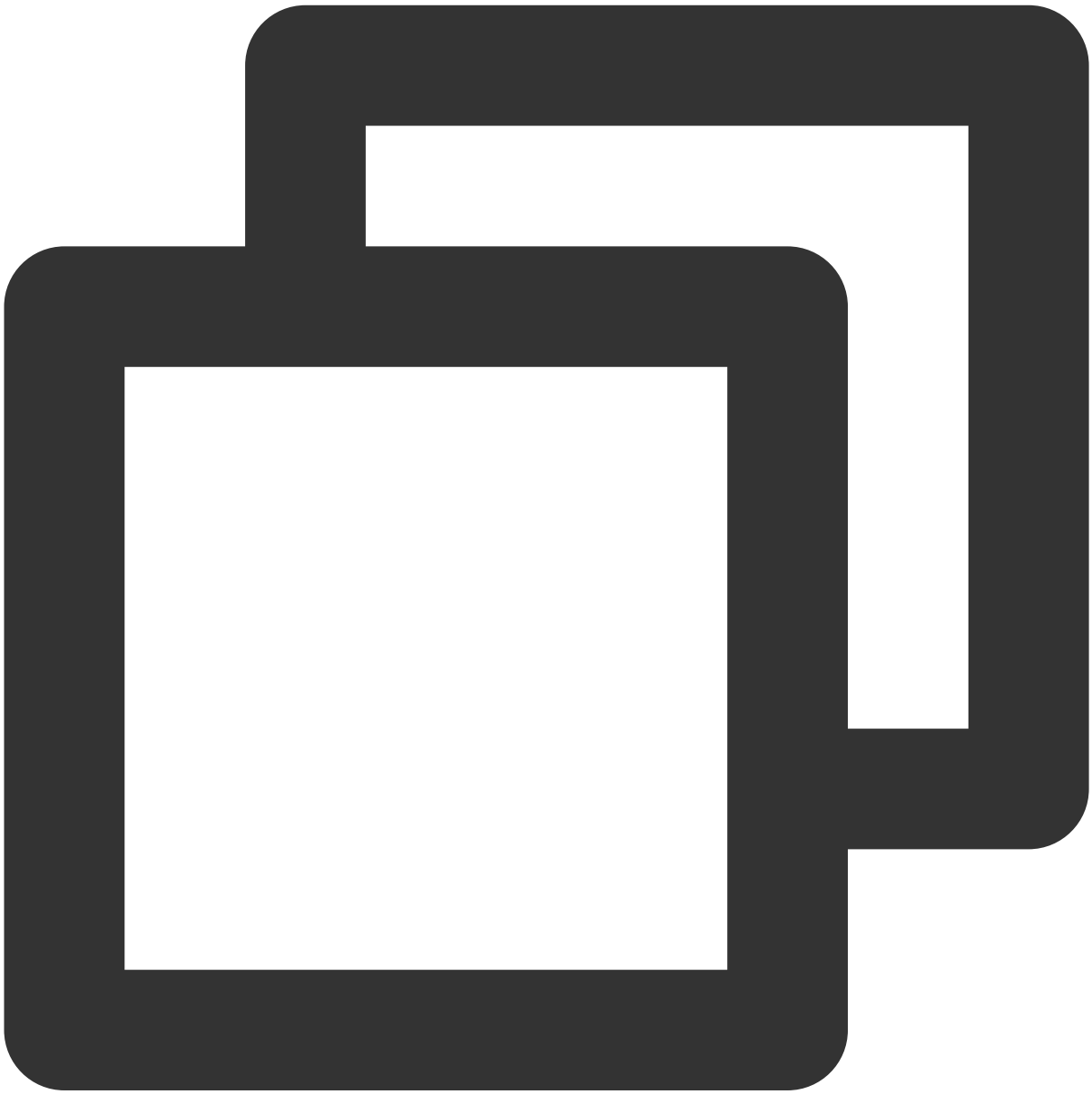
VPC内のDNSサービスはHINFOタイプのDNSリクエストをサポートしていないため、以下のように設定し、CoreDNS側でこのタイプのリクエストをフィルタリングすることを推奨します（特にNodeLocal DNSCacheをインストールした場合）。



```
.:53 {
    template ANY HINFO . {
        rcode NXDOMAIN
    }
}
```

CoreDNSを設定してIPv6タイプのAAAA記録に対してドメイン名が存在しないことが返ってくることを確認する

業務がIPv6のドメイン名解決をする必要がない場合、この設定によって通信コストを削減することができます。



```
.:53 {
    template ANY AAAA {
        rcode NXDOMAIN
    }
}
```

注意

IPv4/IPv6デュアルスタッククラスターはこの設定ができません。

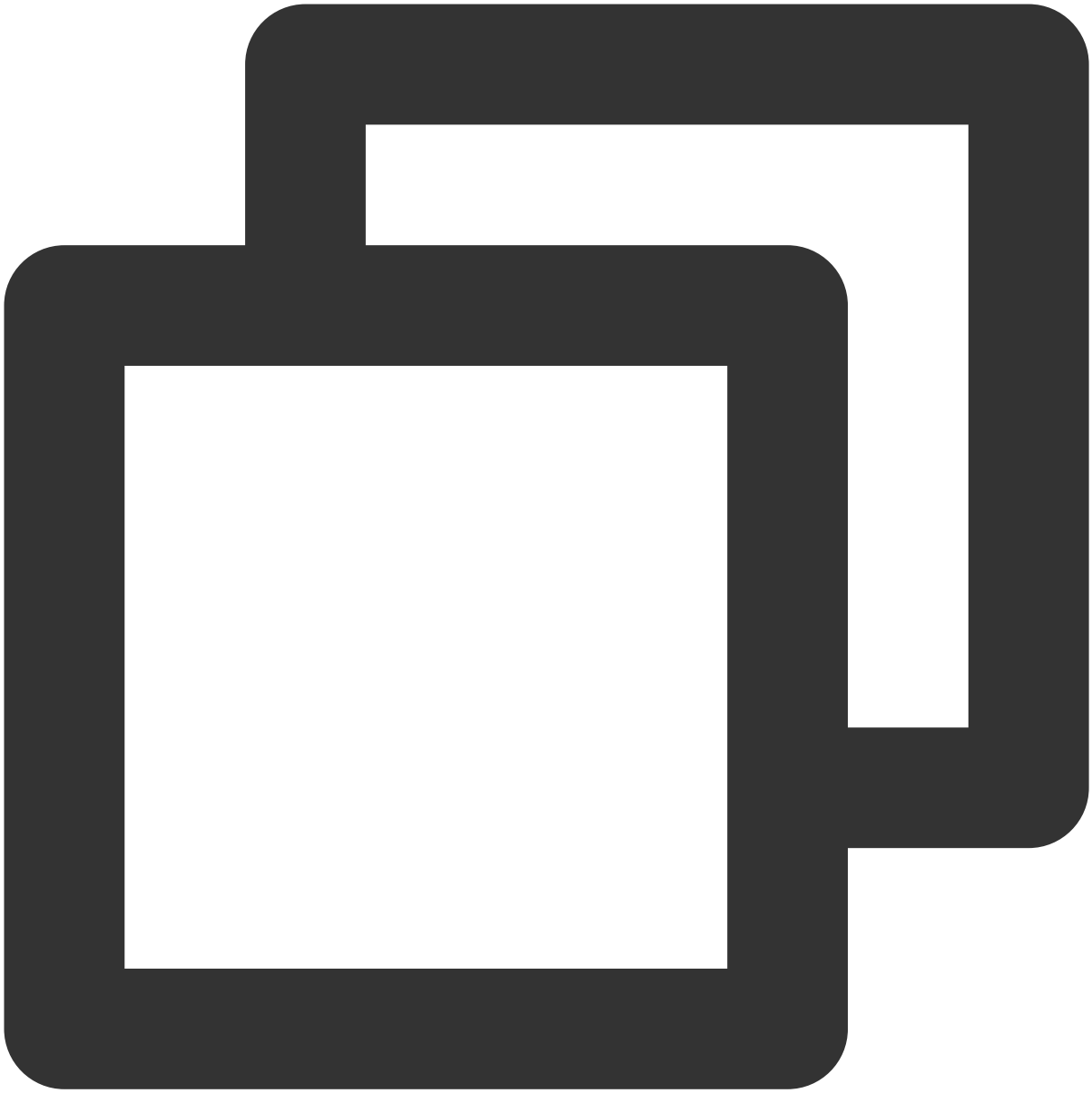
カスタムドメイン名解決の設定

詳細は以下をご参照ください。TKE内でカスタムドメイン名解決を実現する

手動アップグレード

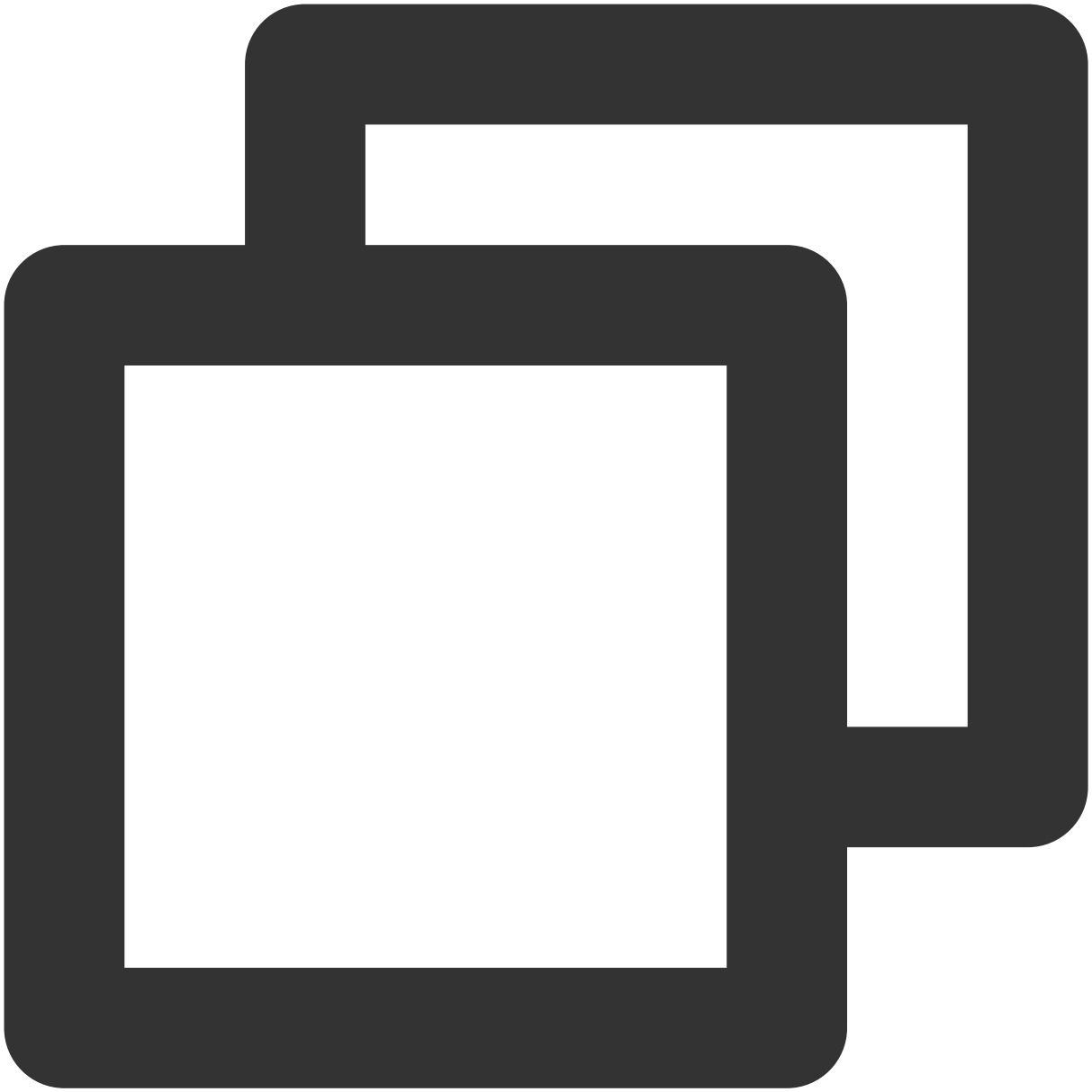
1.7.0にアップグレード

1. coredns configmapの編集



```
kubectl edit cm coredns -n kube-system
```

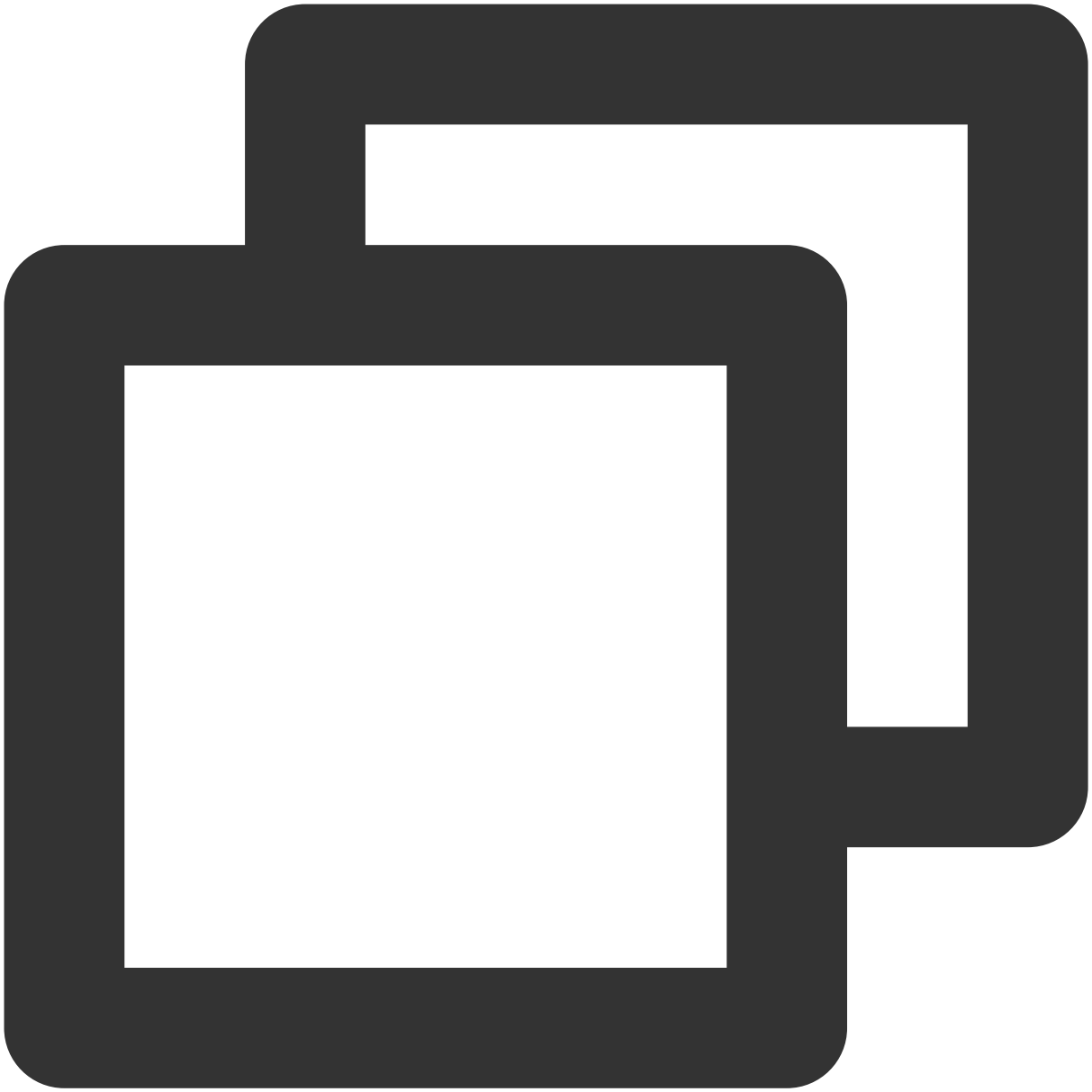
以下の内容に変更します。



```
.:53 {
    template ANY HINFO . {
        rcode NXDOMAIN
    }
    errors
    health {
        lameduck 30s
    }
    ready
    kubernetes cluster.local. in-addr.arpa ip6.arpa {
        pods insecure
```

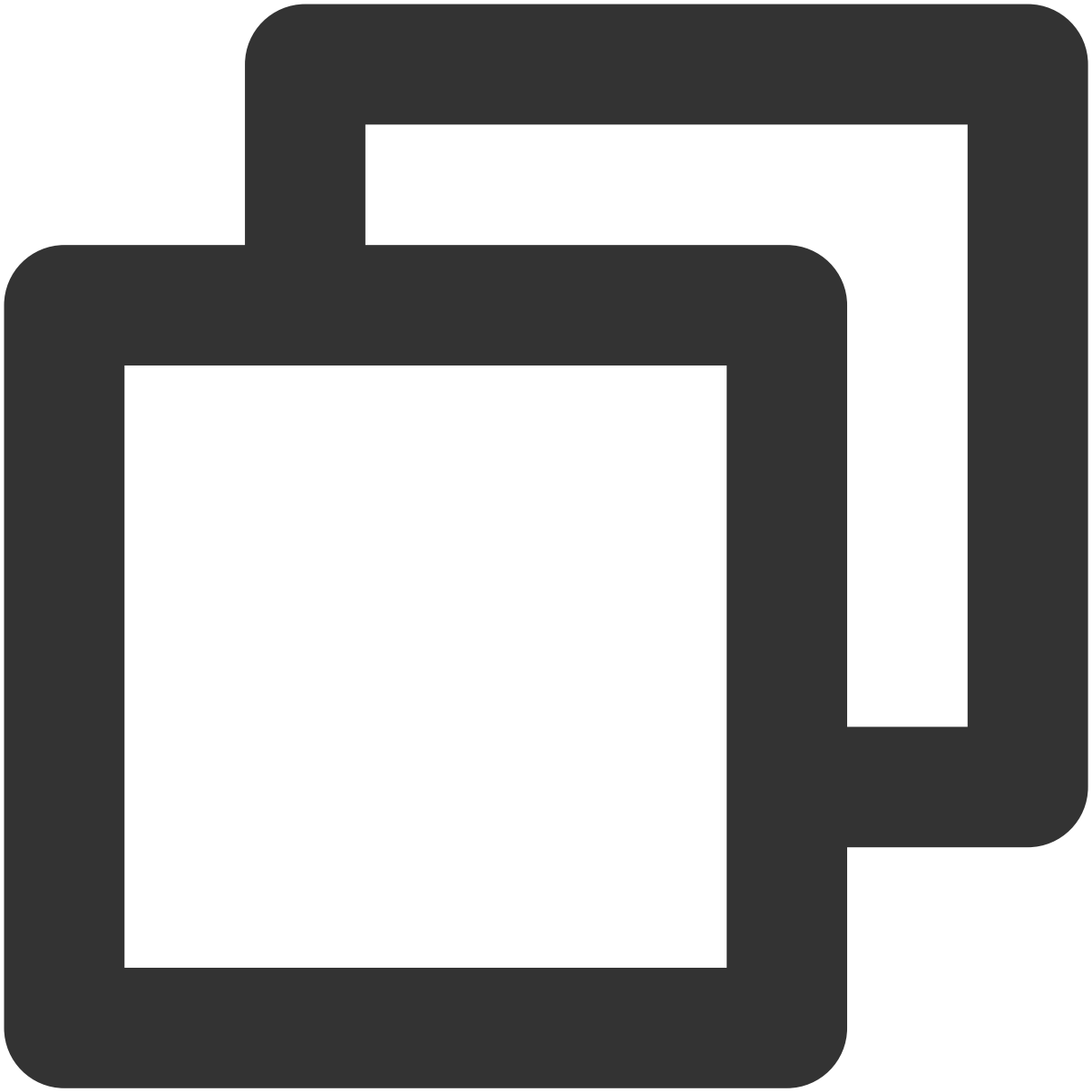
```
            fallthrough in-addr.arpa ip6.arpa
        }
        prometheus :9153
        forward . /etc/resolv.conf {
            prefer_udp
        }
        cache 30
        reload
        loadbalance
    }
```

2. coredns deploymentの編集



```
kubectl edit deployment coredns -n kube-system
```

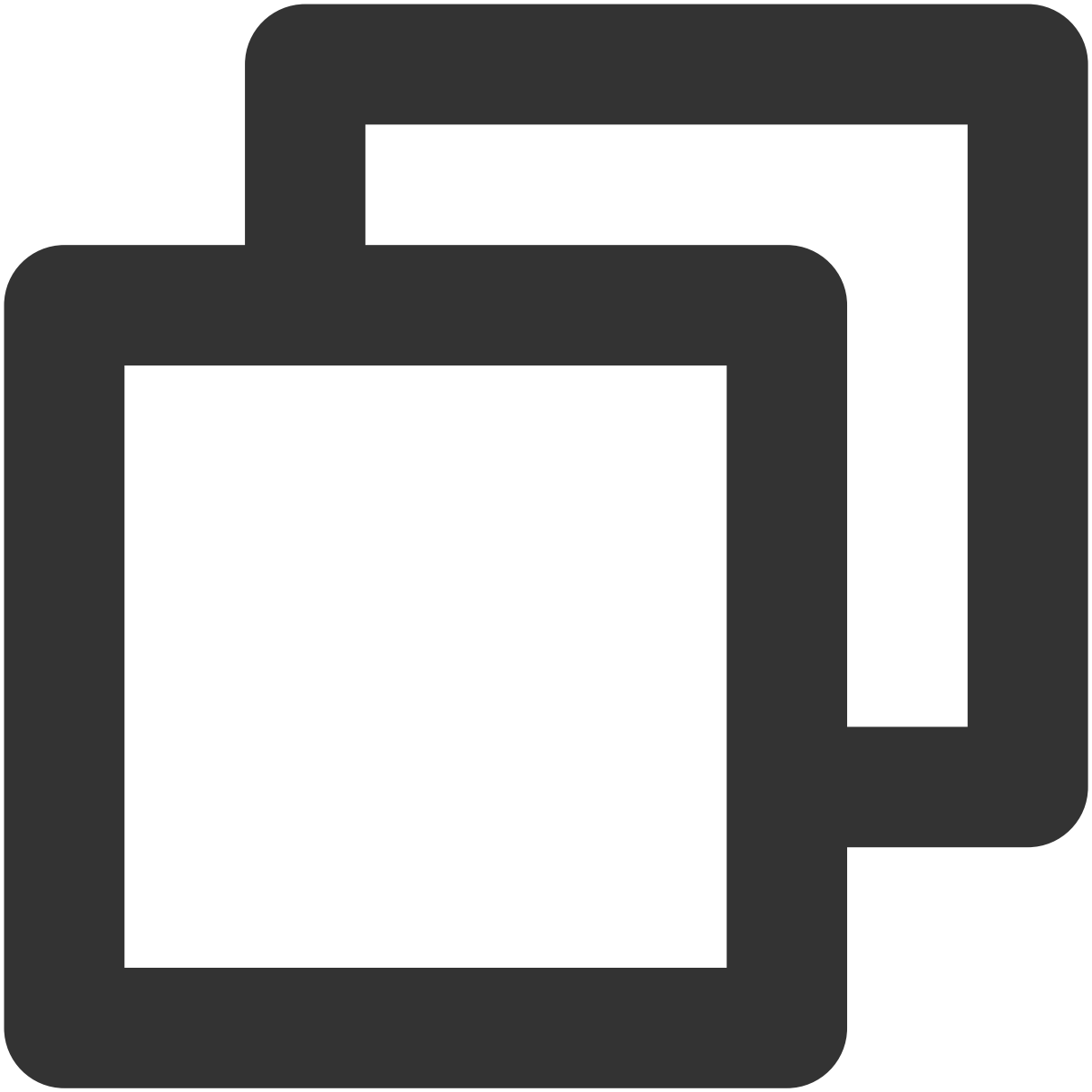
イメージを変換



```
image: ccr.ccs.tencentyun.com/tkeimages/coredns:1.7.0
```

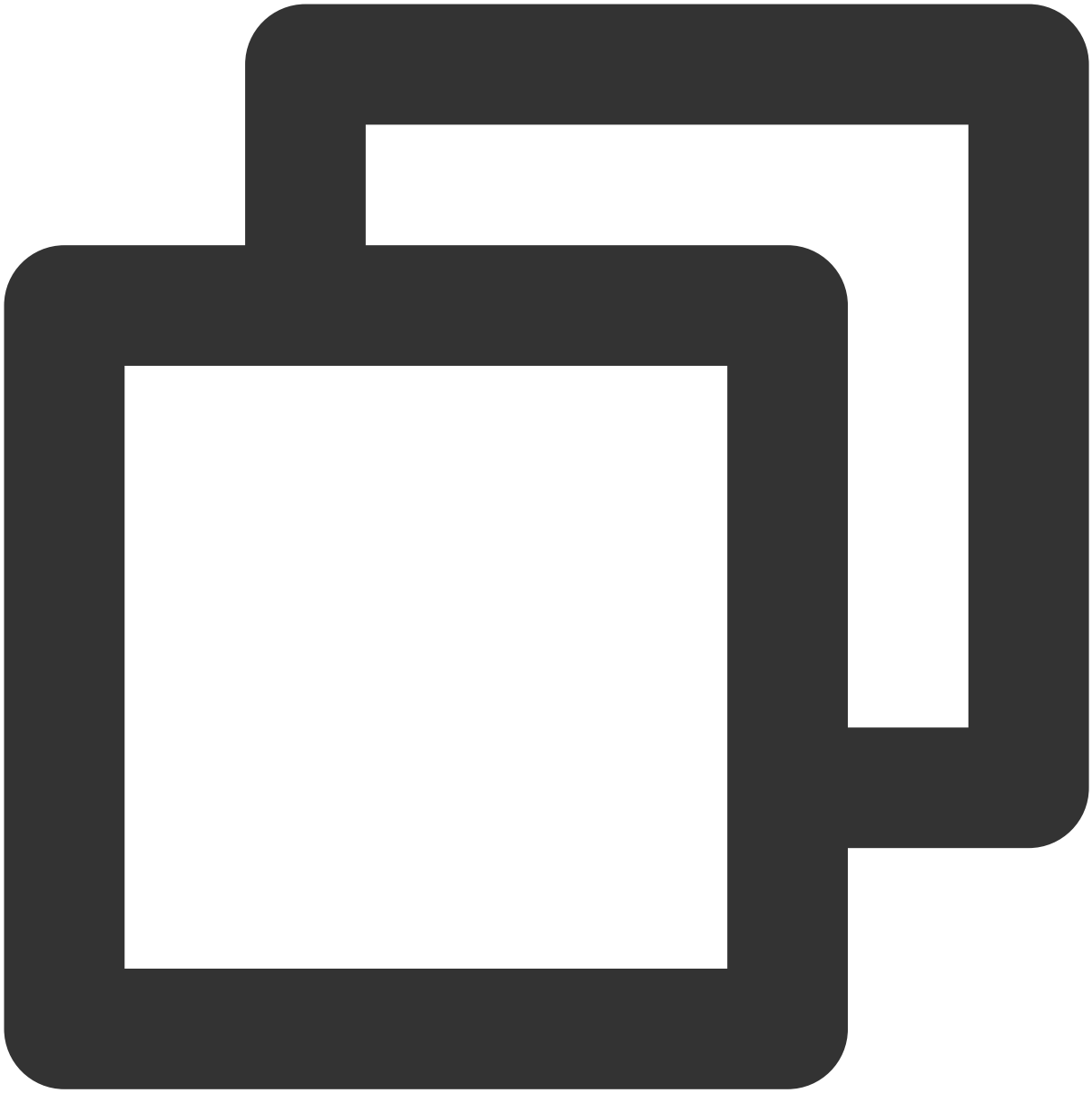
1.8.4にアップグレード

1. coredns clusterroleの編集



```
kubectl edit clusterrole system:coredns
```

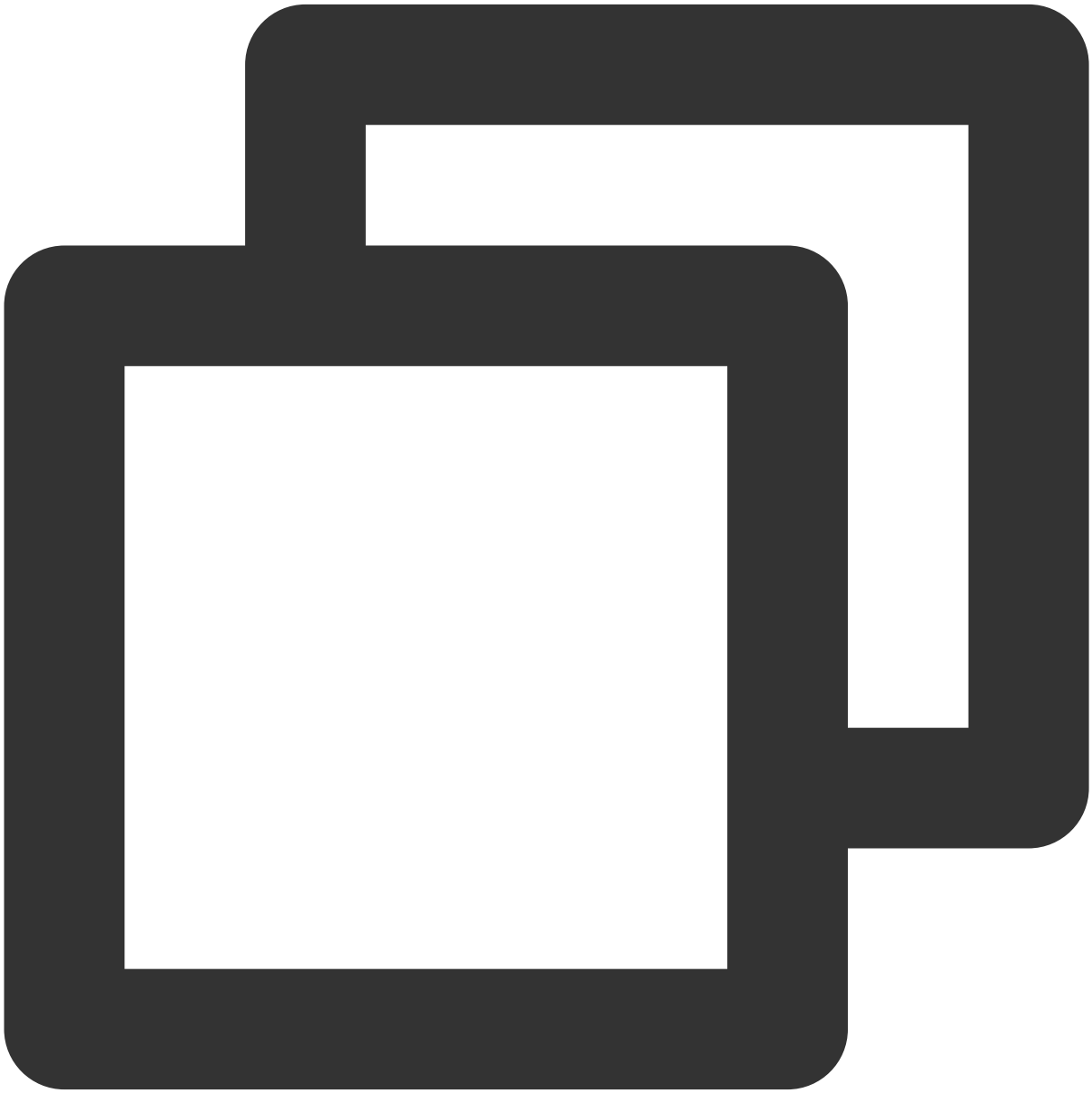
以下の内容に変更します。



```
rules:
- apiGroups:
  - '*'
  resources:
  - endpoints
  - services
  - pods
  - namespaces
  verbs:
  - list
  - watch
```

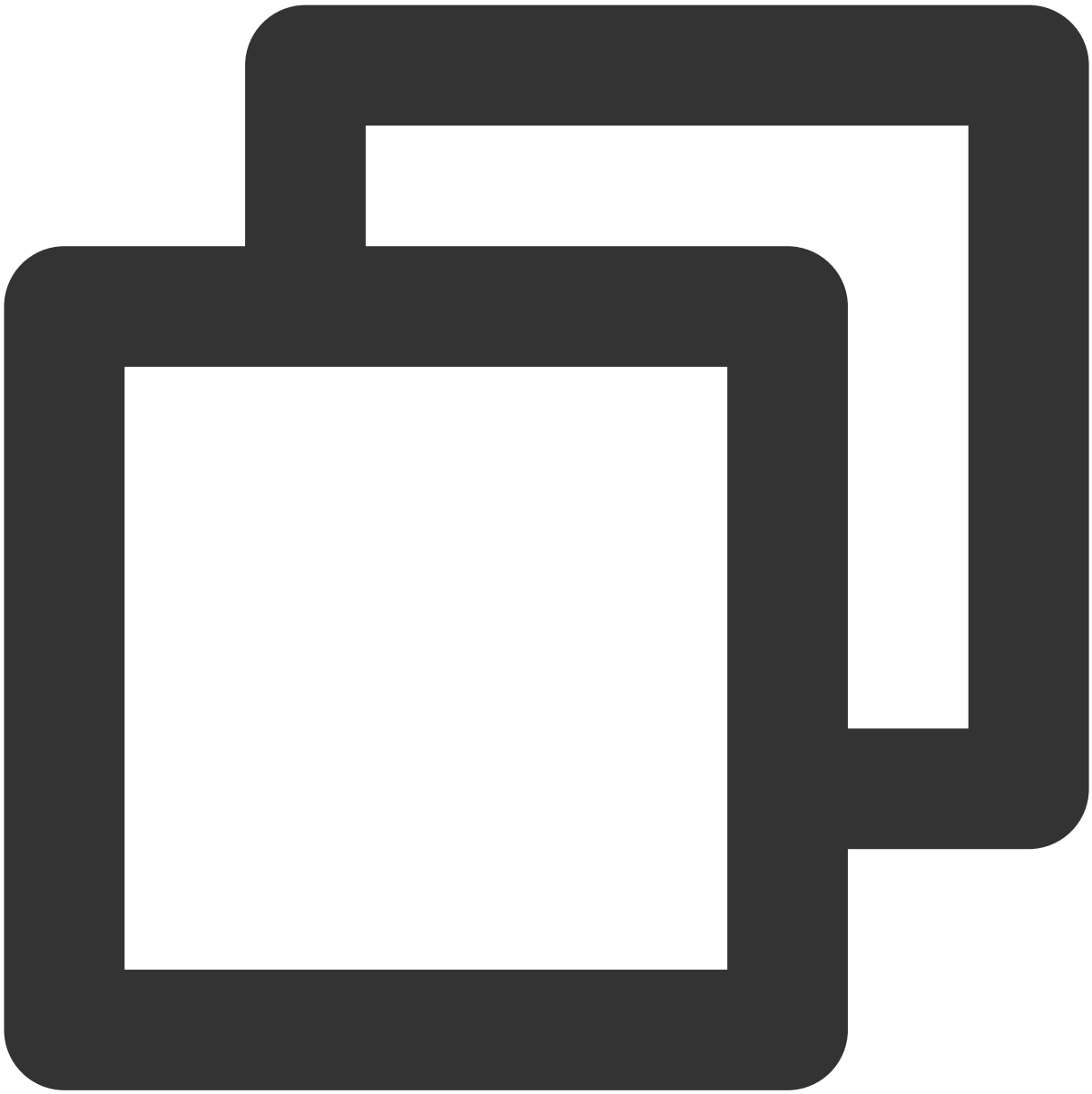
```
- apiGroups:
  - discovery.k8s.io
  resources:
  - endpointslices
  verbs:
  - list
  - watch
```

2. coredns configmapの編集



```
kubectl edit cm coredns -n kube-system
```

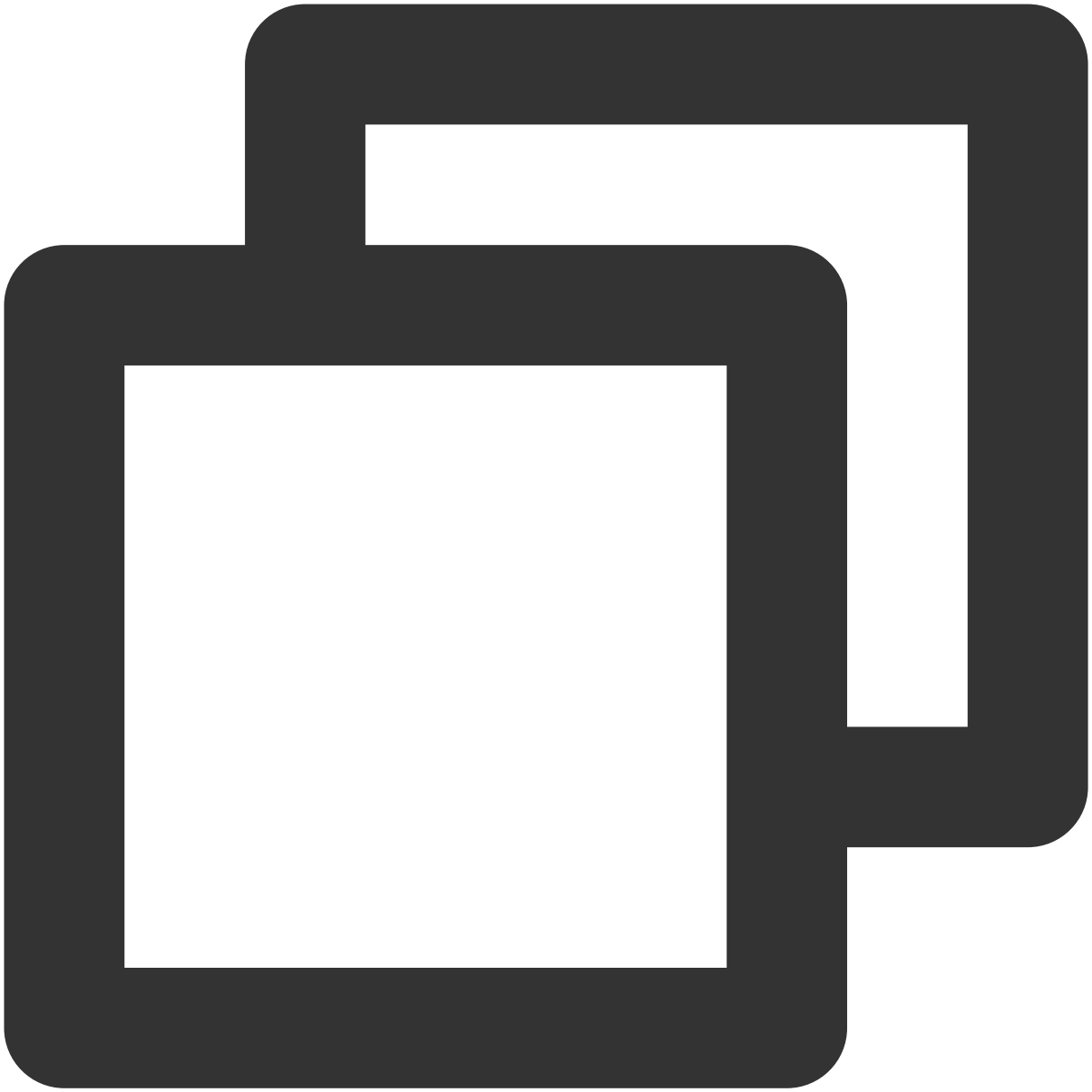
以下の内容に変更します。



```
.:53 {
    template ANY HINFO . {
        rcode NXDOMAIN
    }
    errors
    health {
        lameduck 30s
    }
    ready
```

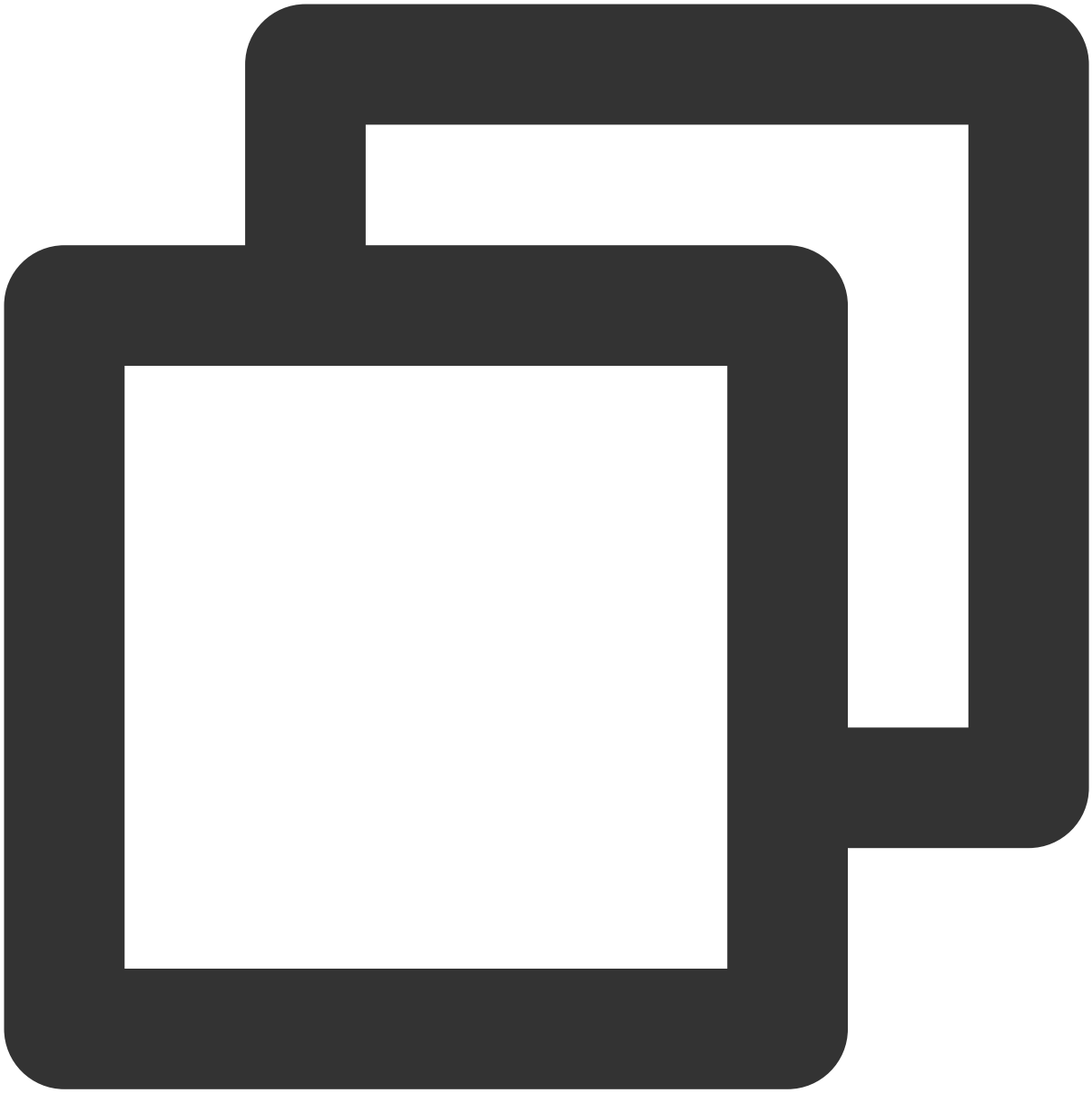
```
        kubernetes cluster.local. in-addr.arpa ip6.arpa {
            pods insecure
            fallthrough in-addr.arpa ip6.arpa
        }
        prometheus :9153
        forward . /etc/resolv.conf {
            prefer_udp
        }
        cache 30
        reload
        loadbalance
    }
```

3. coredns deploymentの編集



```
kubectl edit deployment coredns -n kube-system
```

イメージを変換

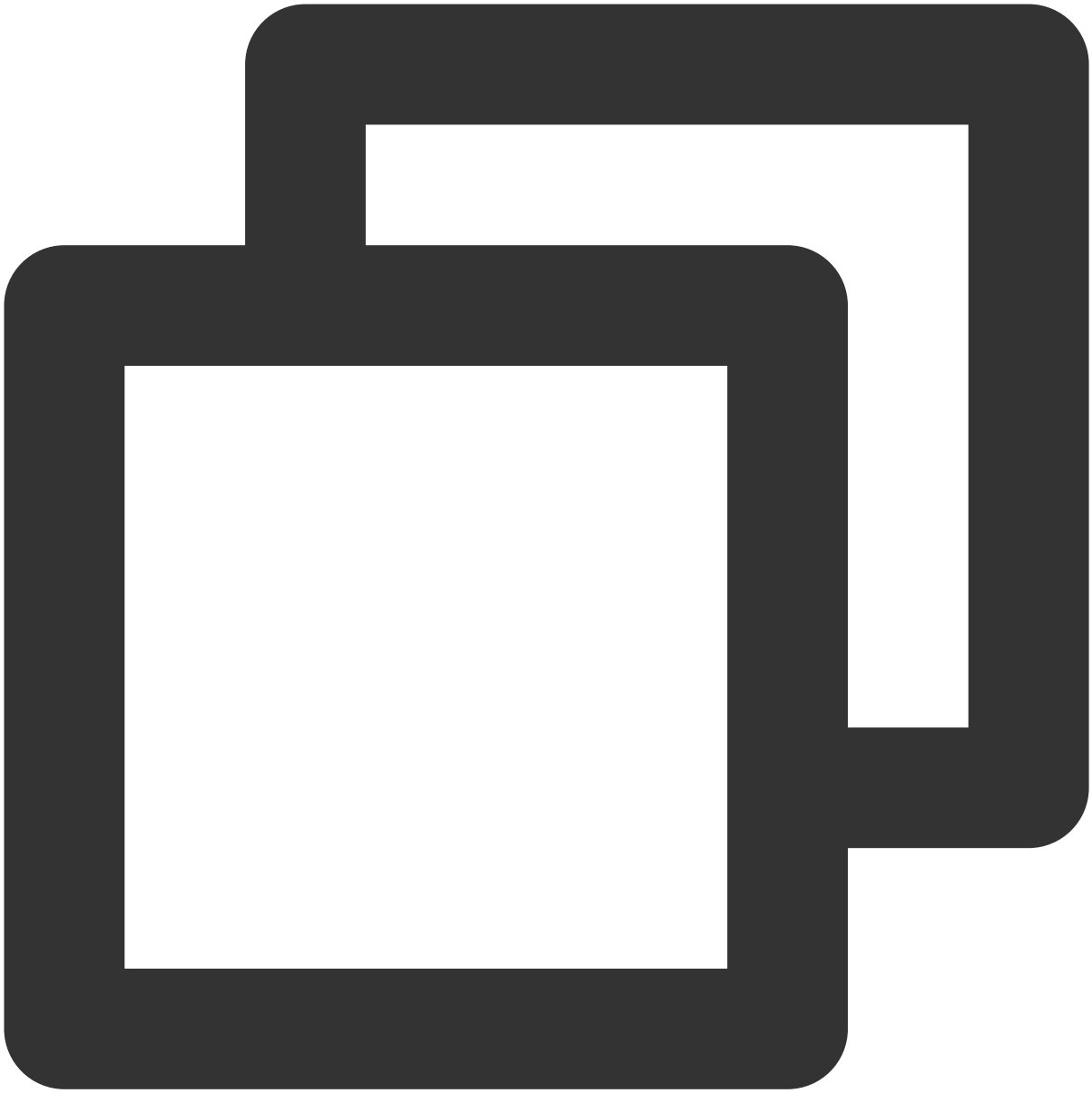


```
image: ccr.ccs.tencentyun.com/tkeimages/coredns:1.8.4
```

業務提案の設定

DNSサービスのベストプラクティス以外に、業務側で、適切な最適化設定をすることでも、DNSの使用体験を向上させることができます。

1. デフォルト状態では、Kubernetesクラスター内のドメイン名解決は多くの場合、複数回のリクエストによって解決する必要があります。pod内の `/etc/resolv.conf` を確認すると `ndots` オプションがデフォルトで5であることがわかります。例えば、debugネームスペースで `kubernetes.default.svc.cluster.local` というserviceを照会します。
ドメイン名に4つの `.` があり、5未満の場合、スプライシングを試みて最初のsearchの照会を行います。すなわち `kubernetes.default.svc.cluster.local.debug.svc.cluster.local` で、このドメイン名は見つかりません。
継続して `kubernetes.default.svc.cluster.local.svc.cluster.local` を試みても、このドメイン名は見つかりません。
継続して `kubernetes.default.svc.cluster.local.cluster.local` を試みても、依然としてこのドメイン名は見つかりません。
拡張子、すなわち `kubernetes.default.svc.cluster.local` を追加せずに試み、照会に成功すると、対応するClusterIPを返します。
2. 上記の簡単なserviceドメイン名解決は4回の解決に成功する必要があり、クラスター内には大量の不要なDNSリクエストに溢れています。そのため業務設定のアクセス方法に基づいてその合理的なndotsを設定することによって照会回数を低下させます。



```
spec:
  dnsConfig:
    options:
    - name: ndots
      value: "2"
  containers:
  - image: nginx
    imagePullPolicy: IfNotPresent
    name: diagnosis
```

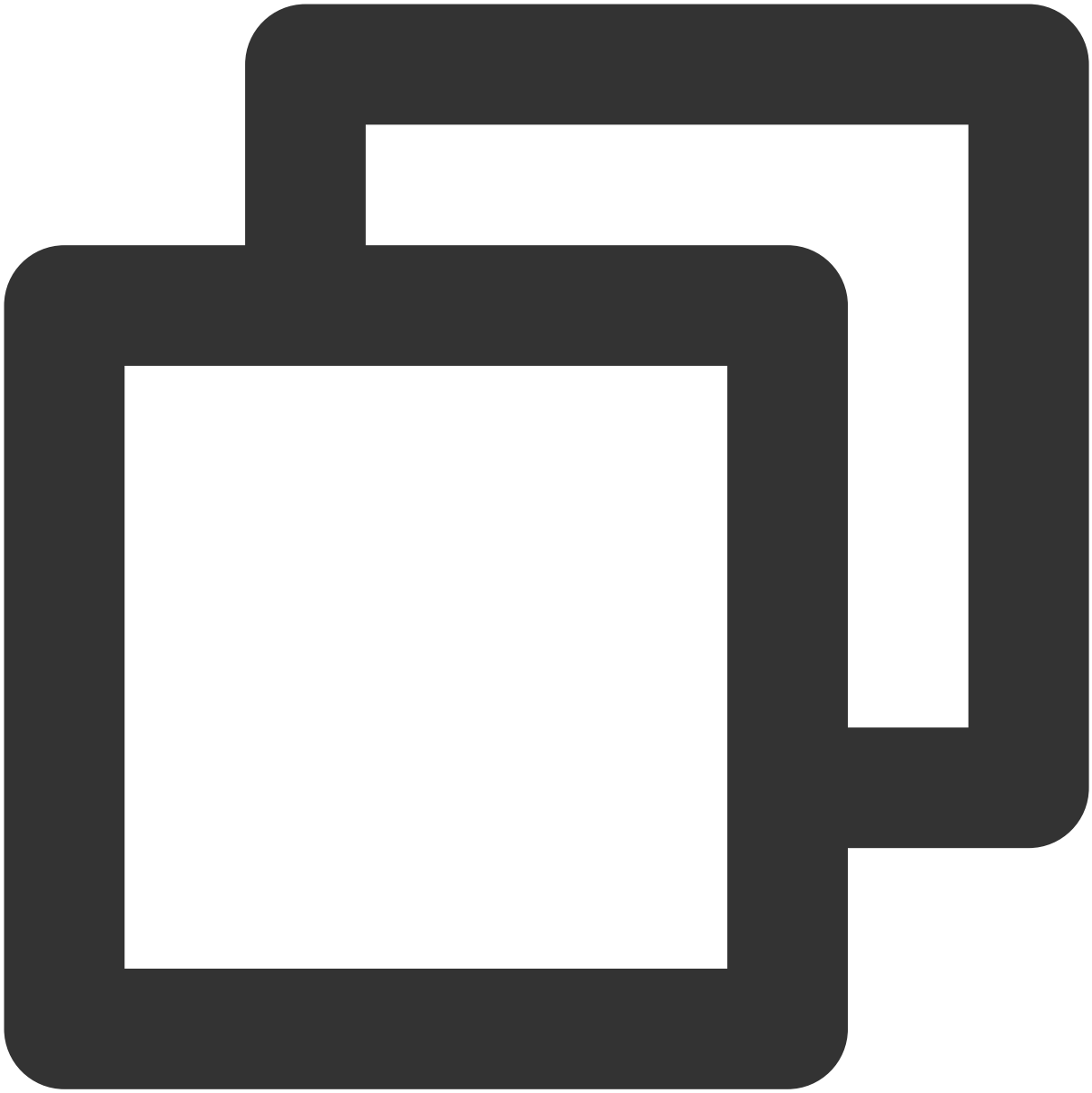
3. 同時に、業務アクセスサービスのドメイン名設定を最適化することができます。

Podはこのネームスペースの Serviceにアクセスし、 `<service-name>` を使用してアクセスします。
Podはその他のネームスペースの Serviceにアクセスし、 `<service-name>.<namespace-name>` を使用してアクセスします。
Podは外部ドメイン名にアクセスし、 FQDNタイプのドメイン名を使用してアクセスします。ドメイン名の最後に `.` を追加することによって無効な検索を減少させます。

関連する内容

設定紹介

errors
エラー情報を出力します。
health
健康状態を報告し、ヘルスチェックの設定に使用します。例えば、 `livenessProbe` は、デフォルトのリスナーは8080ポートで、パスは `http://localhost:8080/health` です
注意
複数のServerブロックがある場合、healthは1回だけ、または異なるポートで設定できます。



```
com {
  whoami
  health :8080
}

net {
  erratic
  health :8081
}
```

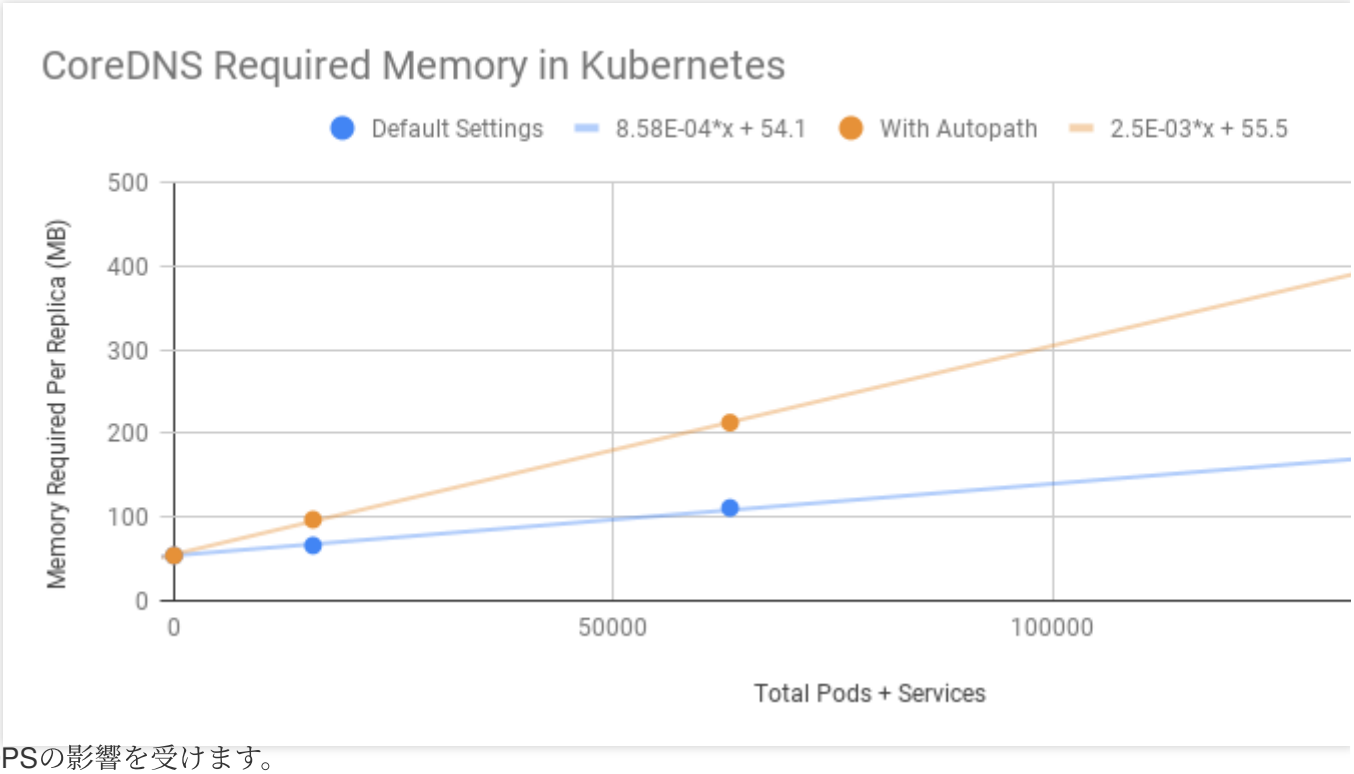
lameduck
グレースフルシャットダウンの時間を設定するために使用されます。実現方法はhookがCoreDNSで終了信号を受信した時に、その中でsleepを実行することによって、制限時間内に継続してサービスを提供することを保証します。
ready
プラグイン状態を報告し、サービス準備チェックの設定に使用します。例えば `readinessProbe` は、デフォルトのリスナーが8181ポートで、パスは `http://localhost:8181/ready` です
kubernetes
Kubernetesプラグインは、クラスター内のサービス解析をサポートしています。
prometheus
metricsデータインターフェースは、監視データの取得に使用されます。パスは `http://localhost:9153/metrics` です
forward(proxy)
処理できないリクエストをアップストリームDNSサーバーに転送します。デフォルトはホストの `/etc/resolv.conf` を使用して設定します。
forward aaa bbbの設定に基づいて、内部にudnsのリスト[aaa,bbb]が保持されます
リクエストが届いた時、プリセットポリシー（random|round_robin|sequential、デフォルトではrandom）に基づき、リスト [aaa,bbb]内でudnsが送信するリクエストを見つけます。失敗した場合、次のudnsで試し、同時に失敗したudnsに対して定期的なヘルスチェックを開始し、正常になるまで、ヘルスチェックを行います。
ヘルスチェックのプロセスで、連続数回（デフォルトでは2回）監視に失敗した場合、そのudnsステータスをdownに設定します。その後リストからudnsを選択する時にステータスがdownのudnsをスキップします。
すべてのudnsがdownの場合、ランダムに1つのudnsを選択して転送します。
このため、corednsには複数のupstream間のインテリジェント切り替えの機能があると考えることができ、forwardリスト内に利用可能なudnsが1つあれば、リクエストは成功します。
cache
DNSキャッシュです。
reload
Corefileをホットリロードし、ConfigMapを変更した後、2分以内に新しい設定をロードします。
loadbalance
DNSに基づくCLB機能を提供し、ランダムにレコードの順序に応答します。

CoreDNSリソース占有

メモリ
CPU
主にクラスター内のPod数およびService数によって決まります。
開いているキャッシュサイズの影響を受けます。
QPSの影響を受けます。

以下のデータはCoreDNS公式からのものです。

MB required (default settings) = (Pods + Services) / 1000 + 54



主にQPSの影響を受けます。

以下のデータはCoreDNS公式からのものです。

単一レプリカのCoreDNS、実行ノード仕様：2 vCPUs、7.5 GB memory

Query Type	QPS	Avg Latency (ms)	Memory Delta (MB)
external	6733	12.02	+5
internal	33669	2.608	+5

TKEでのカスタムドメイン名解決の実現

最終更新日：：2023-04-26 19:23:11

ユースケース

Tencent Kubernetes Engine（TKE）またはTKE Serverlessを使用する際、内部のカスタムドメイン名解決の必要性がある場合があります。
クラスター外に集中型ストレージサービスをご自身で構築した場合は、クラスター内のモニタリングまたはログデータを収集し、固定の内部ドメイン名によってストレージサービスに送信する必要があります。
従来の業務をコンテナ化する過程で、一部のサービスのコードが固定ドメイン名を使用して内部の他のサービスを呼び出すように設定されていて、かつその設定が変更できない場合は、KubernetesのService名を使用して呼び出すことはできません。

方法の選択

ここではクラスターでカスタムドメイン名を使用して解決を行う、次の3種類の方法の例をご紹介します。

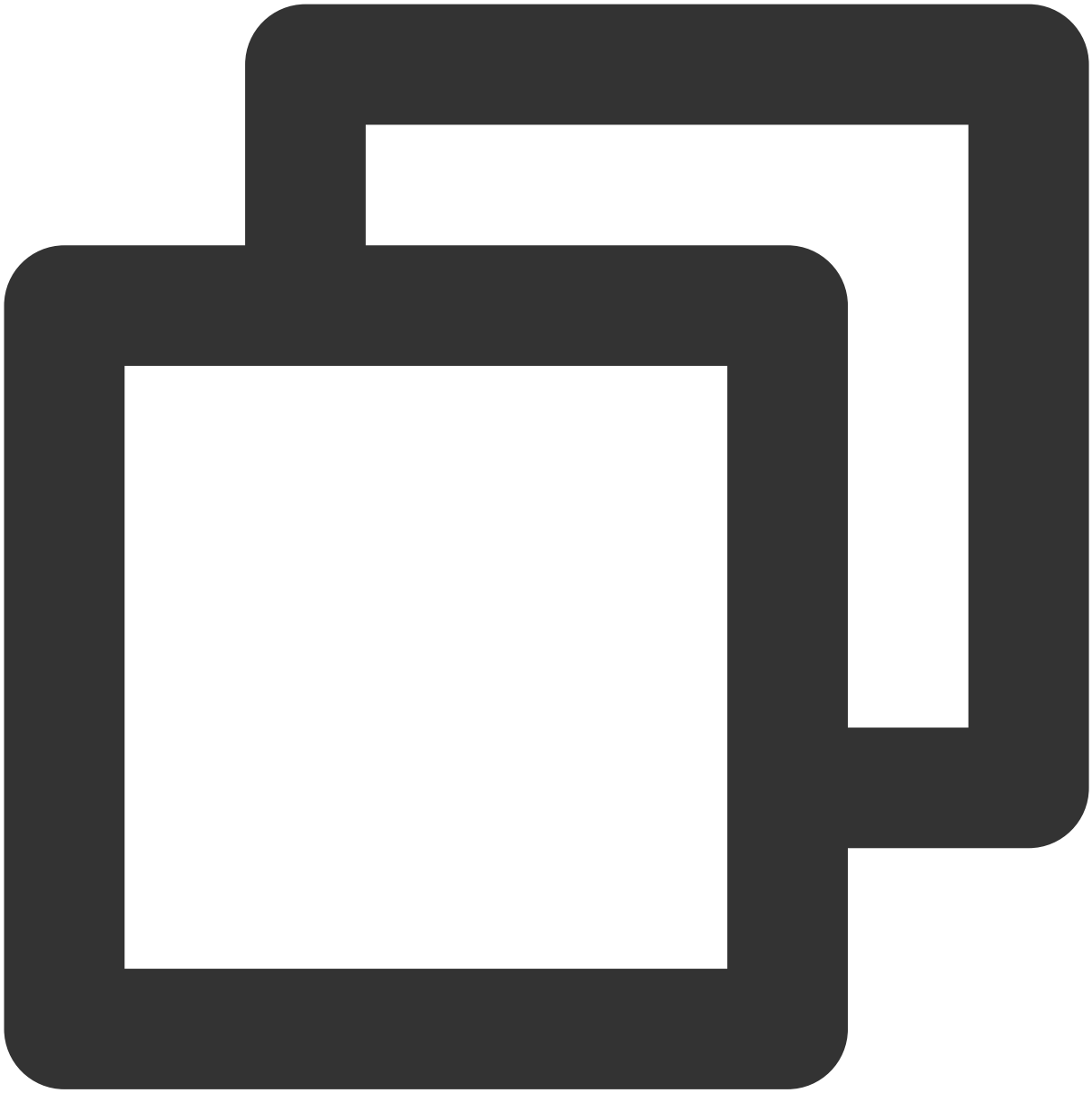
ション	メリット
方法1：CoreDNS Hostsプラグインを使用して任意のドメイン名解決を設定する	シンプルかつ直観的であり、任意の解決レコードを追加できます。
方法2：CoreDNS Rewriteプラグインを使用してドメイン名をクラスター内サービスに指定する	解決レコードのIPアドレスを事前に知る必要はありませんが、解決レコードが指定するアドレスはクラスター内にデプロイされている必要があります。
方法3：CoreDNS Forwardプラグインを使用して自作DNSをアップストリームDNSとして設定する	大量の解決レコードを管理でき、レコードの管理はすべて自作DNSで行えるため、レコードの追加と削除の際にCoreDNS設定を変更する必要がありません。

説明
方法1と方法2では、解決レコードを追加する際に毎回CoreDNSプロファイルを変更する必要があります（再起動は不要です）。ご自身のニーズに応じて評価し、具体的な方法を選択してください。

方法の例

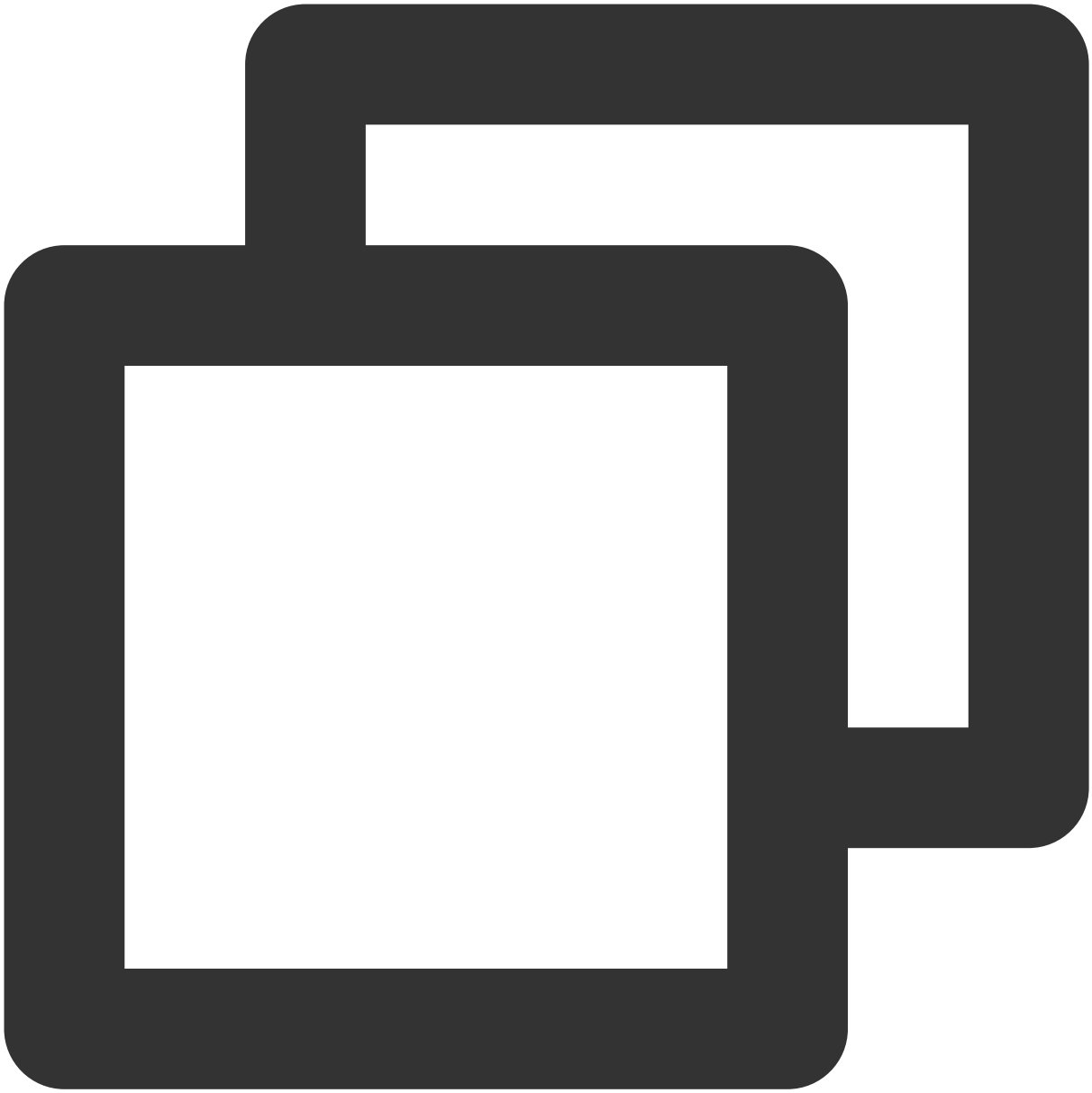
方法1：CoreDNS Hostsプラグインを使用して任意のドメイン名解決を設定する

1. 次のコマンドを実行し、CoreDNSのconfigmapを変更します。次に例を示します。



```
kubectl edit configmap coredns -n kube-system
```

2. hosts設定を変更し、ドメイン名をhostsに追加します。次に例を示します。

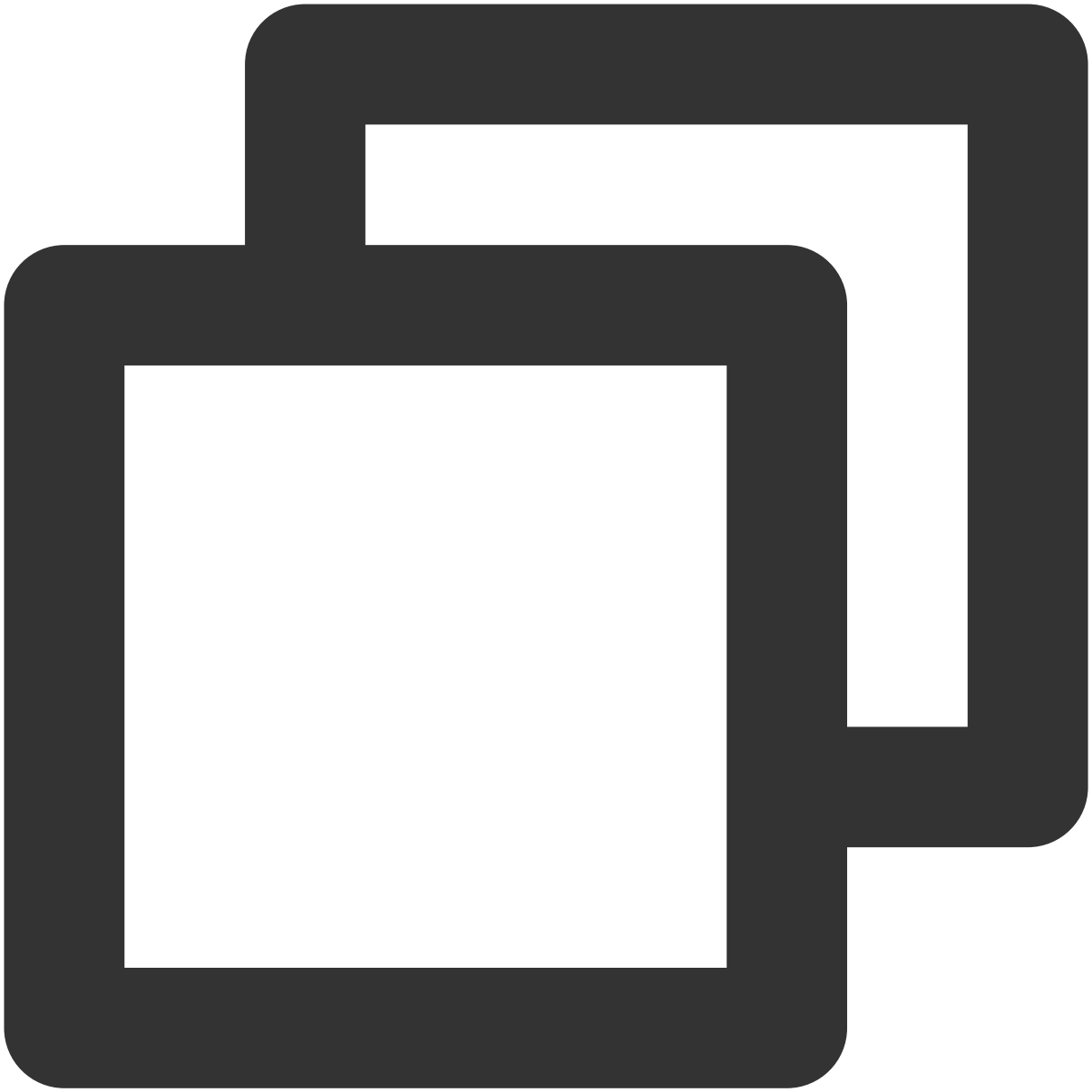


```
hosts {
      192.168.1.6     harbor.example.com
      192.168.1.8     es.example.com
      fallthrough
}
```

説明

`harbor.example.com` は192.168.1.6を指定し、 `es.example.com` は192.168.1.8を指定します。

完全な設定の例は次のとおりです。



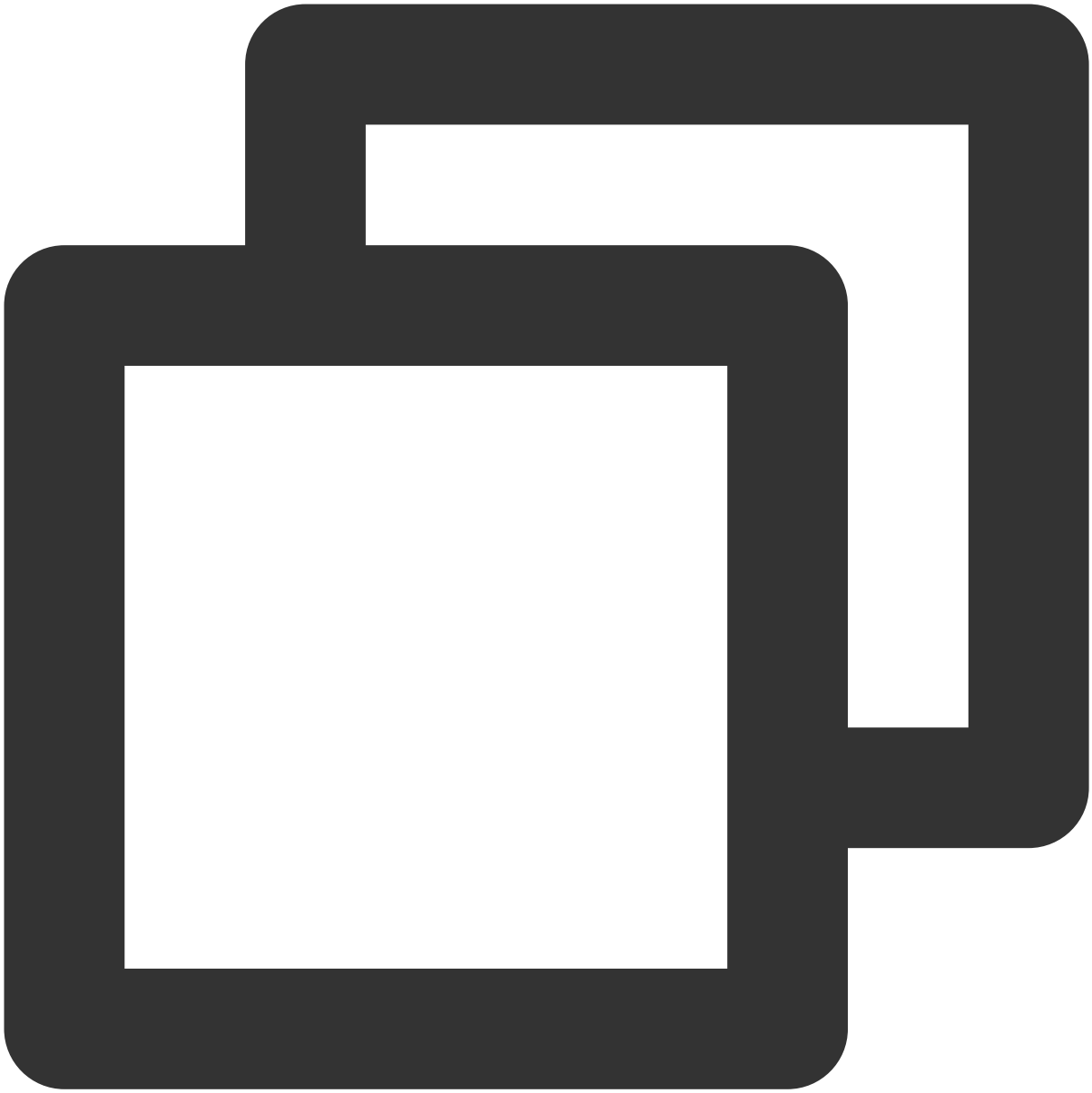
```
apiVersion: v1
data:
      Corefile: |2-
        .:53 {
            errors
            health
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                upstream
                fallthrough in-addr.arpa ip6.arpa
            }
```

```
            hosts {
                192.168.1.6     harbor.example.com
                192.168.1.8     es.example.com
                fallthrough
            }
            prometheus :9153
            forward . /etc/resolv.conf
            cache 30
            reload
            loadbalance
        }
kind: ConfigMap
metadata:
      labels:
        addonmanager.kubernetes.io/mode: EnsureExists
      name: coredns
      namespace: kube-system
```

方法2：CoreDNS Rewriteプラグインを使用してドメイン名をクラスター内サービスに指定する

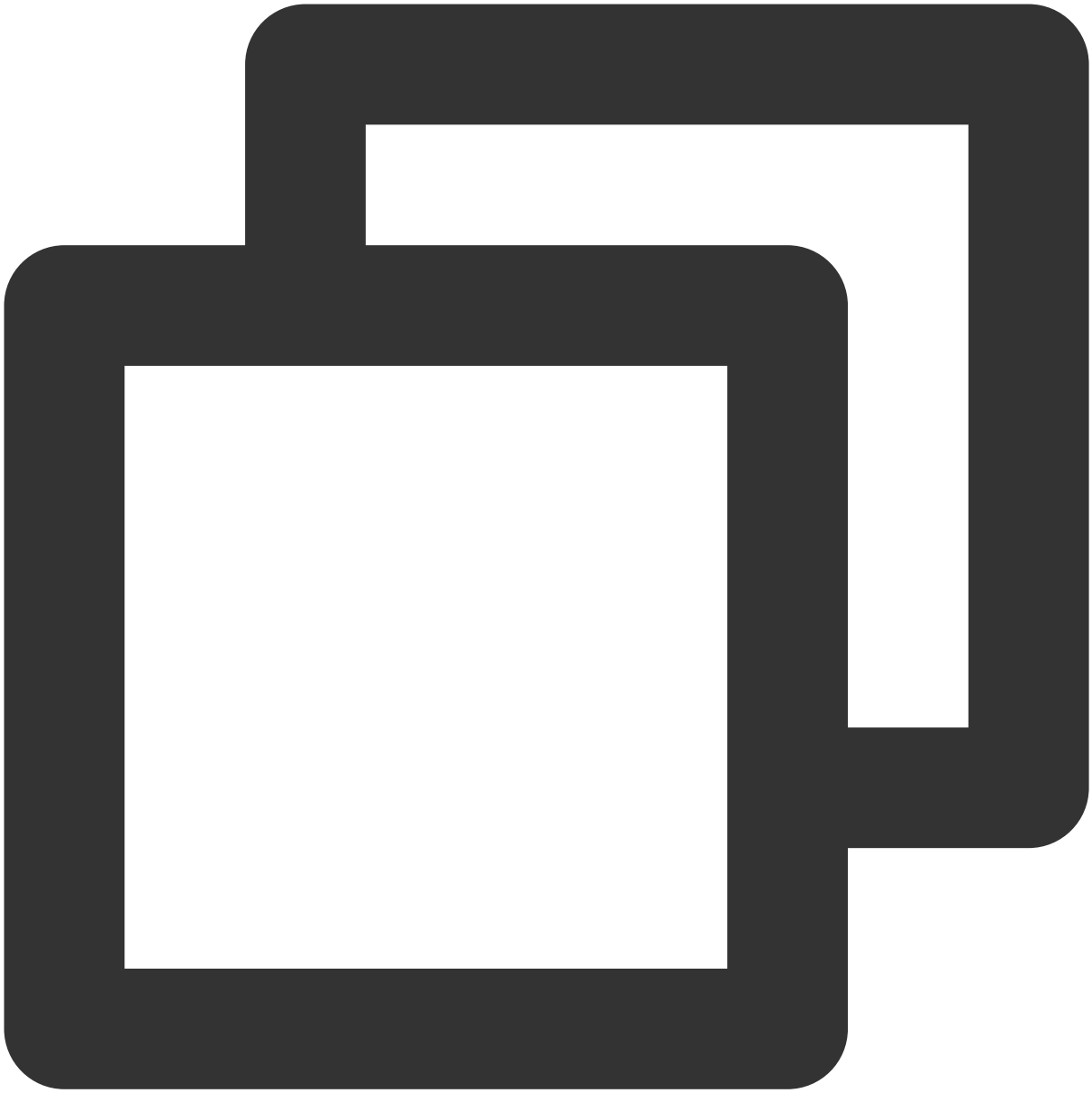
カスタムドメイン名を使用したサービスをクラスターにデプロイしたい場合は、CoreDNSのRewriteプラグインを使用し、指定のドメイン名をあるServiceのClusterIPに解決することができます。

1. 次のコマンドを実行し、CoreDNSのconfigmapを変更します。次に例を示します。



```
kubectl edit configmap coredns -n kube-system
```

2. 次のコマンドを実行し、Rewrite設定を追加します。次に例を示します。

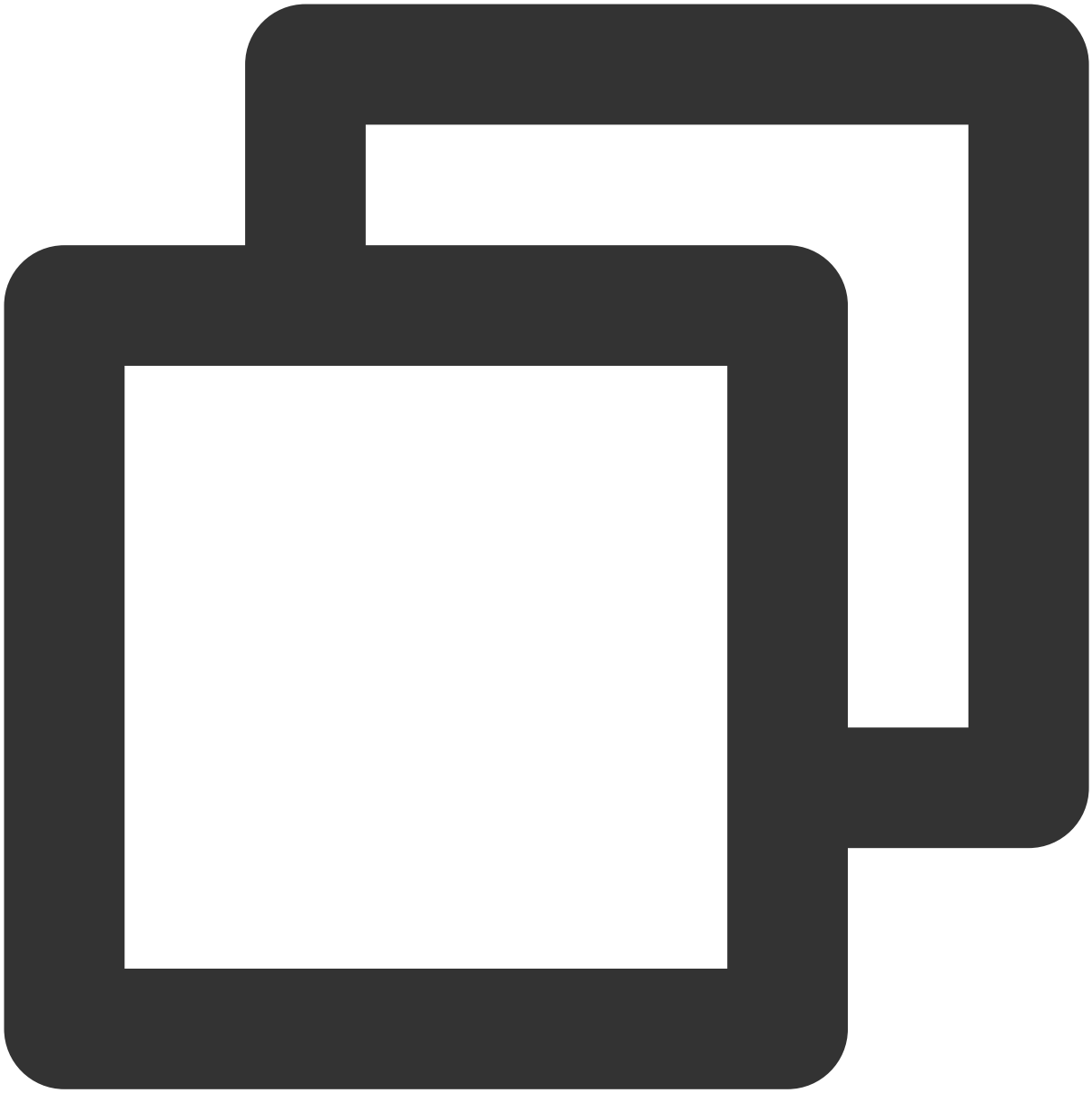


```
rewrite name es.example.com es.logging.svc.cluster.local
```

説明

`es.example.com` は `logging` ネームスペース下にデプロイされた `es` サービスを指定します。複数のドメイン名がある場合は行を追加できます。

完全な設定の例は次のとおりです。

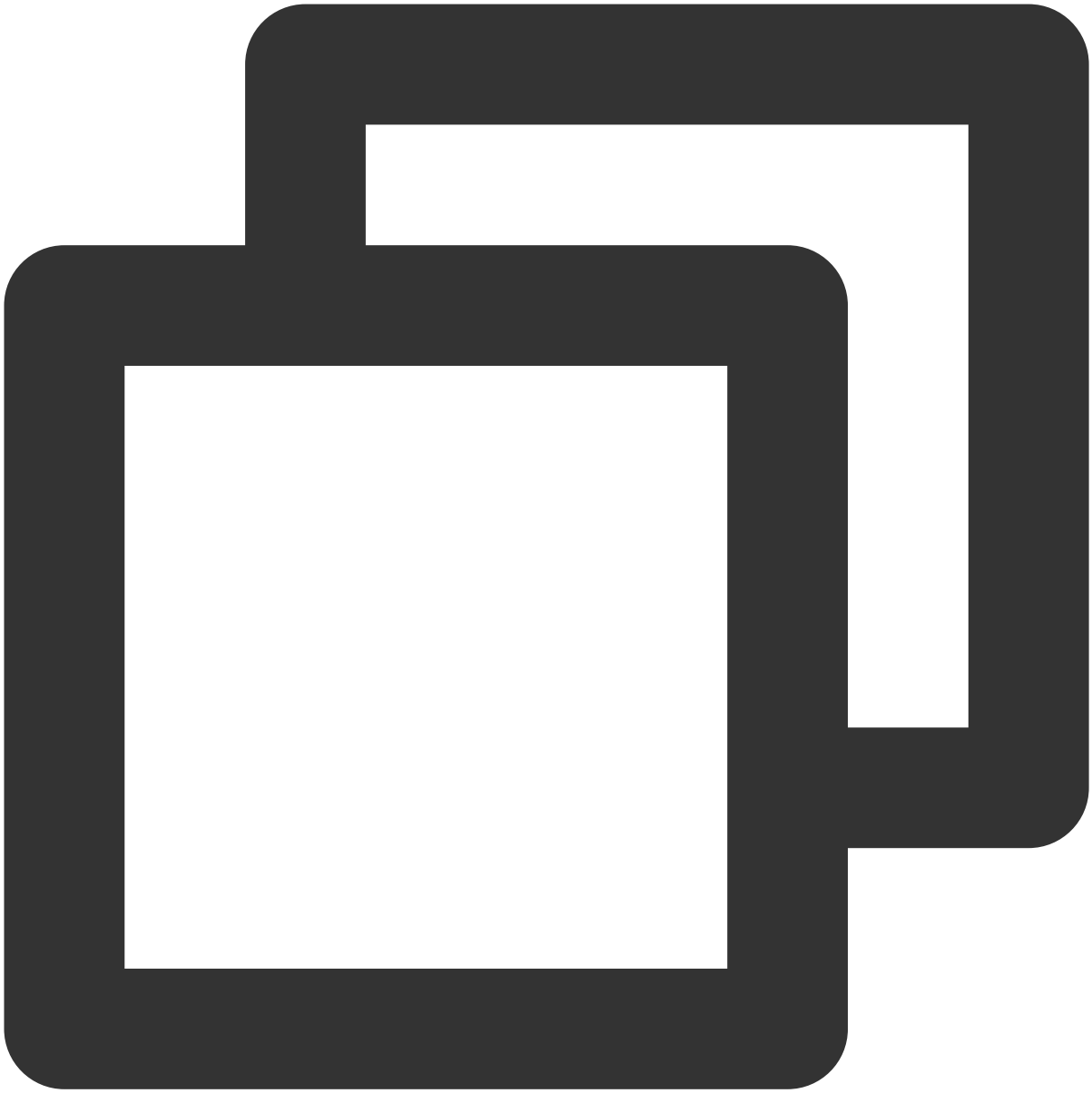


```
apiVersion: v1
data:
      Corefile: |2-
        .:53 {
            errors
            health
            kubernetes cluster.local. in-addr.arpa ip6.arpa {
                pods insecure
                upstream
                fallthrough in-addr.arpa ip6.arpa
            }
```

```
            rewrite name es.example.com es.logging.svc.cluster.local
            prometheus :9153
            forward . /etc/resolv.conf
            cache 30
            reload
            loadbalance
        }
kind: ConfigMap
metadata:
      labels:
        addonmanager.kubernetes.io/mode: EnsureExists
      name: coredns
      namespace: kube-system
```

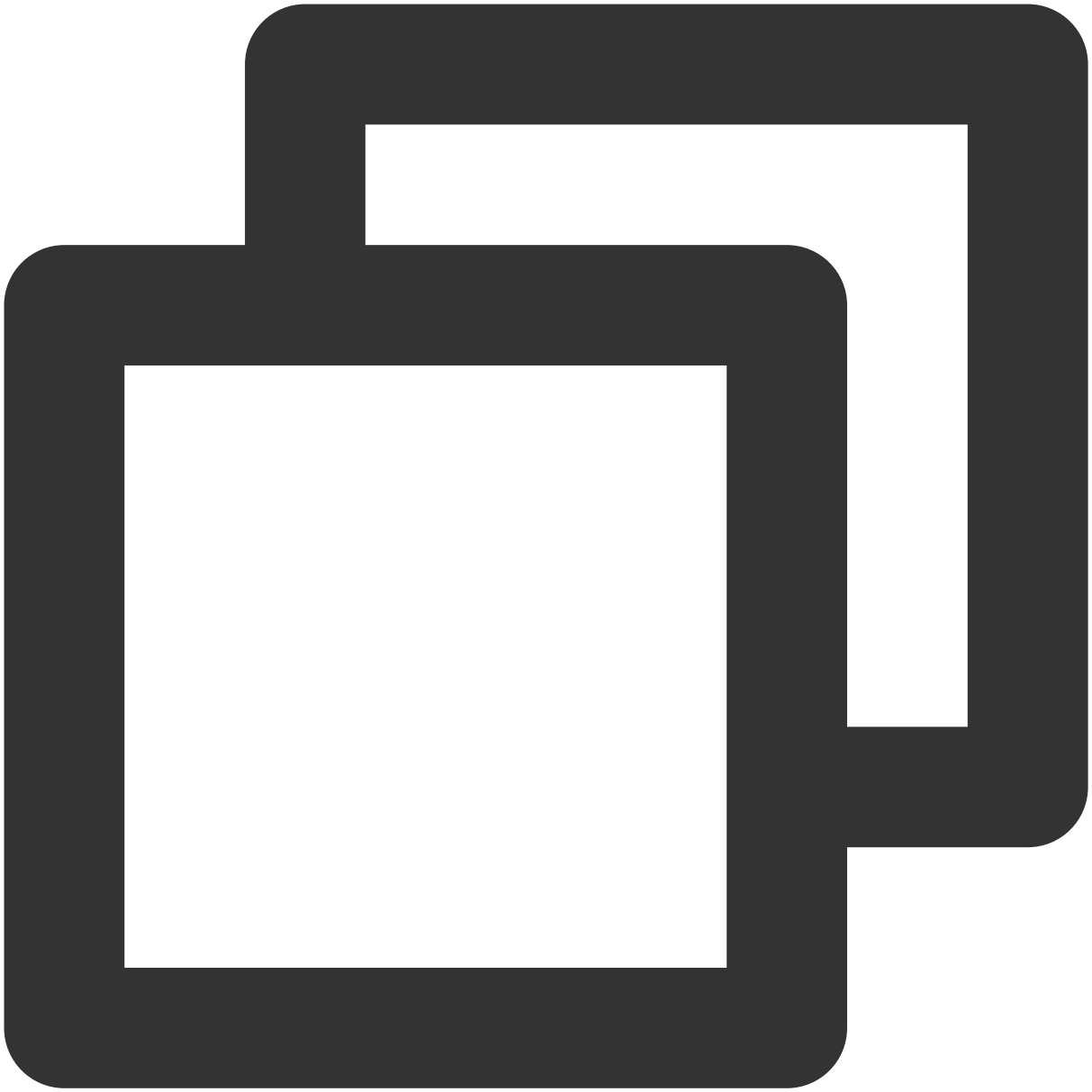
方法3：CoreDNS Forwardプラグインを使用して自作DNSをアップストリームDNSとして設定する

1. forward設定を確認します。forwardのデフォルト設定は次のとおりです。クラスター内以外のドメイン名がCoreDNSの所在ノード `/etc/resolv.conf` ファイル内に設定されたnameserverによって解決されることを指します。



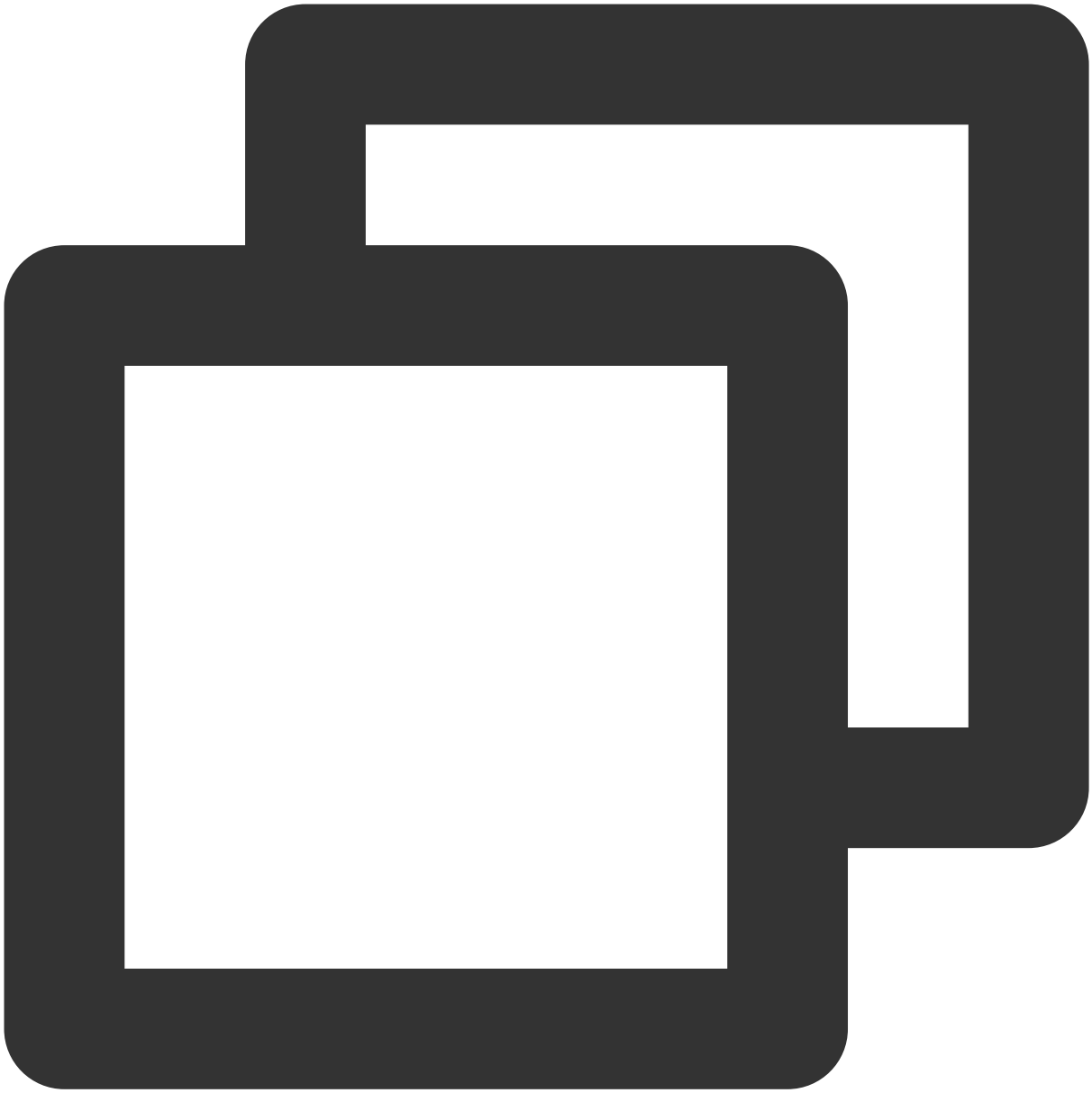
```
forward . /etc/resolv.conf
```

2. forwardを設定し、 `/etc/resolv.conf` を自作DNSサーバーアドレスに明示的に置き換えます。次に例を示します。



```
forward . 10.10.10.10
```

完全な設定の例は次のとおりです。



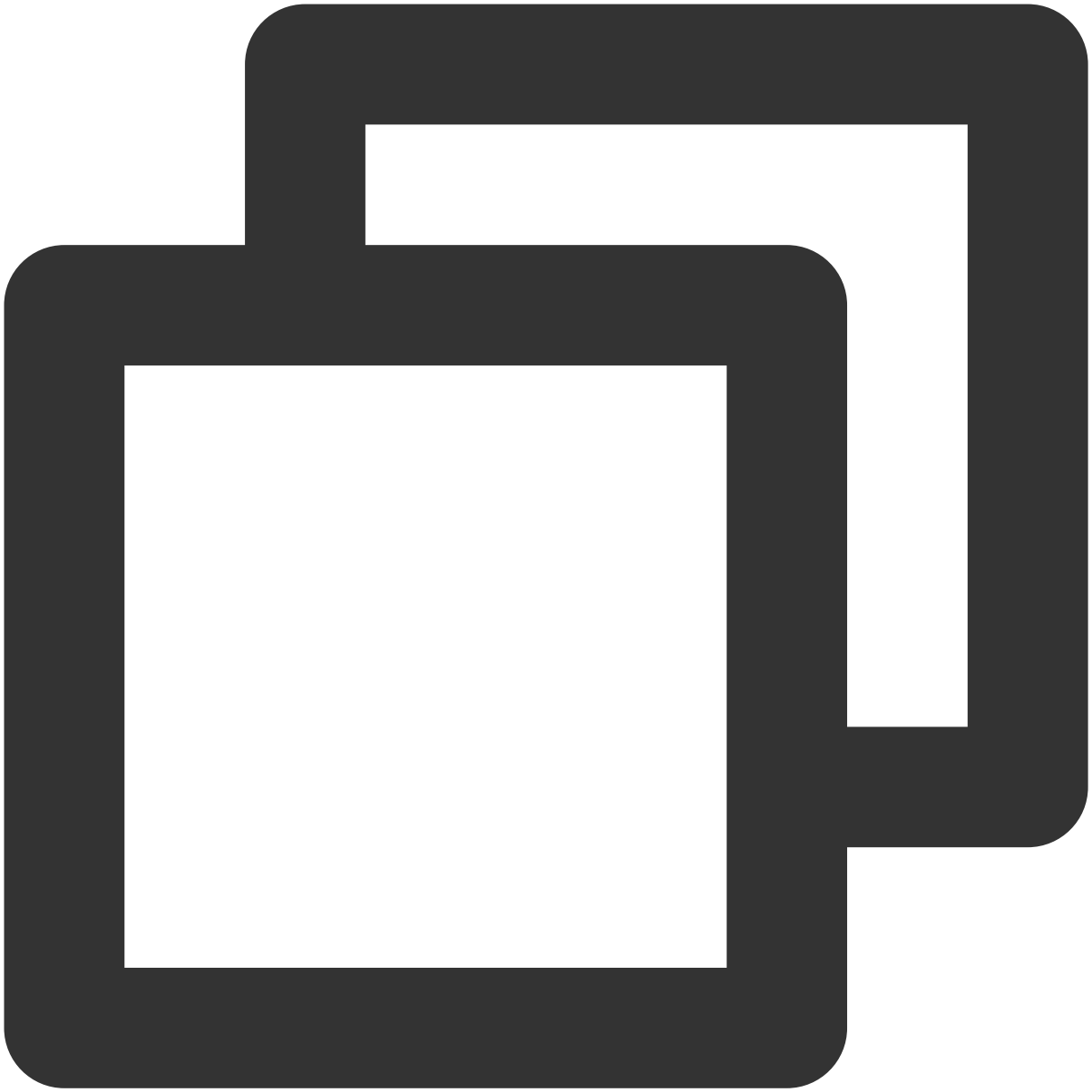
```
apiVersion: v1
data:
   Corefile: |2-
     .:53 {
         errors
         health
         kubernetes cluster.local. in-addr.arpa ip6.arpa {
             pods insecure
             upstream
             fallthrough in-addr.arpa ip6.arpa
         }
```

```
        prometheus :9153
        forward . 10.10.10.10
        cache 30
        reload
        loadbalance
    }
kind: ConfigMap
metadata:
  labels:
    addonmanager.kubernetes.io/mode: EnsureExists
  name: coredns
  namespace: kube-system
```

3. カスタムドメイン名の解決レコードを自作DNSに設定します。ノード上の `/etc/resolv.conf` 内のnameserverを自作DNSのアップストリームに追加することをお勧めします。一部のサービスはTencent Cloud内部のDNS解決に依存しており、上記を自作DNSのアップストリームに設定しなければ、一部のサービスが正常に動作しなくなる場合があるためです。ここではBIND 9を例にとり、プロファイルを変更し、アップストリームDNSアドレスをforwardersに書き込みます。次に例を示します。

注意

自作DNS Serverとリクエスト元が同じRegionにない場合、クロスドメインアクセスをサポートしていない一部のTencentドメイン名が無効になる場合があります。



```
options {
     forwarders {
             183.60.83.19;
             183.60.82.98;
     };
     ...
```

参考ドキュメント

CoreDNS Hostsプラグインドキュメント
CoreDNS Rewriteプラグインドキュメント
CoreDNS Forwardプラグインドキュメント

Nginx Ingressベストプラクティス

最終更新日：：2023-04-28 15:30:11

概要

Tencent Kubernetes Engine（TKE）はNginx-ingress拡張コンポーネントのインストールをサポートし、Nginx-ingressによってIngressトラフィックにアクセスすることができます。Nginx-ingressコンポーネントのその他の紹介については、Nginx-ingressの説明をご参照ください。ここではNginx-ingressコンポーネントの一般的なベストプラクティス操作ガイドをご紹介します。

前提条件

Nginx-ingress拡張コンポーネントをインストール済みであること。

操作手順

クラスターに複数のNginx Ingressトラフィックエントリーを公開する

Nginx-ingress拡張コンポーネントのインストール後、`kube-system`にはNginx-ingressのoperatorコンポーネントがあり、このコンポーネントによって複数のNginx Ingressインスタンスを作成することができ、各Nginx Ingressインスタンスはいずれも異なるIngressClassを使用し、かつ異なるCLBをトラフィックエントリーとして使用し、それによって異なるIngressを異なるトラフィックエントリーバインドすることを実現します。実際のニーズに応じて、クラスターに複数のNginx Ingressインスタンスを作成することができます。

1. TKEコンソールにログインし、左側ナビゲーションバーから**クラスター**を選択します。
2. クラスター管理ページで目標のクラスターIDをクリックし、クラスター詳細ページに進みます。
3. 左側メニューバーの**コンポーネント管理**を選択し、コンポーネントリストページに進みます。
4. インストールしたNginx-ingress拡張コンポーネントをクリックし、コンポーネントページに進みます。
5. **Nginx Ingressインスタンスの追加**をクリックし、必要に応じてNginx Ingressインスタンスを設定し、各インスタンスに異なるIngressClass名を指定します。

説明

Nginx Ingressインスタンスを作成する詳細なステップについては、Nginx-ingressインスタンスのインストールをご参照ください。

6. Ingressの作成時に具体的な IngressClassを指定してIngressを具体的なNginx Ingressインスタンスにバインドすることができます。コンソールまたはYAMLによってIngressを作成することができます。

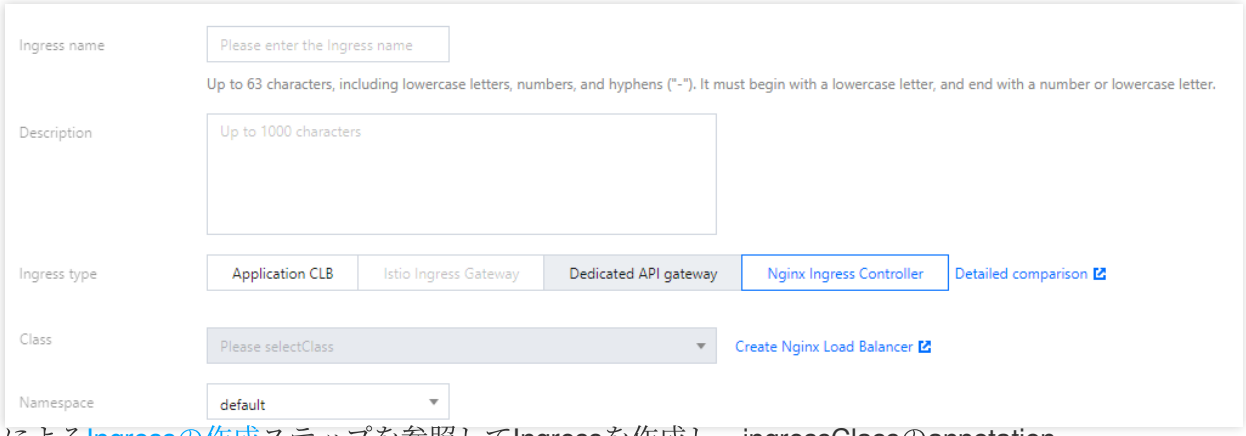
コンソールによるIngressの作成

YAMLによるIngressの作成

コンソールによるIngressの作成ステップを参照してIngressを作成します。このうち、

Ingressタイプ：**NginxのCLB**を選択します。

Class：上記のステップで作成したNginx Ingressインスタンスを選択します。



YAMLによるIngressの作成ステップを参照してIngressを作成し、ingressClassのannotation（ `kubernetes.io/ingress.class` ）を指定します。下図に示すとおりです。

```
1 apiVersion: networking.k8s.io/v1beta1
2 kind: Ingress
3 metadata:
4   annotations:
5     ingress.cloud.tencent.com/direct-access: "false"
6     kubernetes.io/ingress.class: nginx-external
```

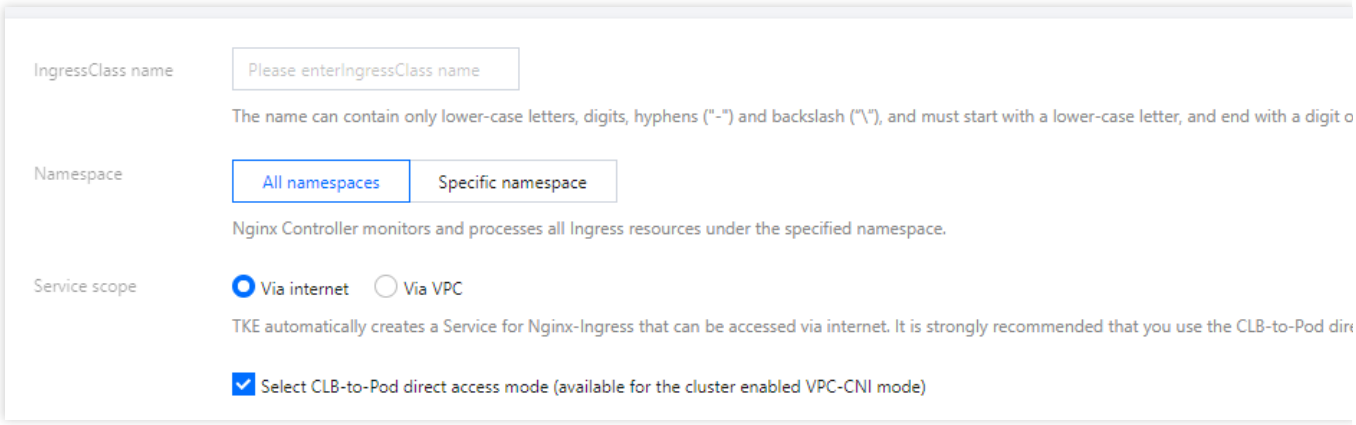
パフォーマンスの最適化

LBパススルーPod

クラスターのネットワークモードがGlobal Routerの場合、デフォルトではLBパススルーPodが有効化されていません。以下のステップに従ってLBパススルーPodを有効化することをおすすめします。

1. クラスターのVPC-CNIを有効化する。

2. Nginx Ingressインスタンスを作成する時、**CLBを使用してPodに直接接続する**にチェックを入れ、トラフィックがNodePortを回避してPodに直接接続できるようにし、それによってパフォーマンスを向上させることができます。下図に示すとおりです。

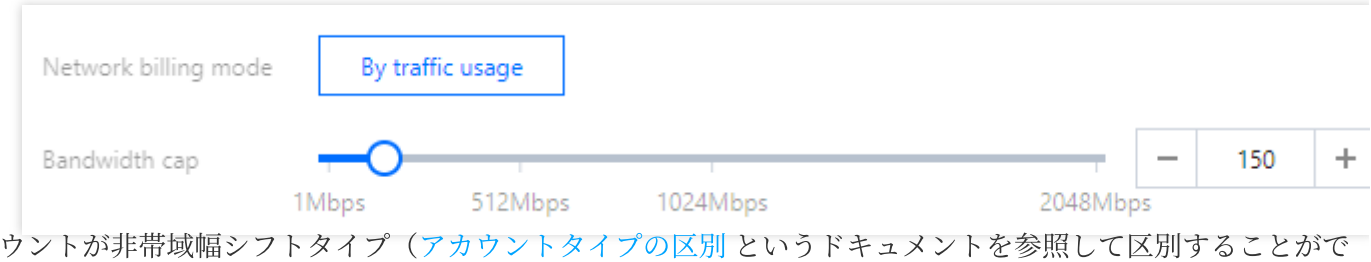


説明

Nginx Ingressインスタンスを作成する詳細なステップについては、Nginx-ingressインスタンスのインストールをご参照ください。

LB帯域幅の上限の引き上げ

LBはトラフィックエントリーとして、高い同時実行またはスループットを必要とする場合、Nginx Ingressインスタンスを作成する時に、実際のニーズに応じて帯域幅の上限を計画し、Nginx Ingressに高い帯域幅を割り当てます。下図に示すとおりです。



アカウントが非帯域幅シフトタイプ（アカウントタイプの区別というドキュメントを参照して区別することができます）の場合、帯域幅の上限はノード帯域幅で決まります。以下の状況に応じてノードの帯域幅の上限を調整することができます。

LBパススルーPodを有効化した場合、LB総帯域幅はNginx Ingressインスタンスの Pod所在ノードの帯域幅の和です。専用の高パブリックネットワーク帯域幅のノードを計画してNginx Ingressインスタンスをデプロイすることをお勧めします（ノードプールのDaemonSetデプロイを指定します）。

LBパススルーPodを使用しない場合、LB総帯域幅はすべてのノードのパブリックネットワーク帯域幅の和です。

Nginx Ingressパラメータを最適化する

Nginx Ingressインスタンスはデフォルトでカーネルパラメータと Nginx Ingress自身の設定に最適化されています。詳細については、Nginx Ingressの高同時実行性の実践をご参照ください。カスタマイズする必要がある場合、以下の紹介を参照して自分で変更してください。

カーネルパラメータの変更

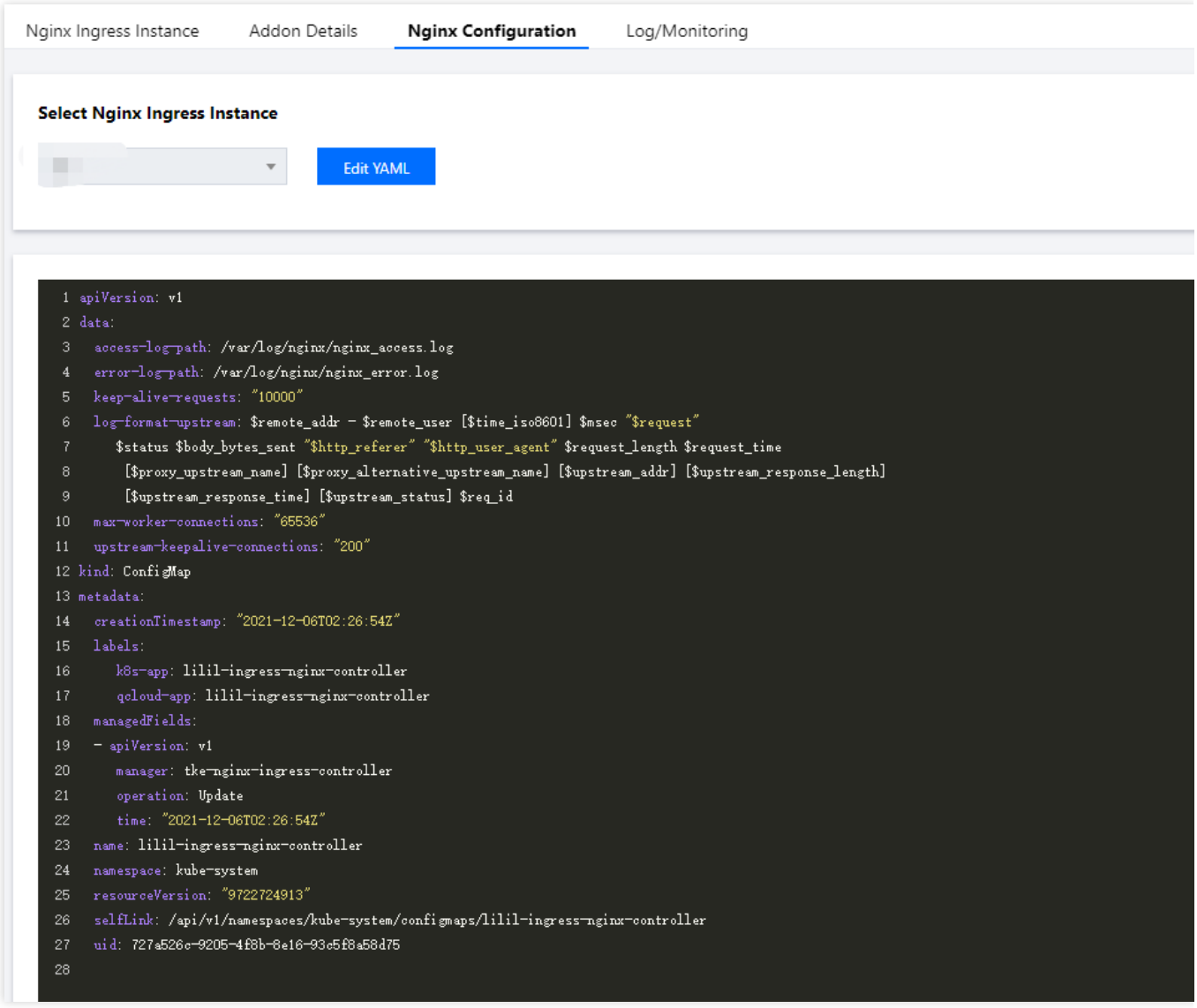
Nginx Ingress自身の設定を変更する

デプロイしたnginx-ingress-conntrollerのDaemonsetまたはDeployment（インスタンスデプロイオプションで決まります）を編集し、initContainers（Kubectlを使用して変更し、コンソールはkube-system下のリソースを変更するこ

とを禁止します）を変更します。下図に示すとおりです。

```
initContainers:
- command:
  - sh
  - -c
  - |-
    sysctl -w net.core.somaxconn=65535
    sysctl -w net.ipv4.ip_local_port_range="1024 65535"
    sysctl -w net.ipv4.tcp_tw_reuse=1
    sysctl -w fs.file-max=1048576
```

Nginx設定で対応するインスタンスを選択し、**YAMLの編集**をクリックすると、Nginx Ingressインスタンスの ConfigMap設定を変更することができます。下図に示すとおりです。



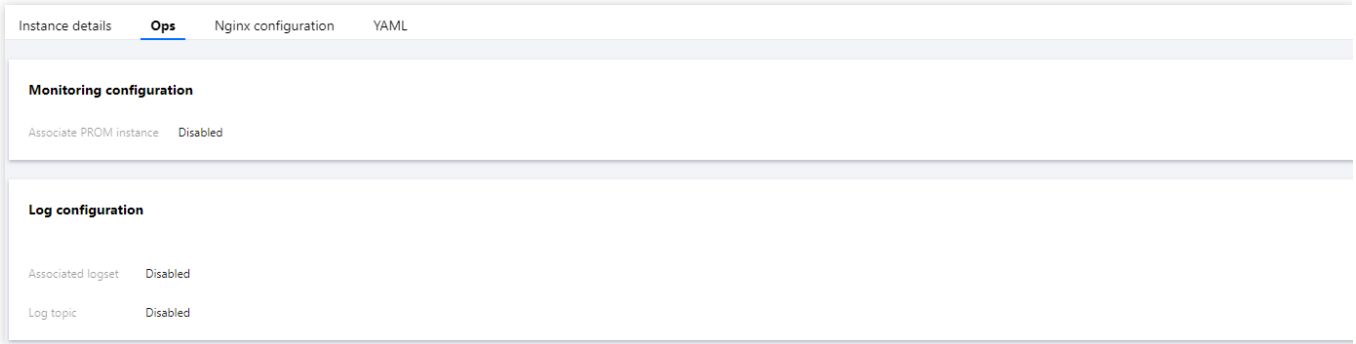
説明

ConfigMap設定の詳細については公式ドキュメントをご参照ください。

Nginx Ingressの監視可能性を向上させる

ログを有効化する

説明：
ログはCLSに依存します。有効化したい場合は、Nginx-ingressのログ設定をご参照ください。
Nginx Ingressインスタンスを作成した後、インスタンス詳細の**運用保守**機能エントリー内でインスタンスの**ログ**を有効化することができ、インスタンスの各項目の状態指標を確認し、トラブルシューティングを可能にします。下図に示すとおりです。



注意：
バージョンv0.49.3のインスタンス、ログ収集のインデックス設定ファイルはLogConfigという名前のCRDリソースオブジェクト内に存在し、このリソースオブジェクトを変更した後に、ログ収集機能をオフ/再びオンにした場合、この LogConfigのリソースオブジェクト設定はリセットされます。適時このリソースオブジェクト内のデータをバックアップしてください。Nginx Ingressインスタンス自体の削除およびNginx Ingressコンポーネントのアップグレードはこのインデックス設定ファイルに影響はありません。
ログをカスタマイズする必要がある場合は、ドキュメントに従って設定してください。

ログ検索とログダッシュボード

ログの設定を有効化すると、Nginx Ingressリストページでインスタンス右側の**操作**の**その他**をクリックし、ポップアップしたメニュー内で対応する機能を選択してログ検索またはログダッシュボードの確認を行うことができます。
CLSに移動してアクセスログを確認するをクリックしてログサービスにジャンプし、**検索分析**でインスタンスに対応するログセットとトピックを選択すると、Nginx Ingressのアクセスとエラーログを確認することができます。
アクセスログダッシュボードの確認をクリックすると、Nginx Ingressのログデータに基づいて統計情報を表示するダッシュボードに直接ジャンプすることができます。

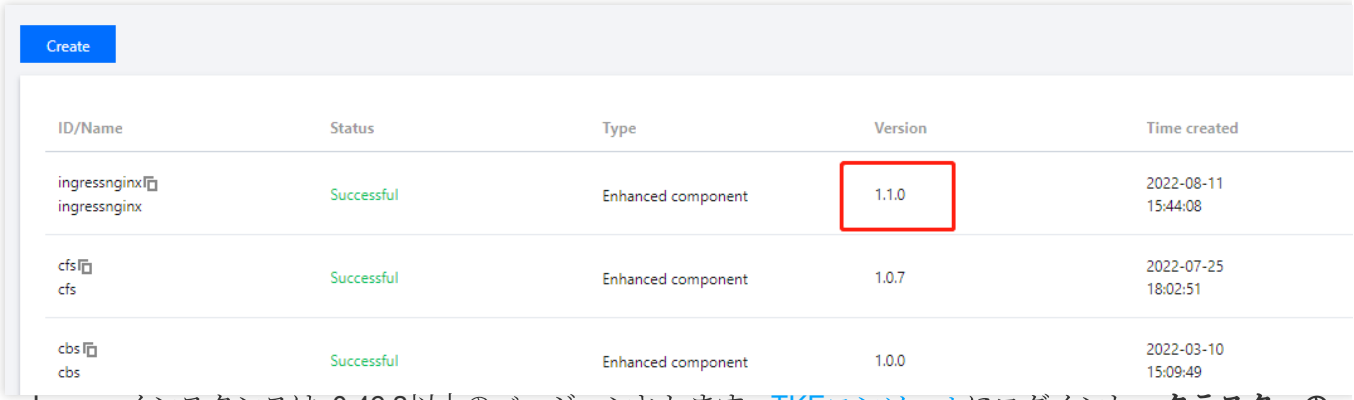
ログ
NginxIngressカスタムログ

最終更新日：：2023-04-28 11:08:19

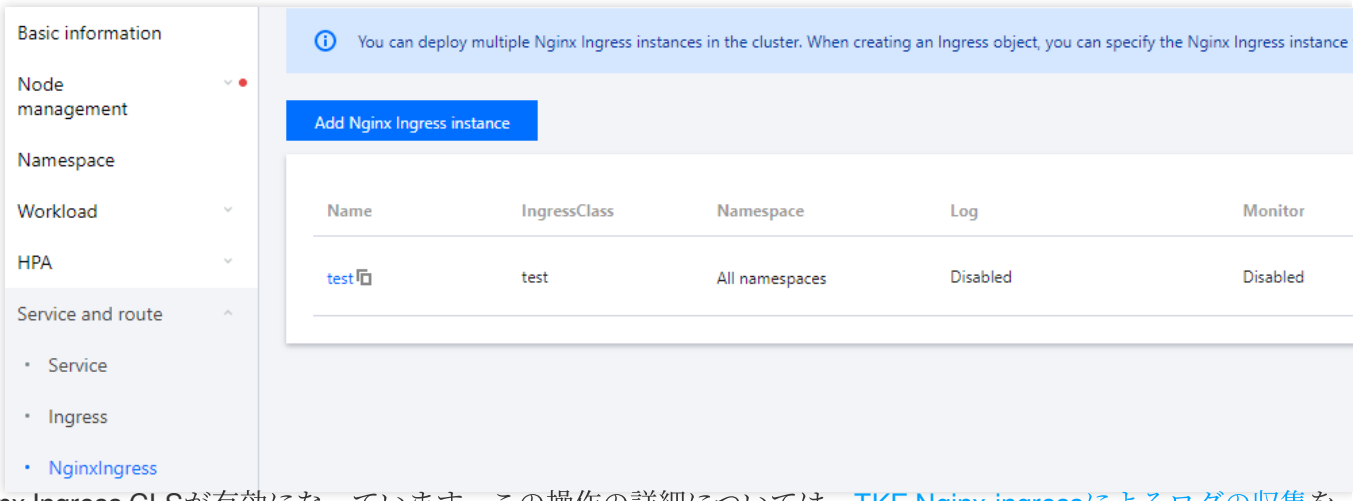
TKEは、CLSとの統合により、フルセットかつパーフェクトな製品化機能を提供し、Nginx-ingressログの収集および消費機能を実装します。詳細については、Nginx-ingressログの設定をご参照ください。デフォルトのログインデックスではログのニーズを満たせない場合、ログインデックスをカスタマイズすることができます。ここでは、Nginx Ingressのログインデックスの更新方法についてご説明します。

前提条件

1. Nginx Ingressはv1.1.0以上のバージョンとします。TKEコンソールにログインし、**クラスターの詳細 > コンポーネント管理**でNginx Ingressのコンポーネントバージョンをご確認ください。
注意：
Nginx Ingress v1.1.0以上のバージョンのみがこの機能をサポートしています。v1.0.0など、v1.1.0以下のバージョンでは、ユーザーによるログインデックスの変更は、コンポーネントのロールバックによって上書きされます。



2. Nginx Ingressインスタンスはv0.49.3以上のバージョンとします。TKEコンソールにログインし、**クラスターの詳細 > サービスとルート**から**NginxIngress**を選択してインスタンスの右側にある**YAMLの確認**をクリックします。YAMLでは、イメージ `ccr.ccs.tencentyun.com/paas/nginx-ingress-controller` のバージョンはv0.49.3以上である必要があります。



3. Nginx Ingress CLSが有効になっています。この操作の詳細については、TKE Nginx-ingressによるログの収集をご参照ください。

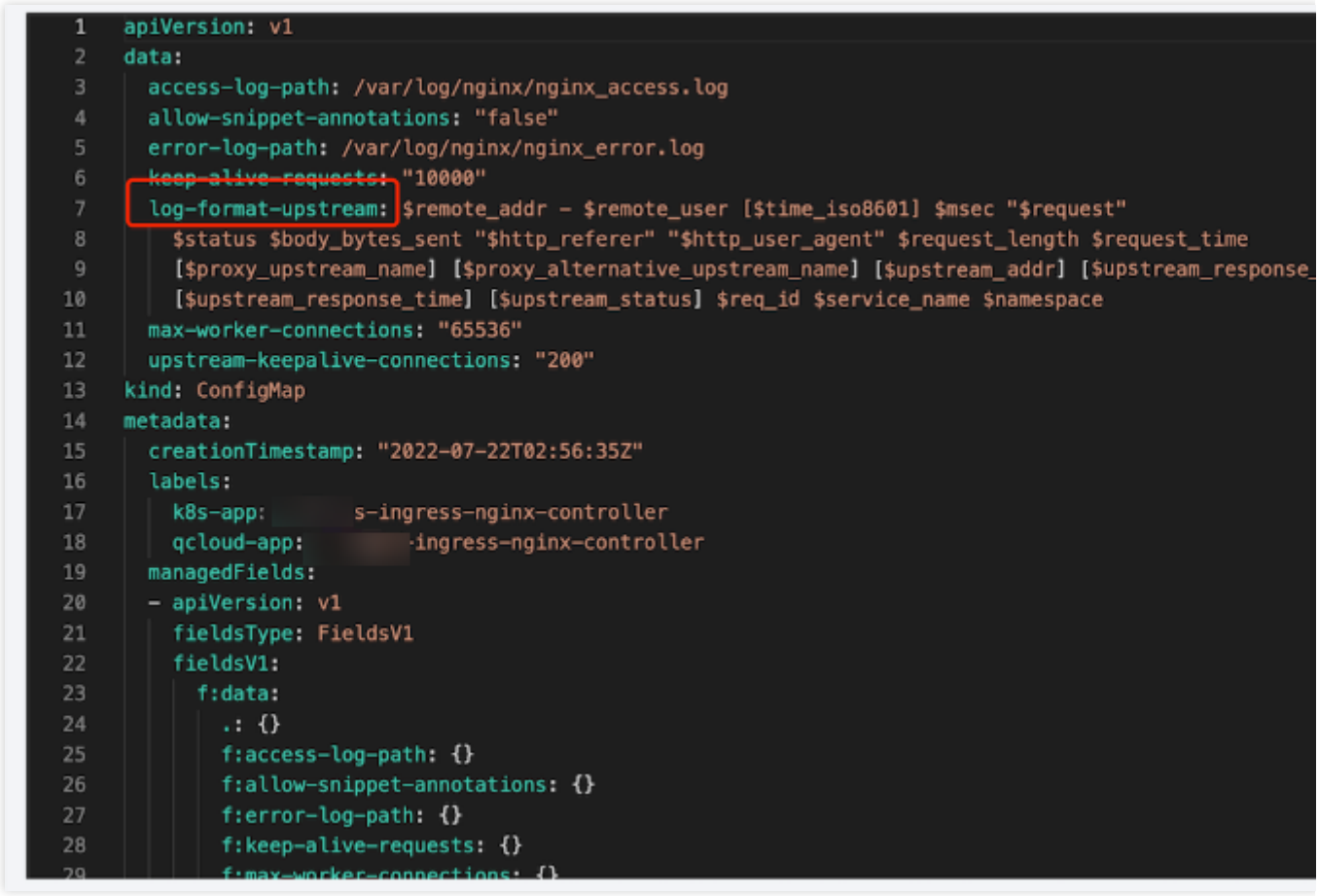
操作手順

注意

ログ構造を変更するには、ログ出力、ログ収集、ログインデックスの設定など、Nginx Ingressのログフローを理解する必要があります。中でもログ出力やログ収集が不十分であったり、設定が間違っていたりすると、ログの変更に失敗してしまいます。

ステップ1：Nginx Ingressインスタンスのログ出力形式の変更

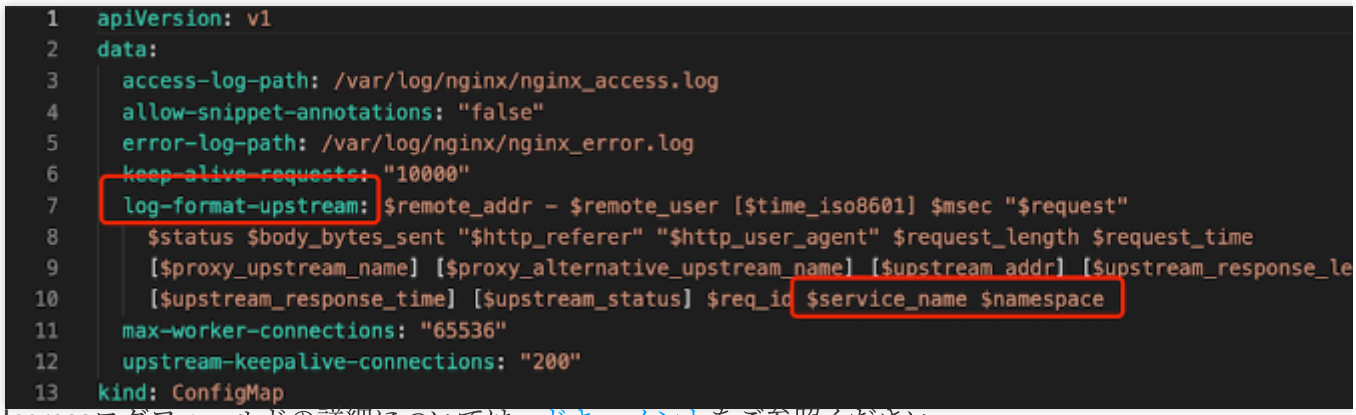
Nginx Ingressインスタンスのログ設定は、そのインスタンスのメイン設定ConfigMapにあります。ConfigMapの名前は、 `インスタンス名-ingress-nginx-controller` とします。変更するKeyは `log-format-upstream` です。下図に示すとおりです。



```
apiVersion: v1
data:
  access-log-path: /var/log/nginx/nginx_access.log
  allow-snippet-annotations: "false"
  error-log-path: /var/log/nginx/nginx_error.log
  keep-alive-requests: "10000"
  log-format-upstream: $remote_addr - $remote_user [$time_iso8601] $msec "$request"
    $status $body_bytes_sent "$http_referer" "$http_user_agent" $request_length $request_time
    [$proxy_upstream_name] [$proxy_alternative_upstream_name] [$upstream_addr] [$upstream_response_
    [$upstream_response_time] [$upstream_status] $req_id $service_name $namespace
  max-worker-connections: "65536"
  upstream-keepalive-connections: "200"
kind: ConfigMap
metadata:
  creationTimestamp: "2022-07-22T02:56:35Z"
  labels:
    k8s-app:       s-ingress-nginx-controller
    qcloud-app:          -ingress-nginx-controller
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:data:
        .: {}
        f:access-log-path: {}
        f:allow-snippet-annotations: {}
        f:error-log-path: {}
        f:keep-alive-requests: {}
        f:max-worker-connections: {}
```

事例

ログに$namespace と $service_name`という2つの連続する文字列を追加し、ログ内容の末尾に配置します。追加する場所は、下図に示すとおりです。

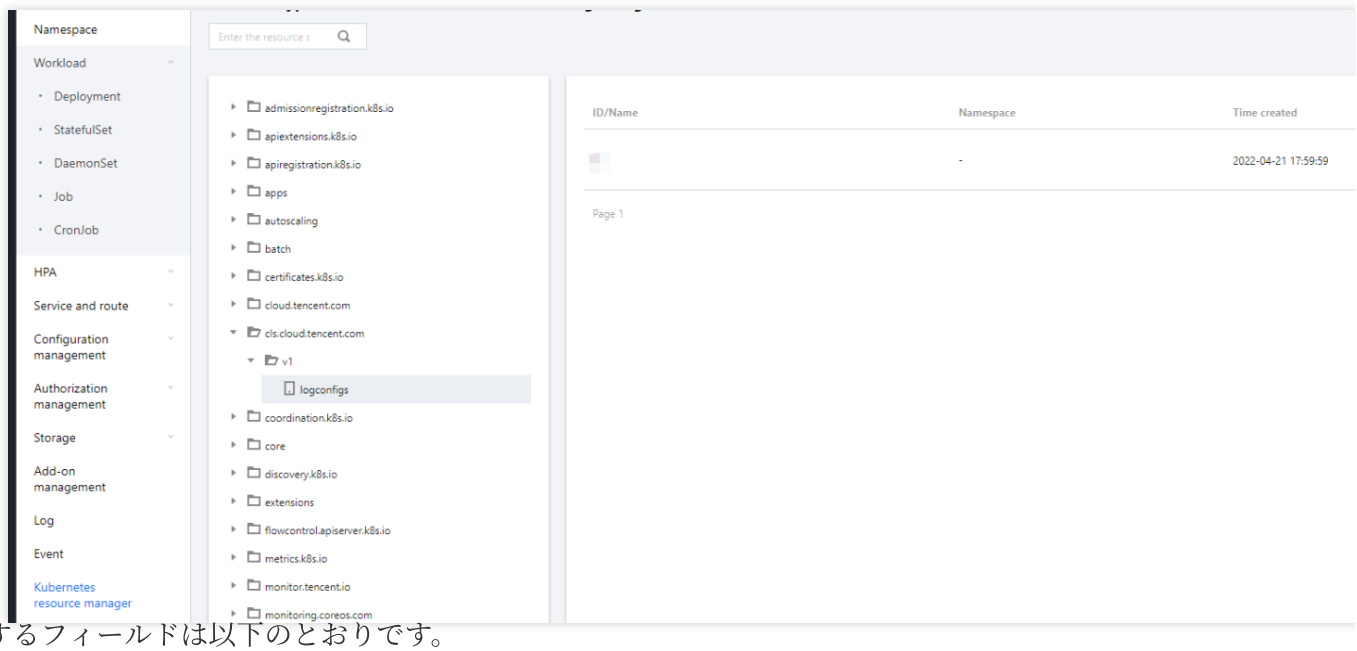


```
apiVersion: v1
data:
  access-log-path: /var/log/nginx/nginx_access.log
  allow-snippet-annotations: "false"
  error-log-path: /var/log/nginx/nginx_error.log
  keep-alive-requests: "10000"
  log-format-upstream: $remote_addr - $remote_user [$time_iso8601] $msec "$request"
    $status $body_bytes_sent "$http_referer" "$http_user_agent" $request_length $request_time
    [$proxy_upstream_name] [$proxy_alternative_upstream_name] [$upstream_addr] [$upstream_response_le
    [$upstream_response_time] [$upstream_status] $req_id $service_name $namespace
  max-worker-connections: "65536"
  upstream-keepalive-connections: "200"
kind: ConfigMap
```

Nginx Ingressログフィールドの詳細については、ドキュメントをご参照ください。

ステップ2：クラスター内ログ収集とレポートAgent形式の変更

クラスター内ログ収集のルールは、logconfigs.cls.cloud.tencent.comタイプのリソースオブジェクトにあります。TKEコンソールにログインし、**クラスターの詳細 > リソースオブジェクトブラウザ**でこのリソースオブジェクトを見つけられます。名前は インスタンス名-ingress-nginx-controller です。**YAMLの編集**で変更することができます。



変更するフィールドは以下のとおりです。

beginningRegex：ログ開始の正規表現

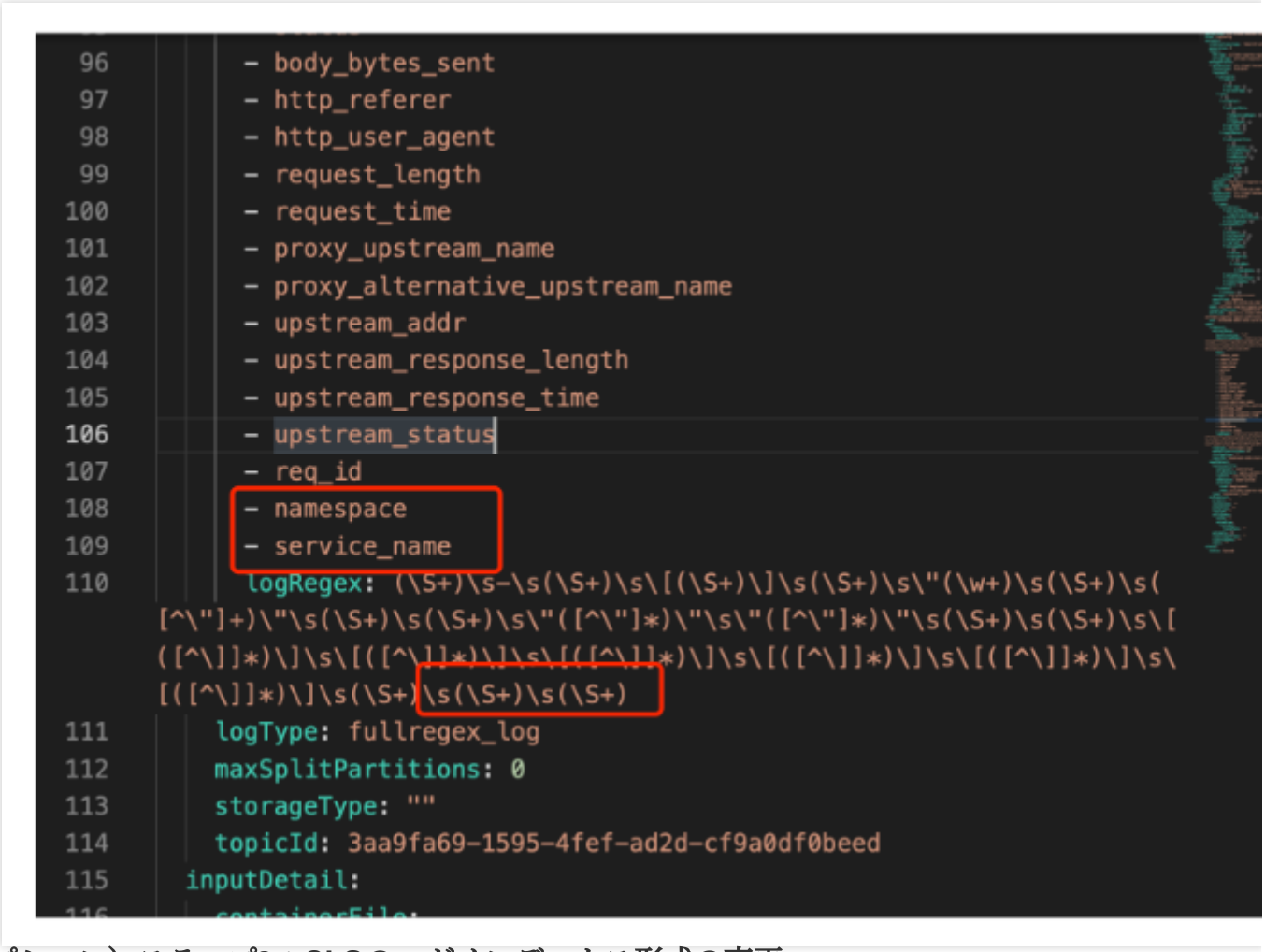
keys：ログのフィールド

logRegex：ログ終了の正規表現

正規表現は、Nginxのログ行形式とマッチします。Nginxの既存のログ形式の後にフィールドを追加するとともに、keysの末尾で宣言することをお勧めします。また、このフィールドの正規解析をbeginningRegex、logRegexの末尾に追加します。

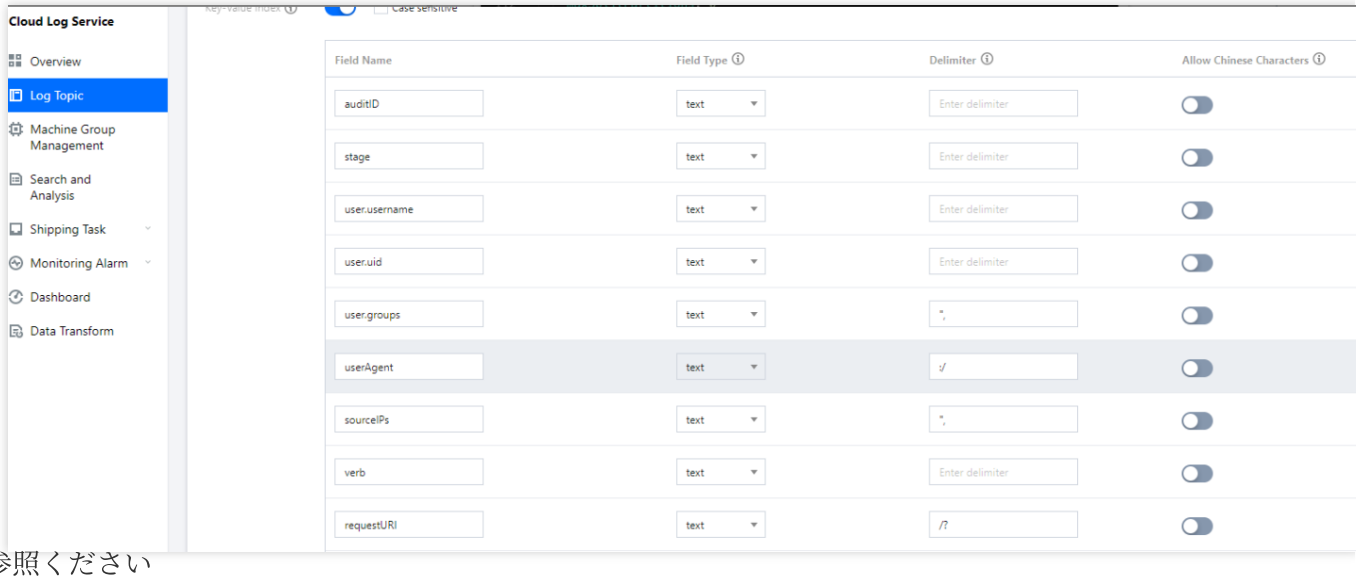
事例

keysの後にステップ1の2つのフィールドを追加した後、beginningRegexとlogRegexの末尾にそれぞれ正規表現文字列を追加します。下図に示すとおりです。



（オプション）ステップ3：CLSのログインデックス形式の変更

このフィールドを検索する機能が必要な場合は、対応するログトピックに新しいフィールドのインデックスを追加する必要があります。これはCLSコンソールで行うことができ、操作を完了すると、収集されたすべてのログがインデックスを介して検索できるようになります。操作の詳細については、インデックスの設定。



をご参照ください

初期設定への復元

ログルールの変更に関する手順が複雑で正規表現にも影響を与えるため、操作手順に何らかの誤りがあると、ログ収集が失敗することがあります。ログ収集エラーが報告された場合、元のログ収集機能を復元することをお勧めします。ログ収集機能をオフにしてから、再度ログ収集のオンを行う必要があります。

監視
PrometheusでMySQLとMariaDBをモニタリングします

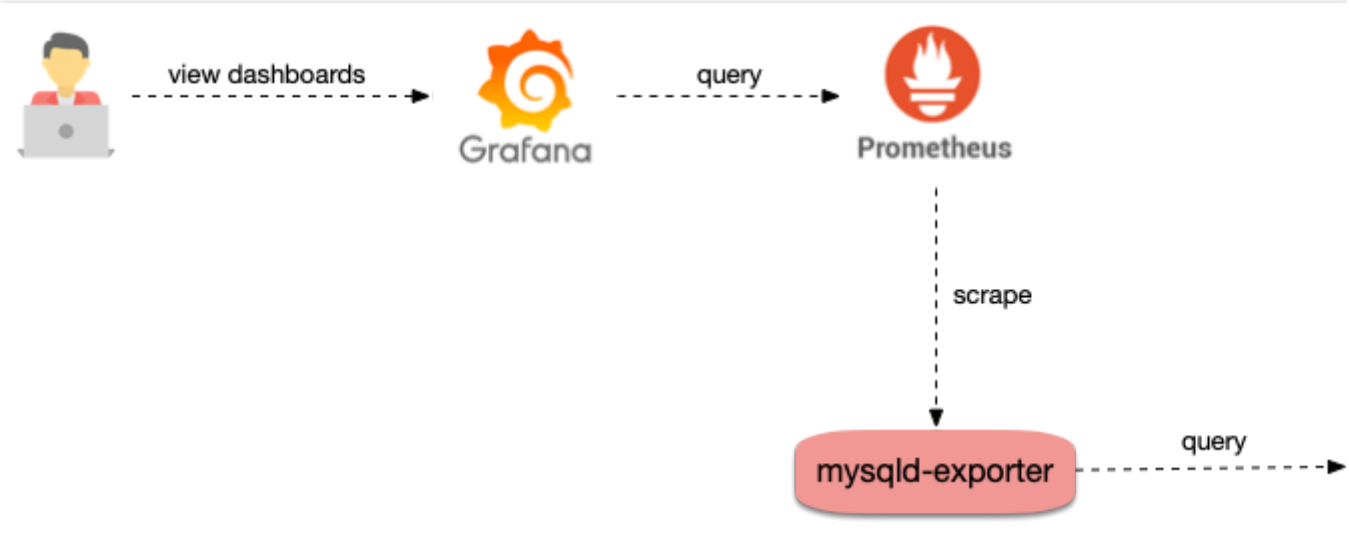
最終更新日：：2023-04-26 18:43:22

ユースケース

MySQLは、一般的に使われているリレーショナルデータベースです。MariaDBは、MySQのブランチバージョンとして、MySQLプロトコルと互換性があり、普及が進んでいます。Kubernetesの環境では、オープンソースmysqld-exporterの助けを借りてPrometheusを使用し、MySQLとMariaDBをモニタリングできます。ここでPrometheusについて学び、使用を開始してください。

mysqld-exporterの概要

mysqld-exporterは、MySQLやMariaDBの特定のデータベースのステータスに関するデータを読み込み、Prometheusのメトリクス形式に変換してHTTPインターフェースとして公開し、PrometheusによってキャプチャされることでメトリクスをサポートしていないMySQLやMariaDBをPrometheusでモニタリングできるようにします。下図に示すとおりです。



操作手順

mysqld-exporterのデプロイ

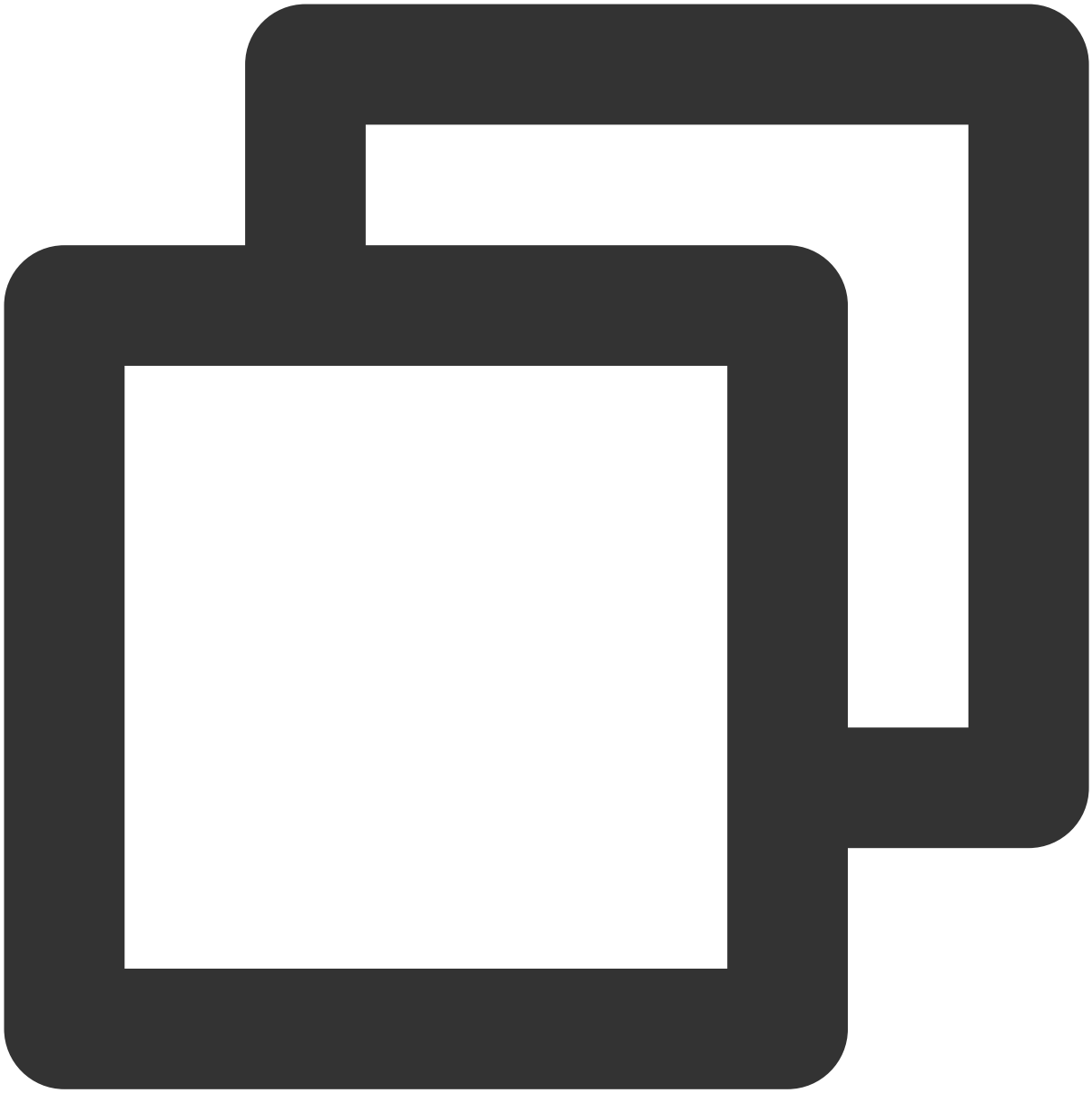
注意

mysqld-exporterをデプロイする前に、クラスター内、クラスター外、または既存のクラウドサービスを使用してMySQLまたはMariaDBをデプロイしていることを確認してください。

MySQLのデプロイ

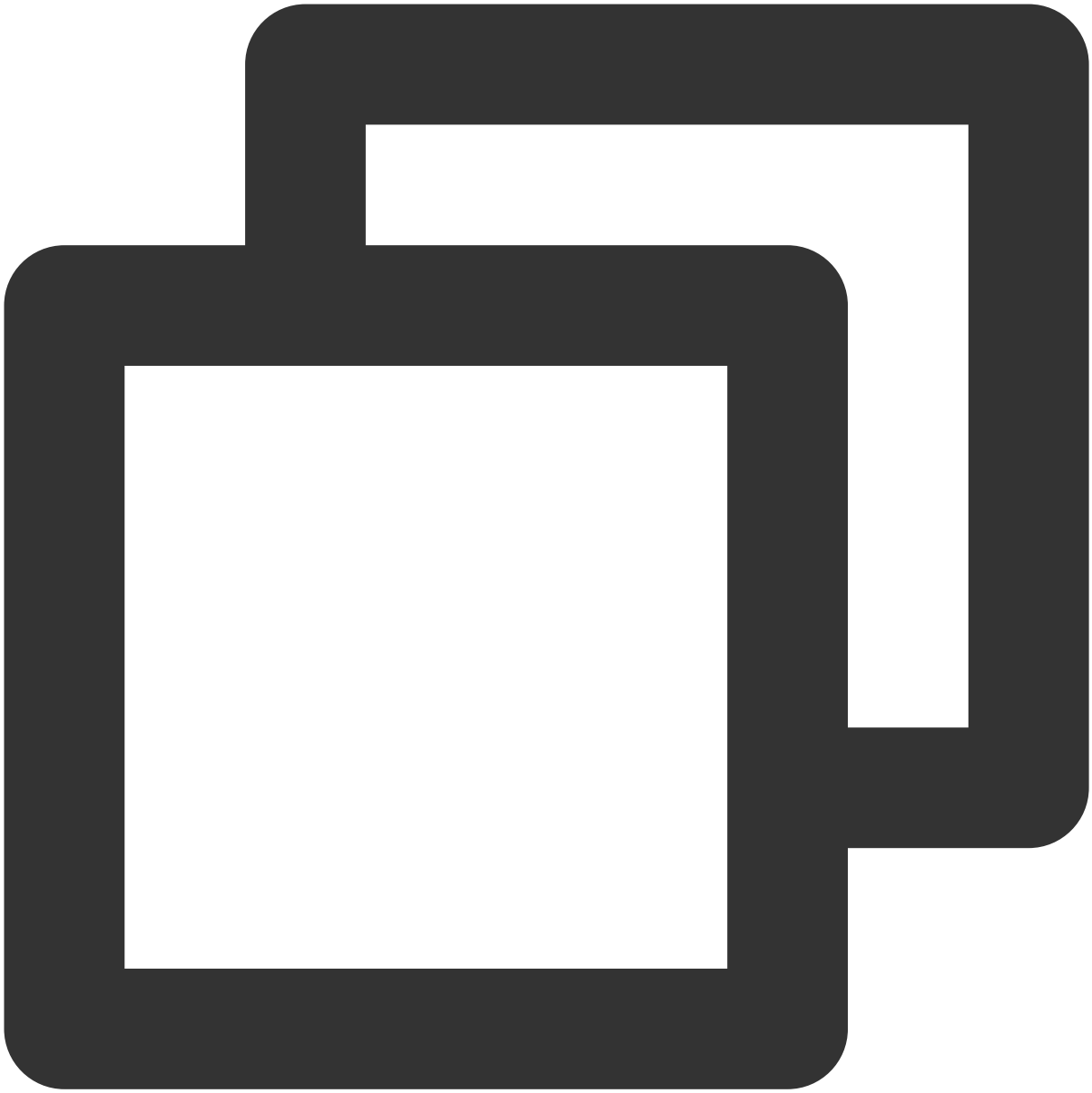
アプリマーケットプレイスからMySQLをクラスターにデプロイすることを例に取ります。手順は次のとおりです。

1. TKEコンソールにログインし、左側ナビゲーションバーから**アプリマーケットプレイス**を選択します。
2. **アプリマーケットプレイス**ページで、**MySQL**を検索し、選択します。
3. **アプリケーションの詳細**ページで、**アプリケーションの作成**をクリックします。
4. **アプリケーションの作成**ページで、アプリケーションの情報を入力し、**作成**をクリックします。
5. アプリケーションの作成が完了したら、左側ナビゲーションバーから**アプリケーション**を選択し、**アプリケーション**のページでアプリケーションの詳細を確認します。
6. 以下のコマンドを実行し、MySQLが正常に動作しているか確認します。



```
$ kubectl get pods
NAME                     READY   STATUS        RESTARTS   AGE
mysql-698b898bf7-4dc5k   1/1     Running       0          11s
```

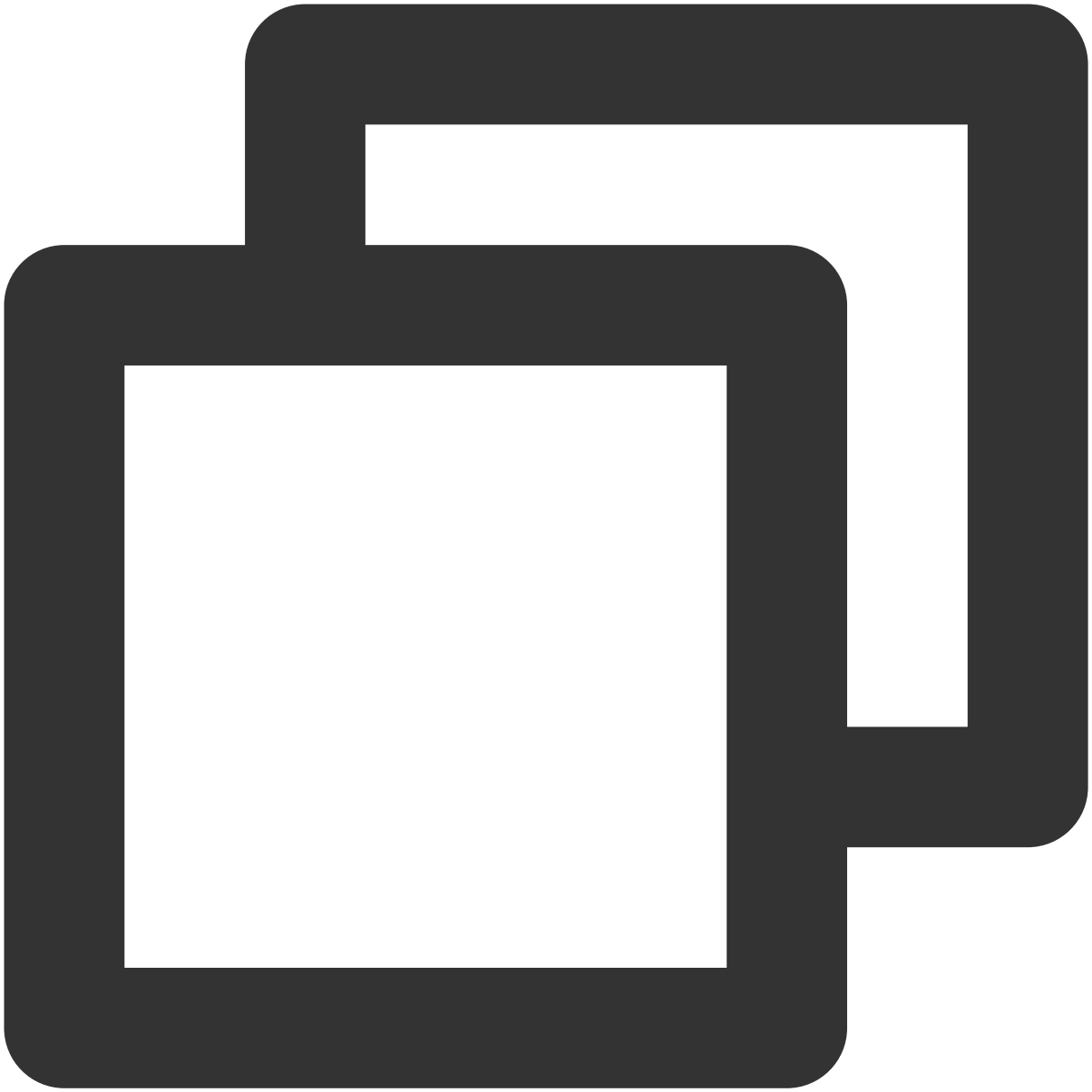
7. 以下のコマンドを実行し、rootパスワードを取得します。



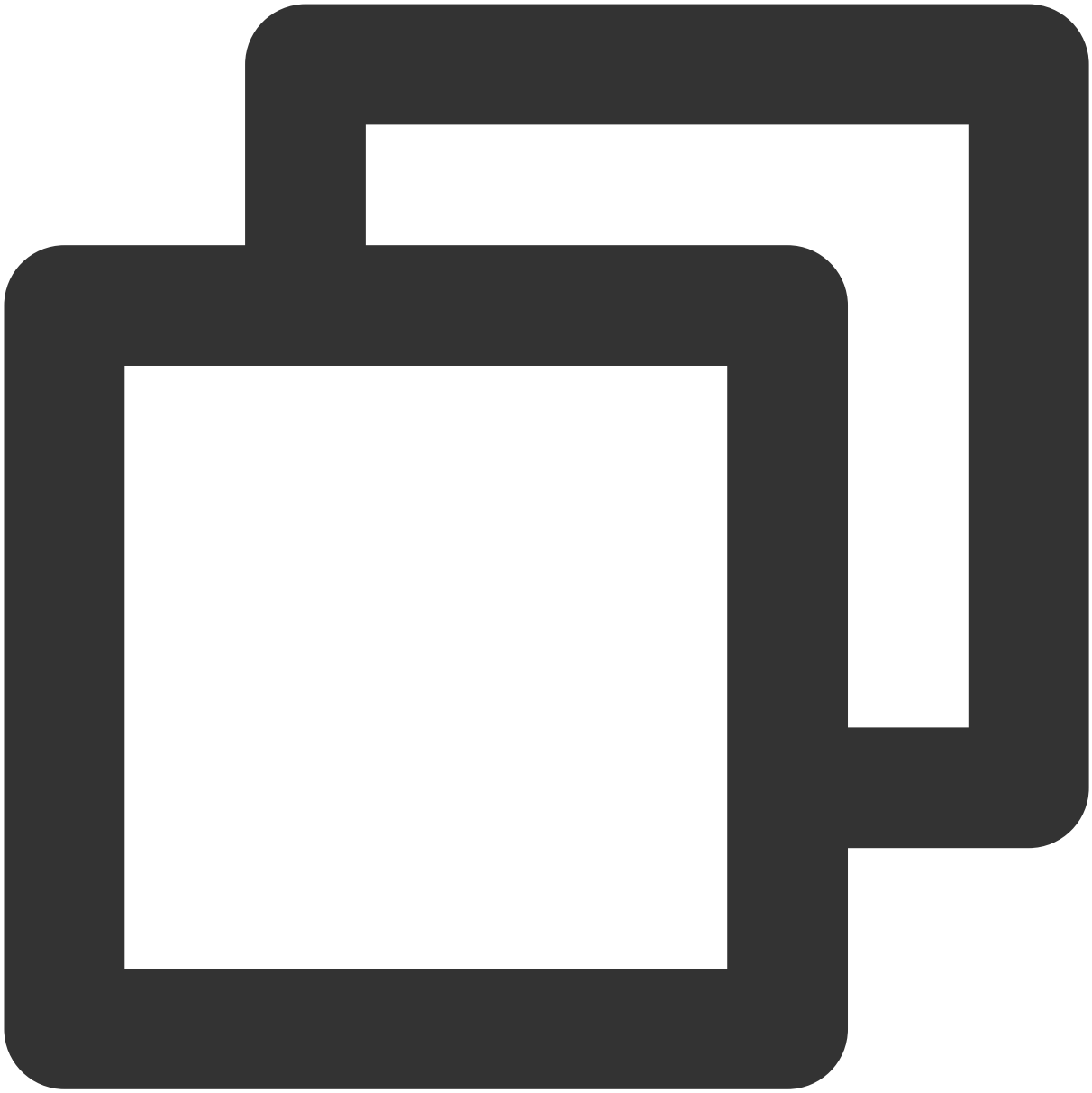
```
$ kubectl get secret -o jsonpath={.data.mysql-root-password} mysql | base64 -d
6ZAj33yLBo
```

mysqld-exporterのデプロイ

MySQLのデプロイを行うと、mysqld-exporterのデプロイを開始できるようになります。手順は次のとおりです。
1. 以下のコマンドを実行し、mysqld-exporterのアカウントを作成し、MySQLにログインします。以下に例を示します。

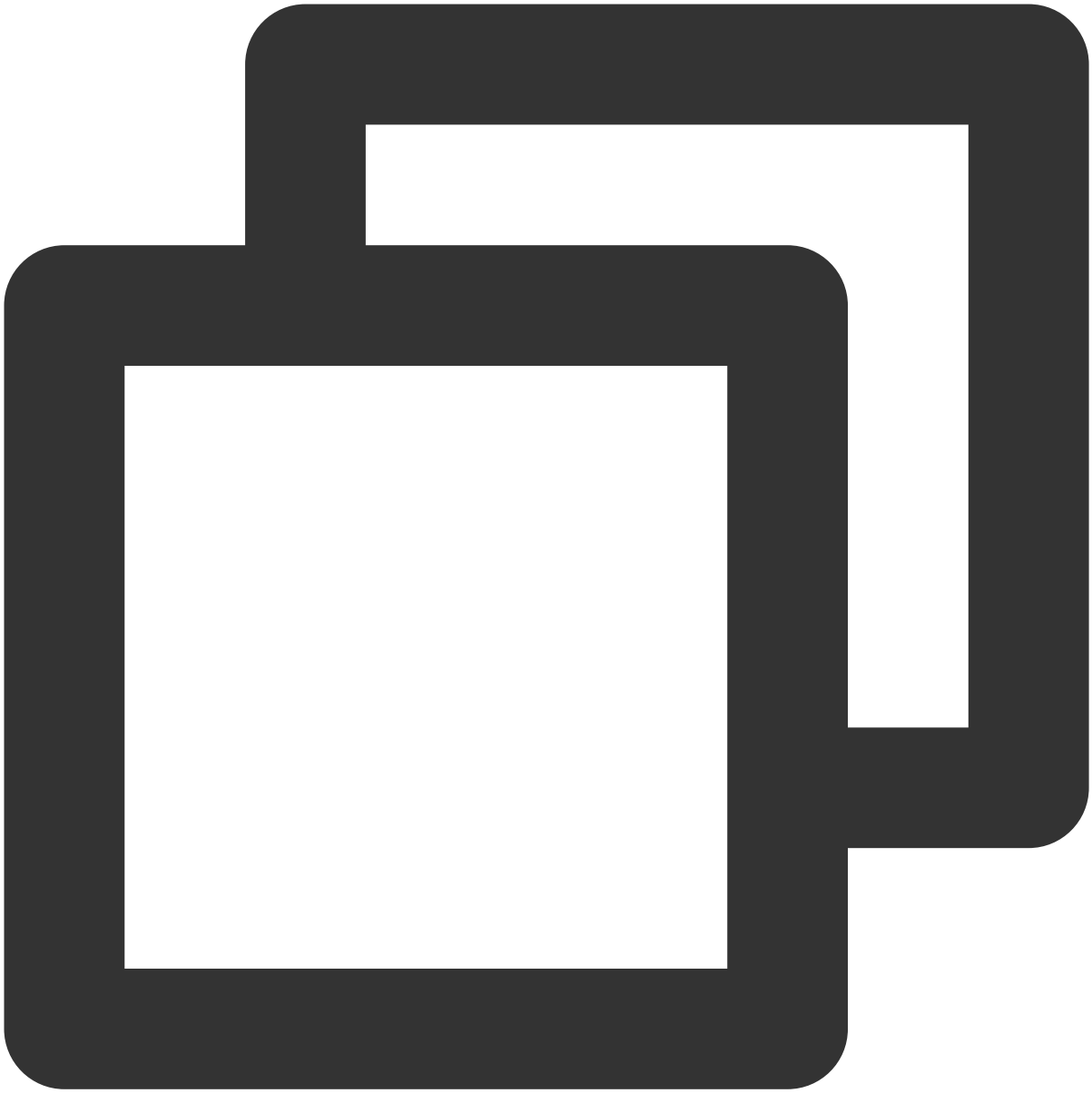


```
$ kubectl exec -it mysql-698b898bf7-4dc5k bash
```



```
$ mysql -uroot -p6ZAj33yLBo
```

2. 以下のコマンドを実行し、SQLステートメントを入力してアカウントを作成します。`mysqld-exporter/123456` を例とした場合、次のようになります。

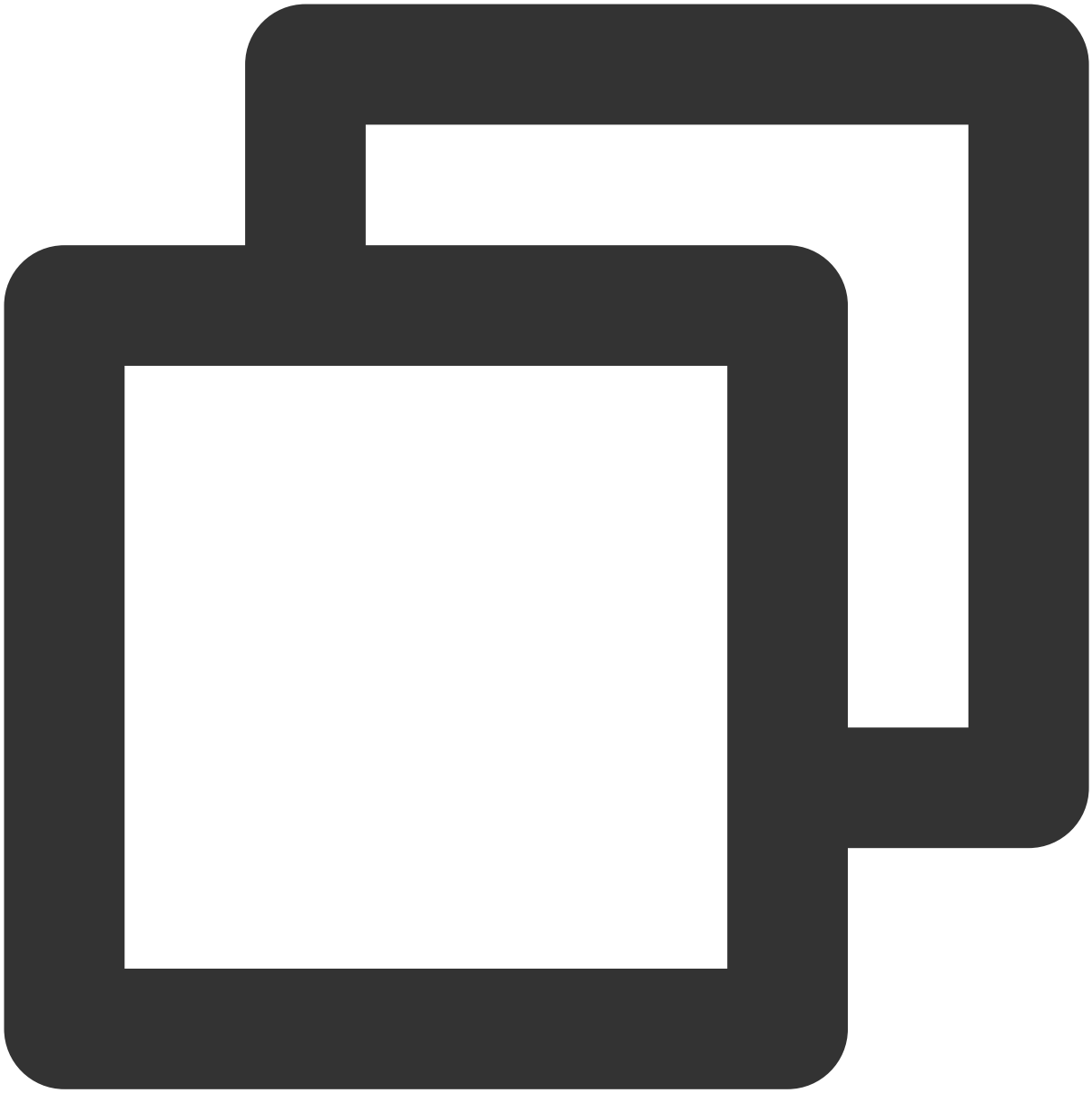


```
CREATE USER 'mysqld-exporter' IDENTIFIED BY '123456' WITH MAX_USER_CONNECTIONS 3;
GRANT PROCESS, REPLICATION CLIENT, REPLICATION SLAVE, SELECT ON *.* TO 'mysqld-expo
flush privileges;
```

3. yamlファイルを使用してmysqld-exporterをデプロイします。以下に例を示します。

注意

実際の状況に応じて、DATA_SOURCE_NAMEのアカウントパスワードとMySQLの接続アドレスを置き換える必要があります。



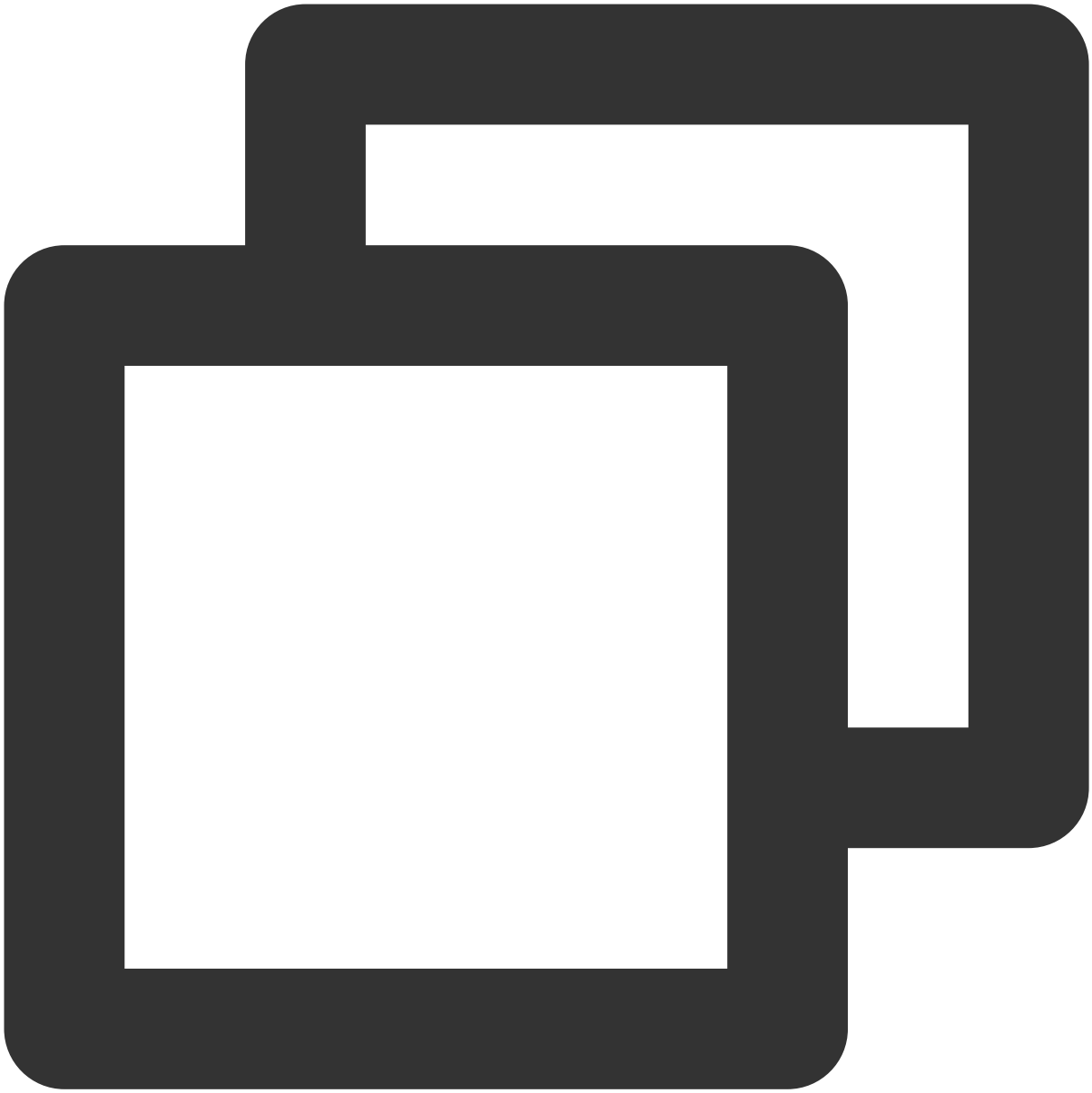
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysqld-exporter
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysqld-exporter
  template:
    metadata:
```

```
      labels:
        app: mysqld-exporter
    spec:
      containers:
      - name: mysqld-exporter
        image: prom/mysqld-exporter:v0.12.1
        args:
        - --collect.info_schema.tables
        - --collect.info_schema.innodb_tablespaces
        - --collect.info_schema.innodb_metrics
        - --collect.global_status
        - --collect.global_variables
        - --collect.slave_status
        - --collect.info_schema.processlist
        - --collect.perf_schema.tablelocks
        - --collect.perf_schema.eventsstatements
        - --collect.perf_schema.eventsstatementssum
        - --collect.perf_schema.eventswaits
        - --collect.auto_increment.columns
        - --collect.binlog_size
        - --collect.perf_schema.tableiowaits
        - --collect.perf_schema.indexiowaits
        - --collect.info_schema.userstats
        - --collect.info_schema.clientstats
        - --collect.info_schema.tablestats
        - --collect.info_schema.schemastats
        - --collect.perf_schema.file_events
        - --collect.perf_schema.file_instances
        - --collect.perf_schema.replication_group_member_stats
        - --collect.perf_schema.replication_applier_status_by_worker
        - --collect.slave_hosts
        - --collect.info_schema.innodb_cmp
        - --collect.info_schema.innodb_cmpmem
        - --collect.info_schema.query_response_time
        - --collect.engine_tokudb_status
        - --collect.engine_innodb_status
        ports:
        - containerPort: 9104
          protocol: TCP
        env:
        - name: DATA_SOURCE_NAME
          value: "mysqld-exporter:123456@(mysql.default.svc.cluster.local:3306)/"
--
apiVersion: v1
kind: Service
metadata:
  name: mysqld-exporter
```

```
    labels:
      app: mysqld-exporter
  spec:
    type: ClusterIP
    ports:
    - port: 9104
      protocol: TCP
      name: http
    selector:
      app: mysqld-exporter
```

モニタリングキャプチャ設定の追加

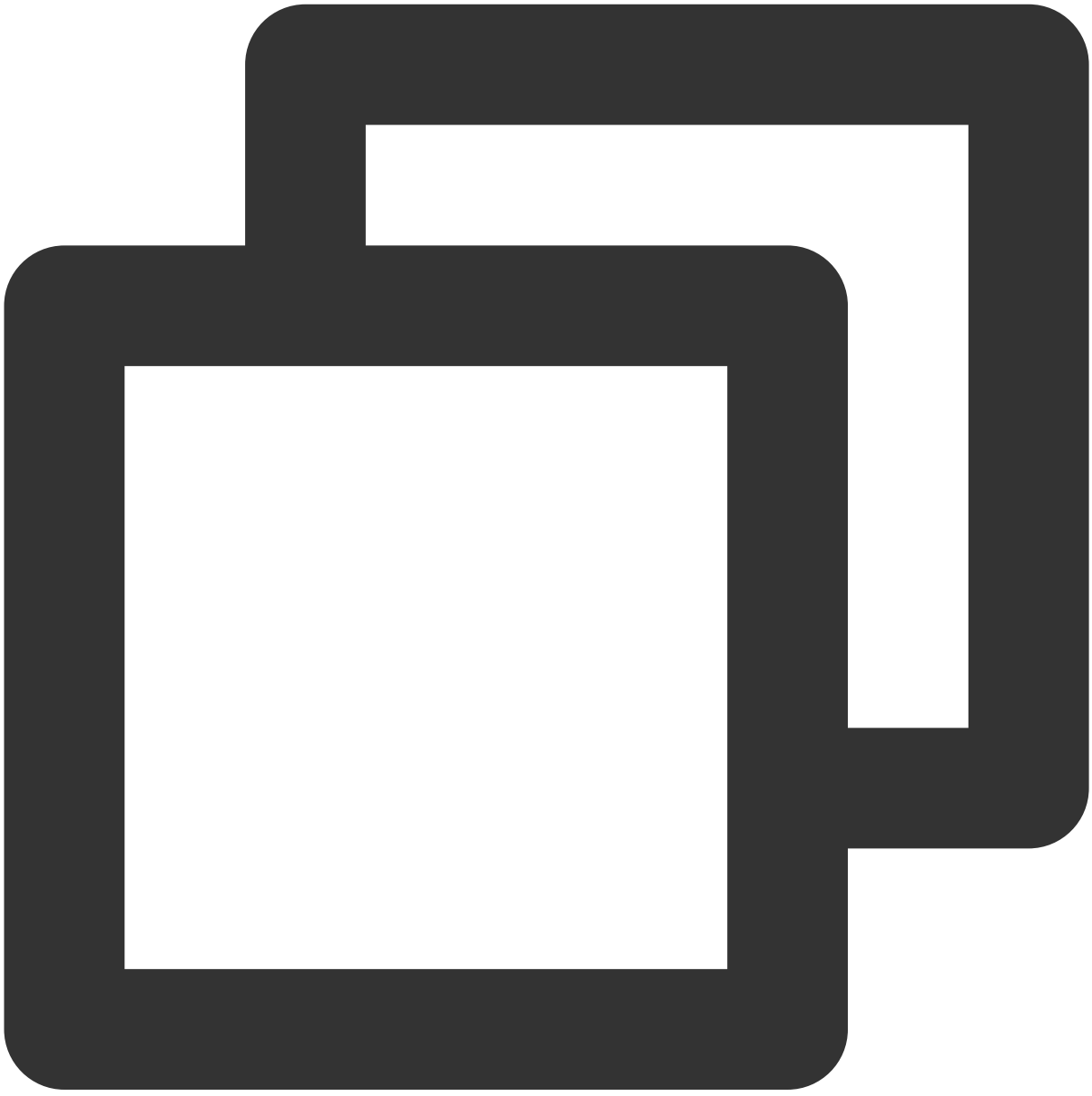
mysqld-exporterのデプロイの後、モニタリングキャプチャ設定を追加して、mysqld-exporterが公開するデータをキャプチャできるようにします。ServiceMonitor定義の例は次のとおりです（クラスターでサポートする必要があります）。



```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: mysqld-exporter
spec:
  endpoints:
    interval: 5s
    targetPort: 9104
  namespaceSelector:
    matchNames:
    - default
```

```
    selector:
      matchLabels:
        app: mysqld-exporter
```

Prometheusのネイティブ設定の例は次のとおりです。

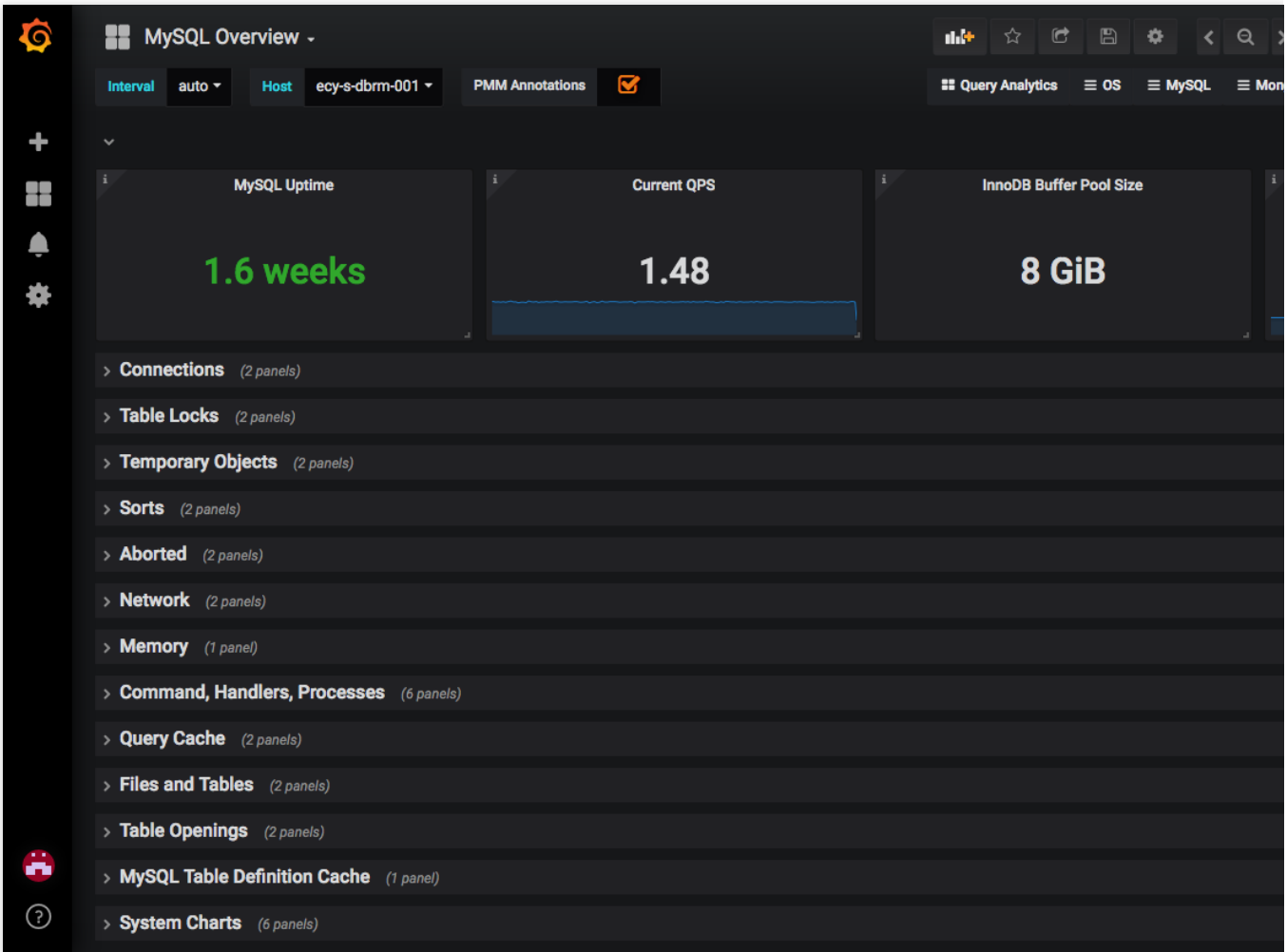


```
    - job_name: mysqld-exporter
      scrape_interval: 5s
      kubernetes_sd_configs:
      - role: endpoints
        namespaces:
          names:
```

```
        - default
    relabel_configs:
    - action: keep
      source_labels:
      - __meta_kubernetes_service_label_app_kubernetes_io_name
      regex: mysqld-exporter
    - action: keep
      source_labels:
      - __meta_kubernetes_endpoint_port_name
      regex: http
```

モニタリングパネルの追加

モニタリングキャプチャ設定で正常にデータをキャプチャできるようになったら、Grafanaにモニタリングパネルを追加して表示する必要があります。
MySQLやMariaDBの概要を見るだけの場合、パネルgrafana.comをインポートできます。下図に示すとおりです。



よりリッチなパネルにしたい場合は、perconaオープンソースパネルの `MySQL_` で始まるjsonファイルの内容をインポートするだけでOKです。

運用・保守
クラスター審査を使用したトラブルシューティング

最終更新日：：2023-04-28 15:30:11

シナリオ

人為的誤操作が発生し、アプリケーションにbugが発生し、悪意のあるプログラムがapiserverインターフェースを呼び出し、クラスターリソースが削除または変更されたとします。このときクラスター審査機能によってapiserverのインターフェース呼び出しを記録し、条件検索および分析審査ログに基づいて問題の原因を見つけることができます。ここではクラスター審査機能の具体的なユースケースおよび使用例をご紹介します。このドキュメントを参照してクラスター審査機能の使用を開始できます。

注意

ここではTencent Kubernetes Engine（TKE）クラスターにのみ適用されます。

前提条件

TKEコンソールにログインし、クラスター審査機能を有効化すること。詳細については、クラスター審査の有効化をご参照ください。

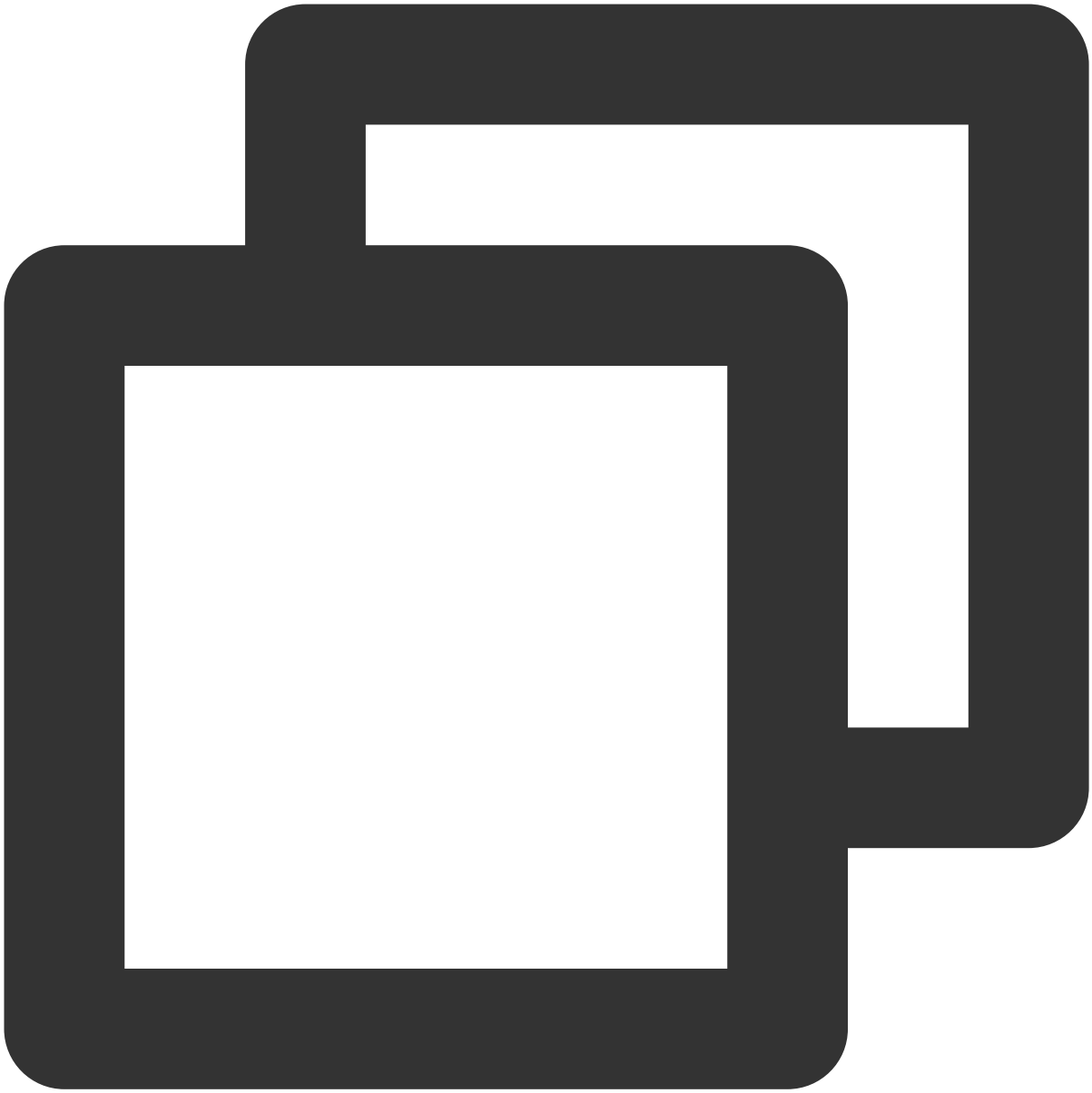
ユースケース

分析結果

1. CLSコンソールにログインし、左側のナビゲーションバーの**検索分析**をクリックします。
2. **検索分析**ページで、検索待ちのログセット、ログトピックおよび選択時間範囲を選択できます。
3. 分析ステートメントを入力した後、**検索分析**をクリックすると、分析結果を取得することができます。

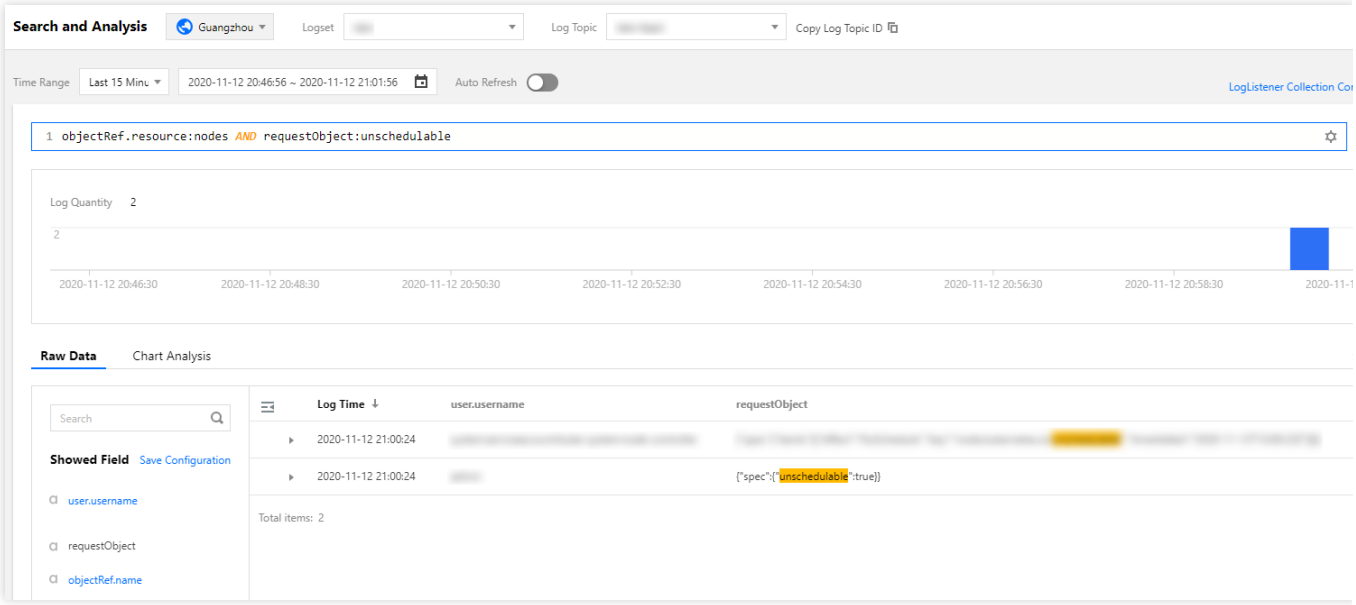
事例1：ノードをブロックしたオペレータを照会する

例えば、ノードをブロックしたオペレータを照会する必要がある場合、以下のコマンドを実行して検索を行うことができます。



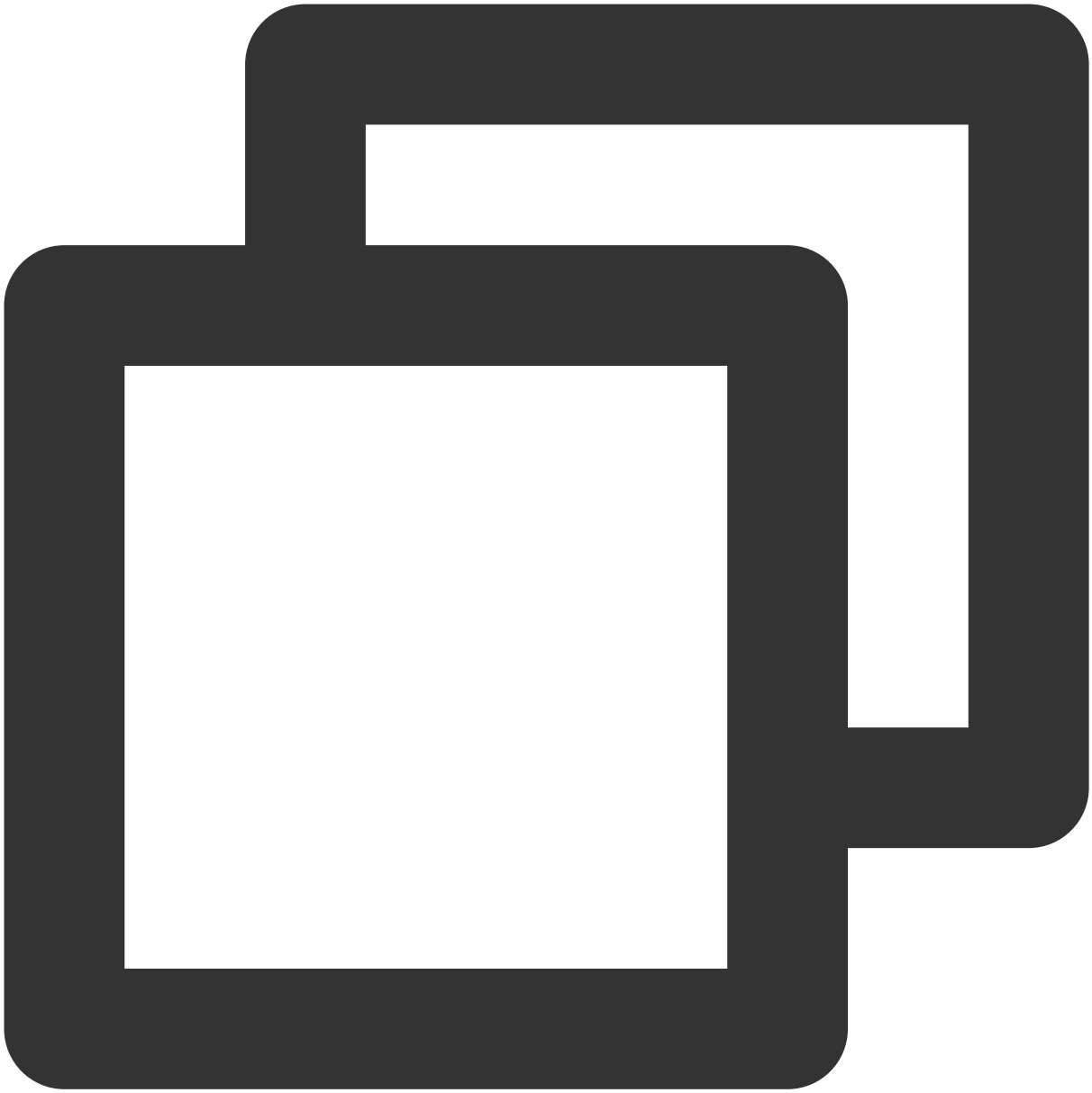
```
objectRef.resource:nodes AND requestObject:unschedulable
```

検索分析ページで、レイアウトの**デフォルト設定**を選択すると、照会結果が下図に示すとおりです。



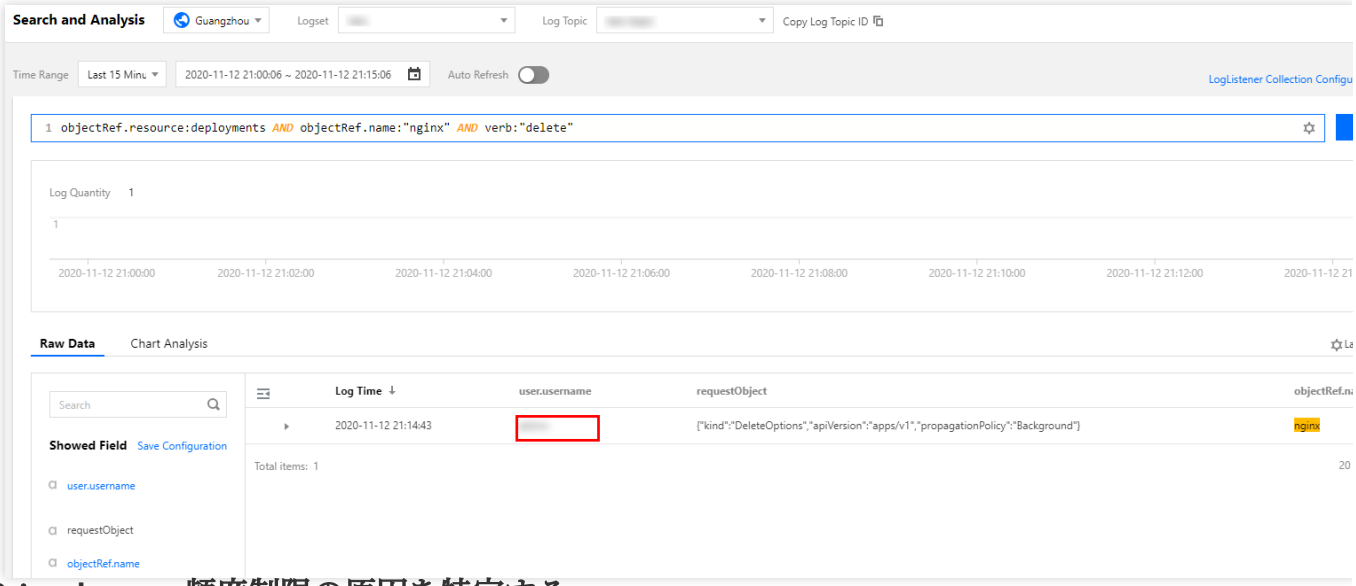
事例2：ワークロードを削除したオペレータを照会する

例えば、ワークロードを削除したオペレータを照会する必要がある場合、以下のコマンドを実行して検索を行うことができます。



```
objectRef.resource:deployments AND objectRef.name:"nginx" AND verb:"delete"
```

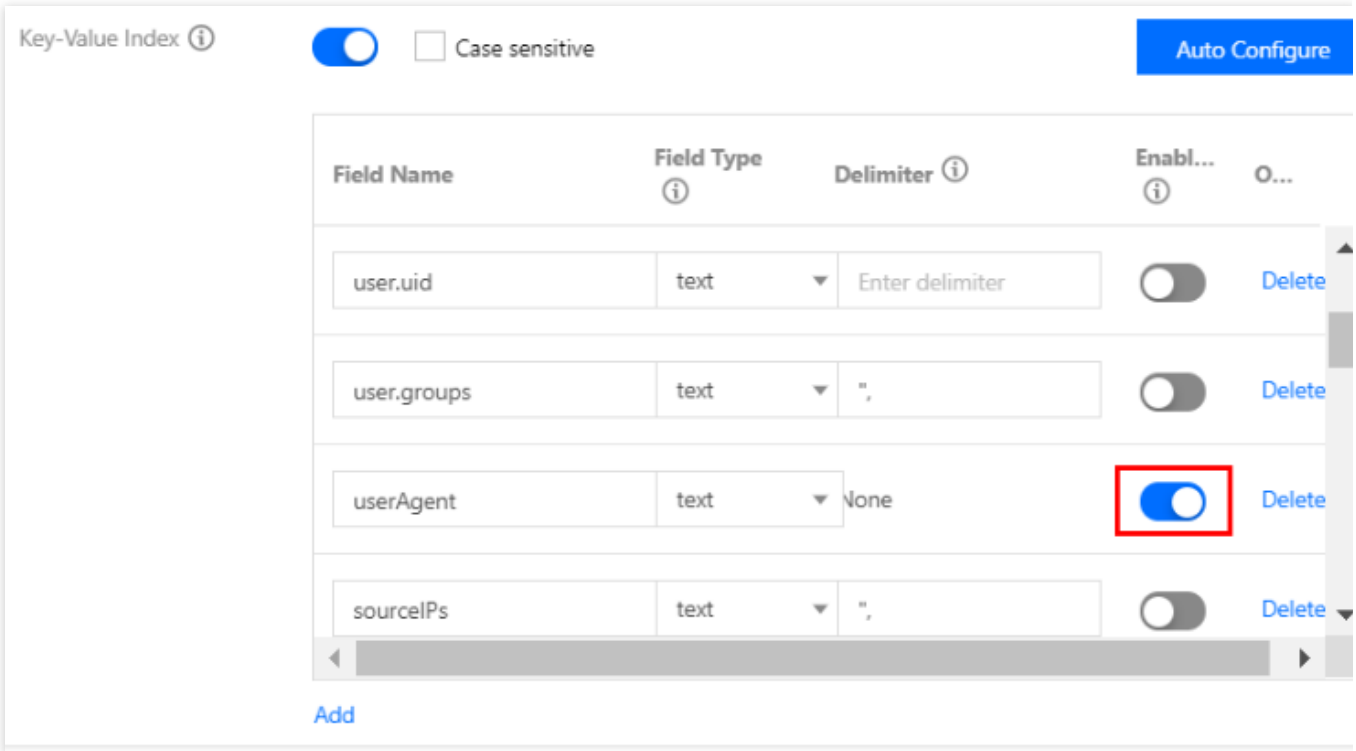
検索結果に基づいてこのサブアカウントの詳細情報を取得することができます。



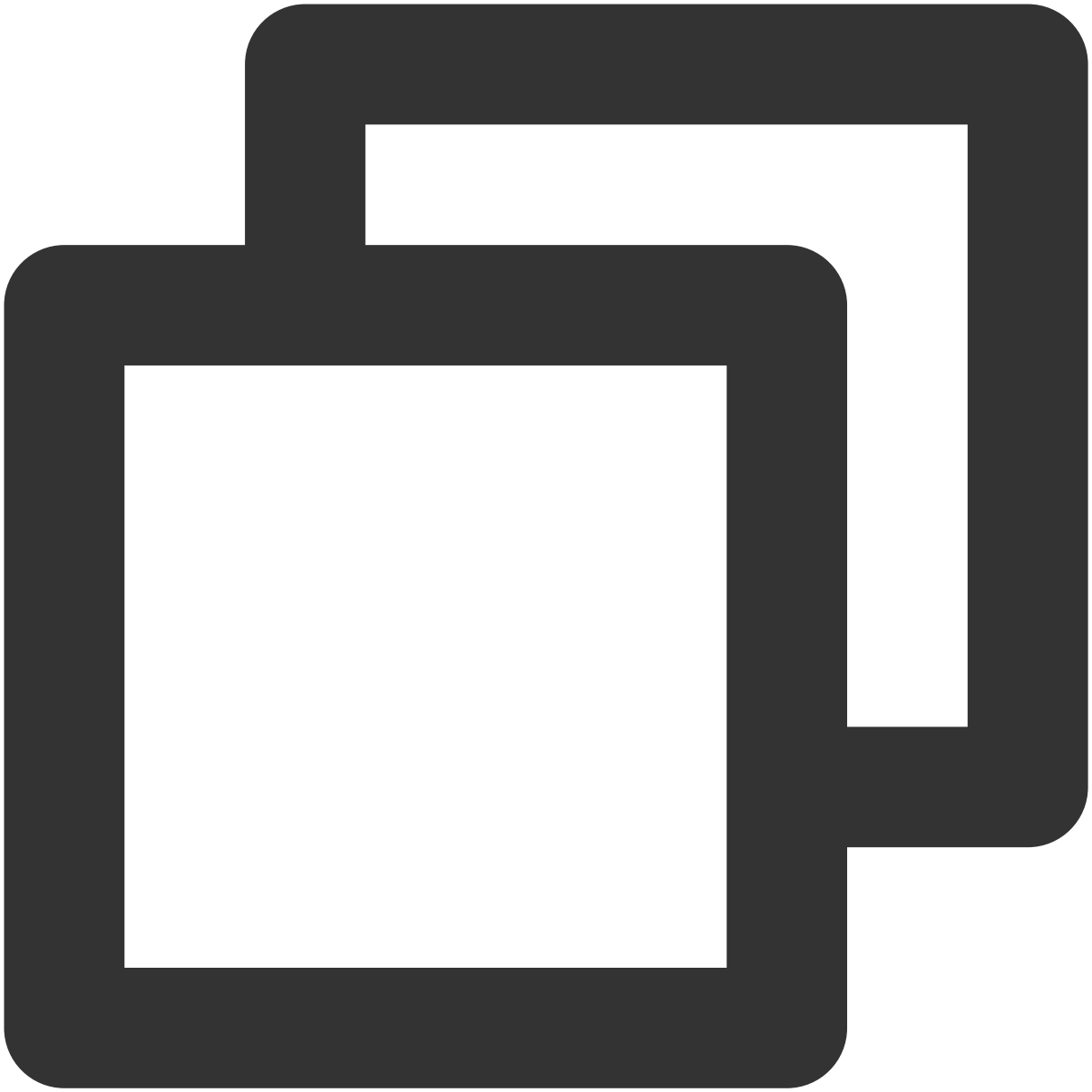
事例3：apiserver頻度制限の原因を特定する

悪意のあるプログラムまたはbugによりapiserverに対するリクエスト頻度が高すぎることによってapiserver/etcdに高い負荷がかかり、正常なリクエストに影響を与えることを回避します。apiserverはデフォルトリクエスト頻度の制限保護機能を搭載しています。頻度制限が発生した場合、審査によって大量のリクエストを送信するクライアントを見つけることができます。

1. 下図のように、userAgentによって統計をリクエストするクライアントを分析する必要がある場合、「キー値インデックス」ウィンドウでログトピックを変更し、userAgentフィールドを有効にして統計する必要があります。

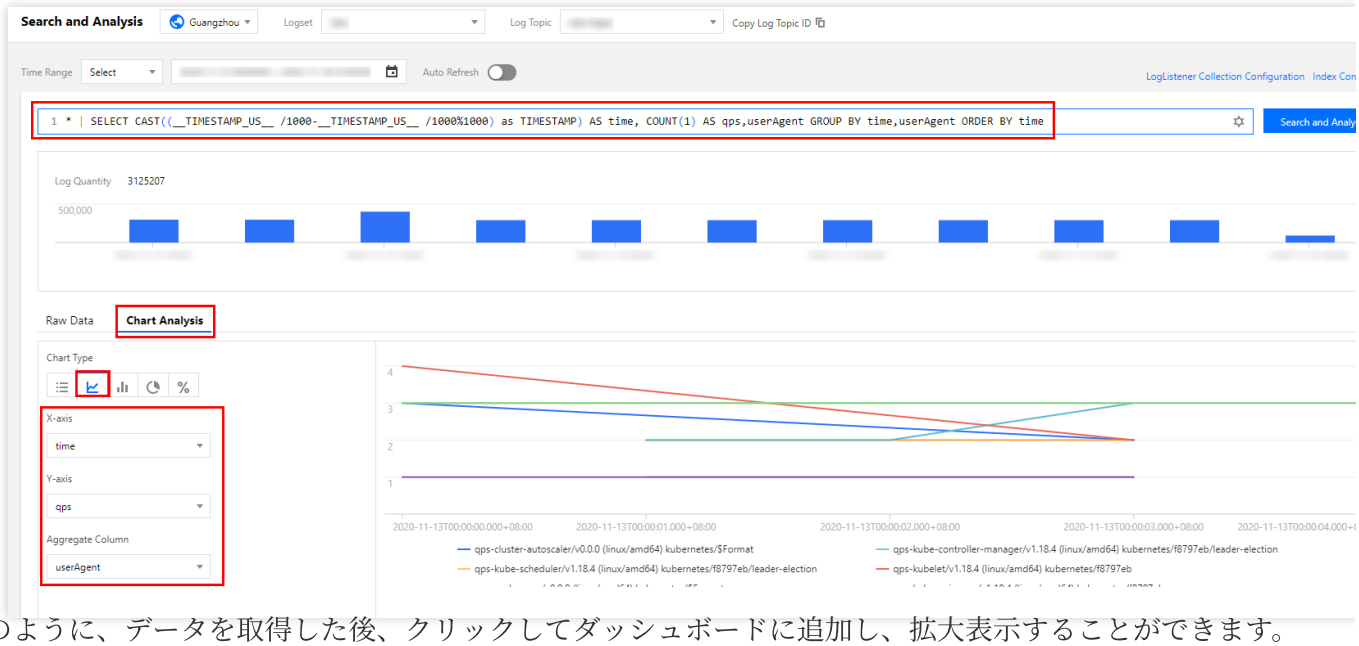


2. 以下のコマンドを実行して、各クライアントに対してapiserverをリクエストするQPSサイズを統計します。

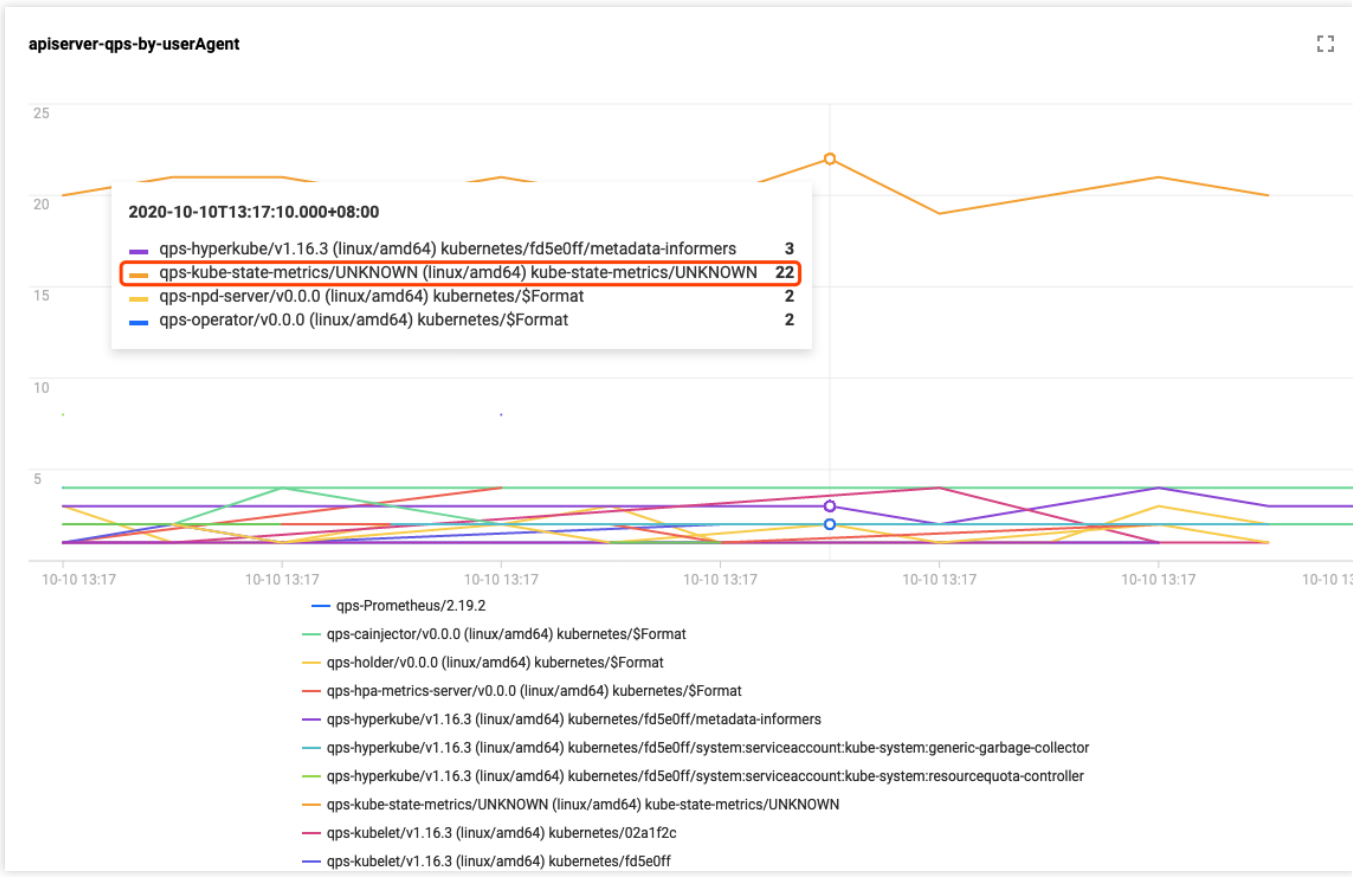


```
* | SELECT histogram( cast(__TIMESTAMP__ as timestamp),interval 1 minute) AS time,
```

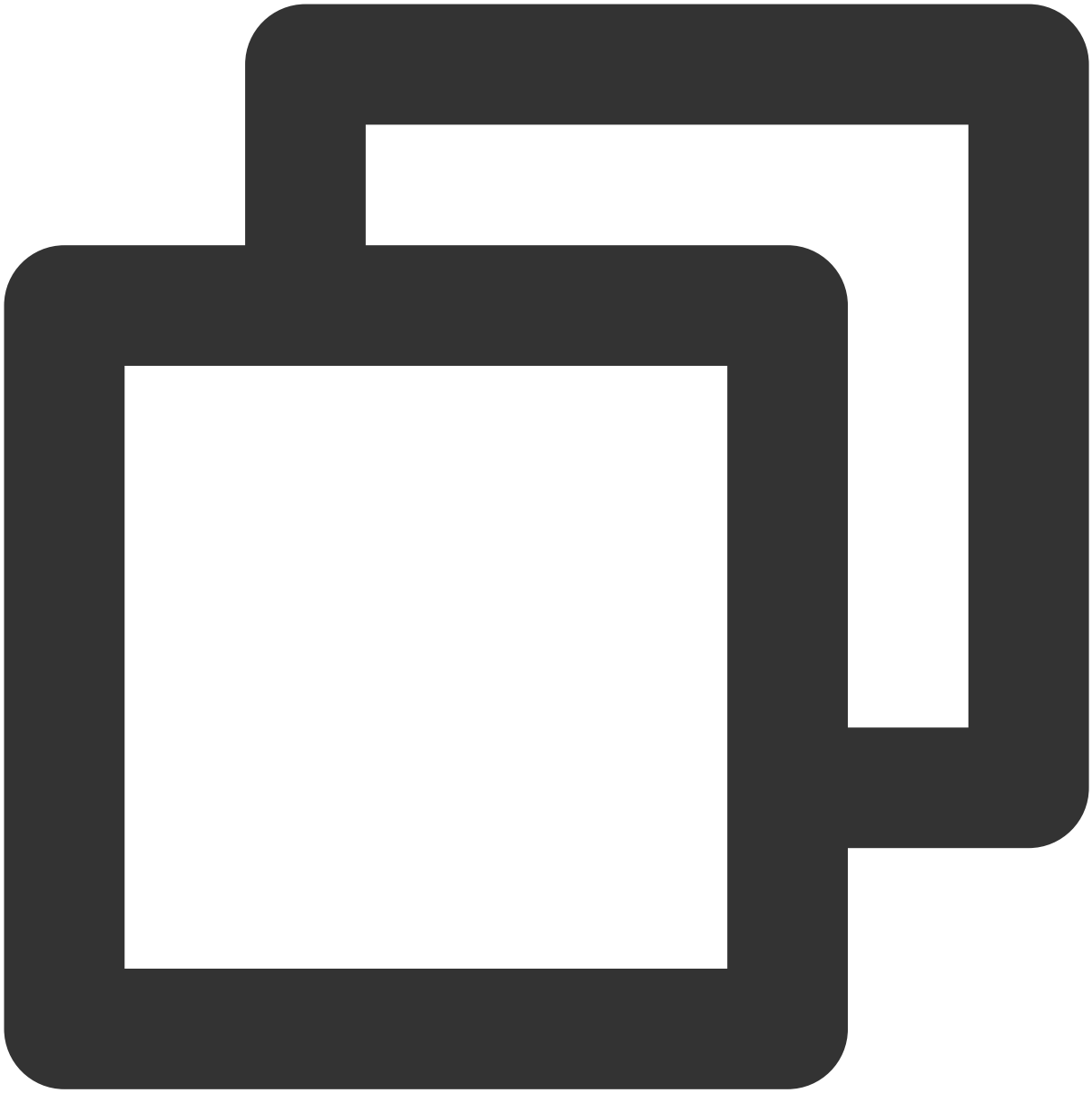
3. 下図のように、統計チャートに切り替え、シーケンス図を選択し、基本情報、座標軸などを設定することができます。



下図のように、データを取得した後、クリックしてダッシュボードに追加し、拡大表示することができます。

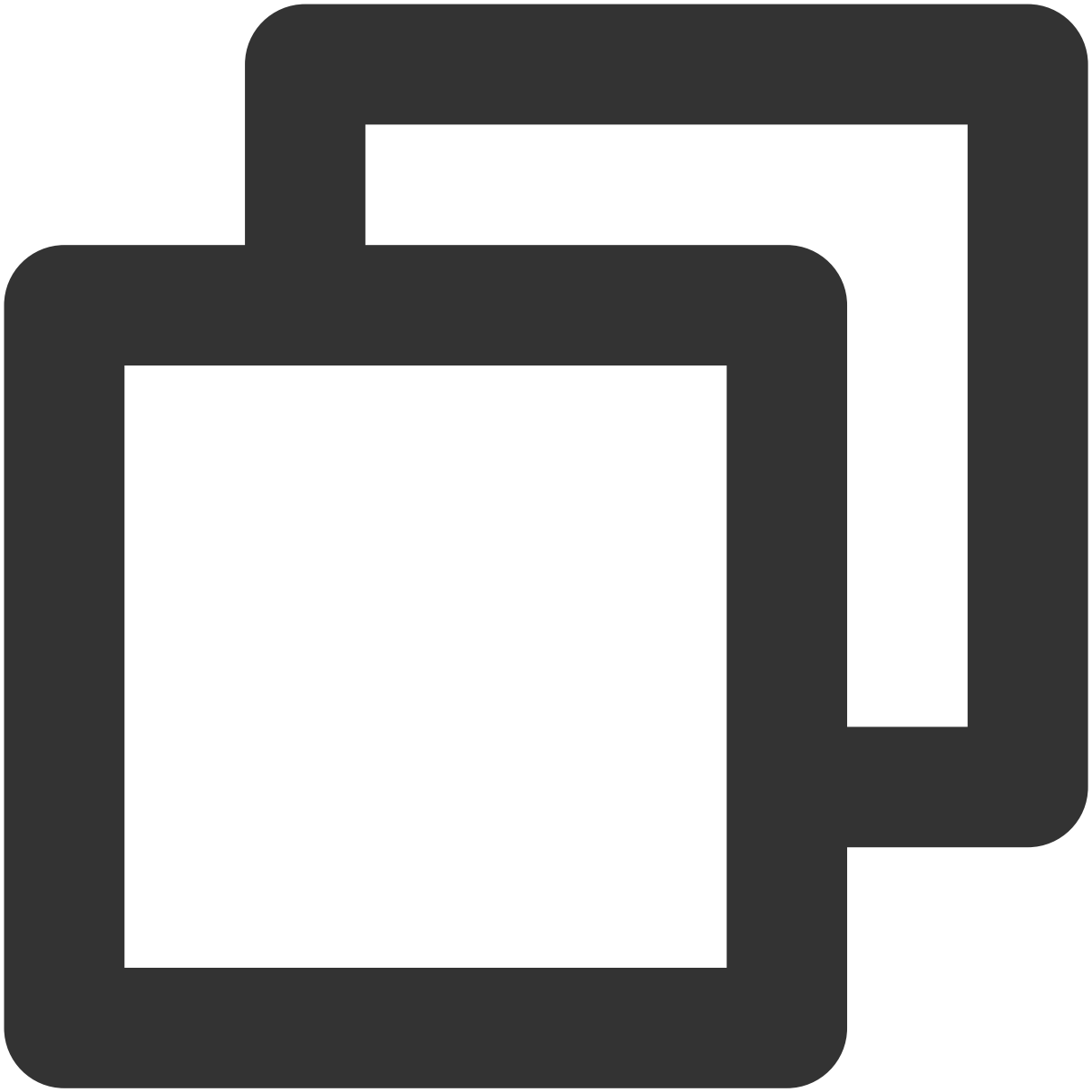


図からわかるように、kube-state-metricsクライアントのapiserverに対するリクエスト頻度はその他のクライアントよりはるかに高くなります。ログを確認するとわかるように、RBAC認証問題によってkube-state-metricsは停止することなくapiserverのリトライをリクエストするため、apiserverの頻度制限をトリガーします。ログは次に示すとおりです。



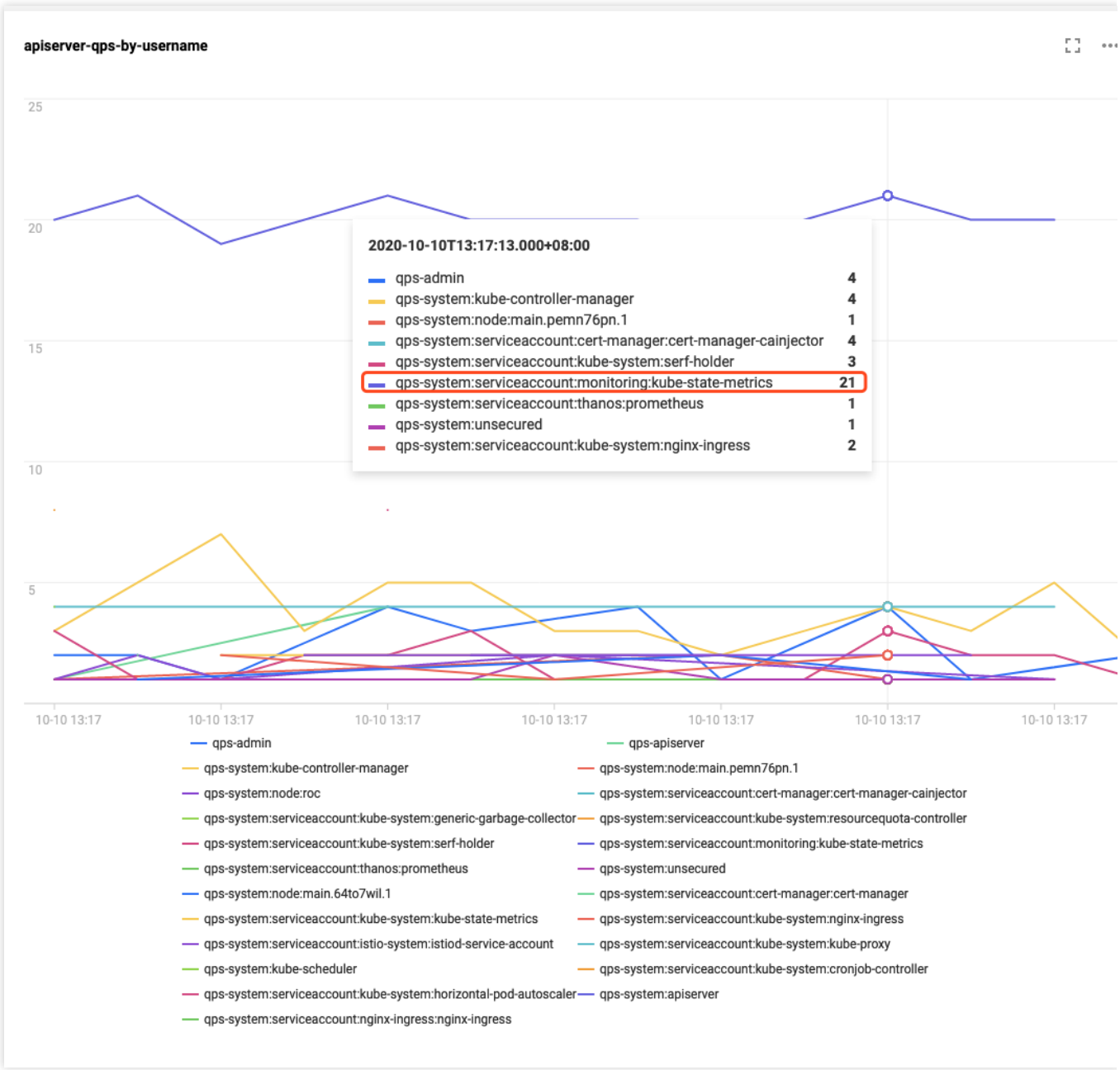
```
I1009 13:13:09.760767       1 request.go:538] Throttling request took 1.393921018s,
E1009 13:13:09.766106       1 reflector.go:156] pkg/mod/k8s.io/client-go@v0.0.0-201
```

同様に、その他のフィールドを使用することで統計するクライアントを区分する場合、必要に応じて柔軟にSQLを変更することができます。例えばuser.usernameを使用することで区分します。SQLステートメントは以下の例を参照することができます。



```
* | SELECT histogram( cast(__TIMESTAMP__ as timestamp),interval 1 minute)  AS time,
```

表示効果は下図に示すとおりです



関連ドキュメント

Tencent Kubernetes Engine（TKE）のクラスター審査の概要と基本操作については、クラスター審査をご参照ください。

クラスター審査のデータはログサービスに保存されています。CLSコンソール内で審査結果に検索と分析を行う必要がある場合、検索構文についてはログ検索構文とルールをご参照ください。

分析を行うにはCLSがサポートするSQLステートメントを提供する必要があります。詳細については、ログ分析の概要をご参照ください。

DevOps
TKEベースのJenkins外部ネットワークアーキテクチャアプリケーションのビルドとデプロイ
ステップ1：TKEクラスター側およびJenkins側の設定

最終更新日：：2023-04-28 11:08:19

TKEクラスター側の設定

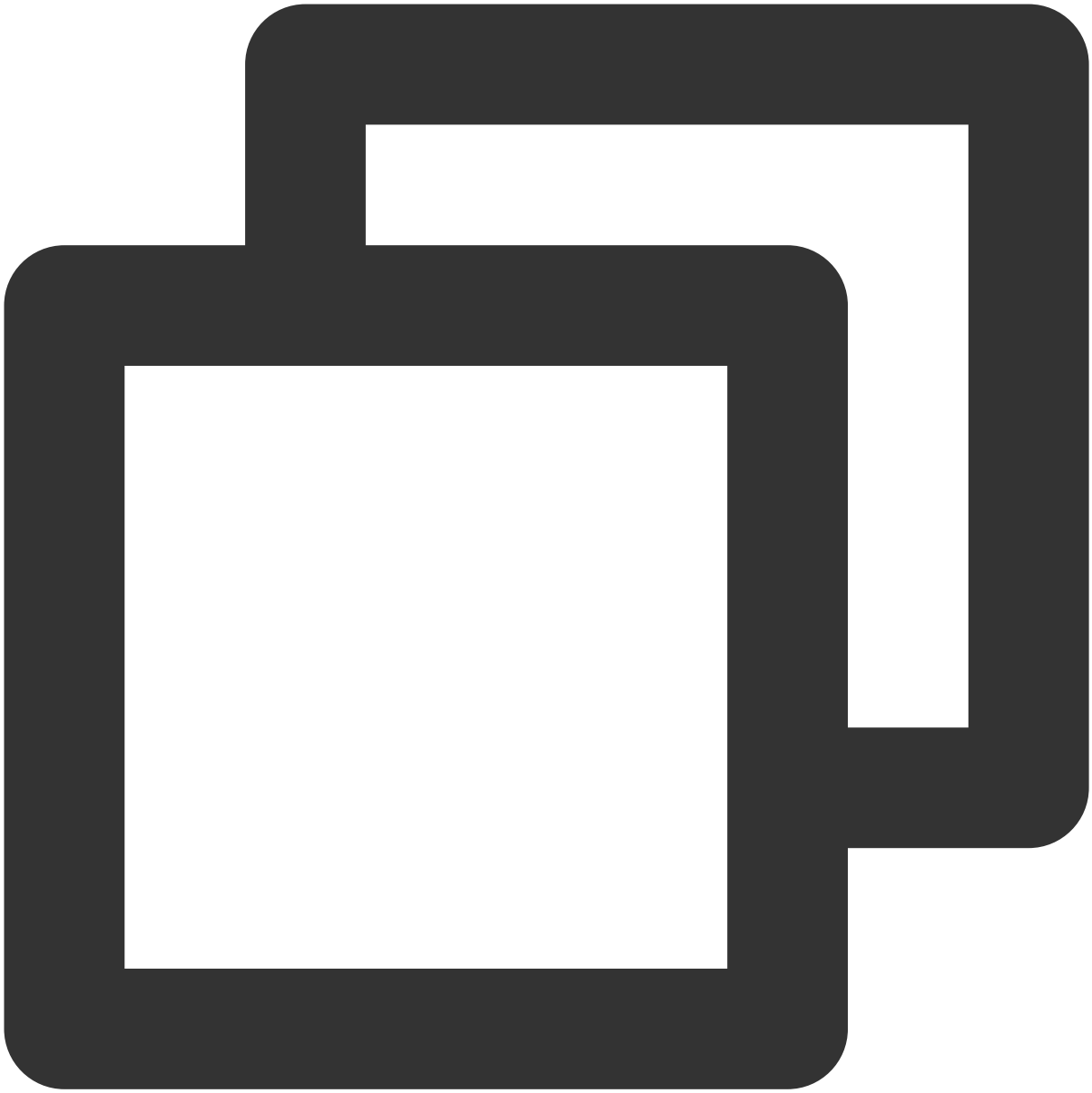
このステップでは、TKEにおけるRBAC承認のカスタマイズServiceAccountを介して、Jenkinsの設定に必要なクラスターアドレス、token、クラスターCA証明書の情報を取得する方法についてご説明します。

クラスター証明書の取得

説明

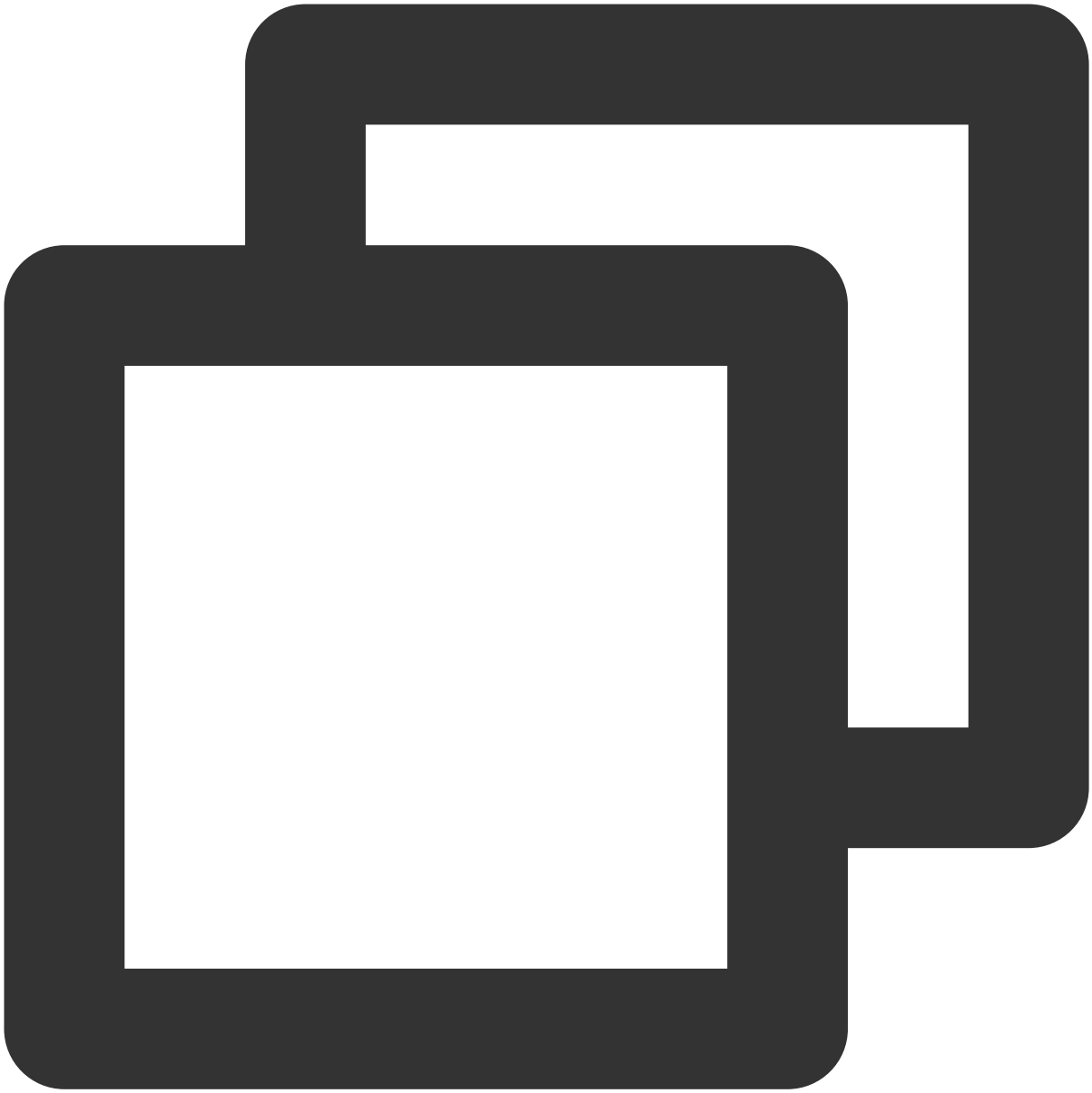
現在のクラスターは、プライベートネットワークアクセスを有効にする必要があります。詳細については、Serviceコンソール操作ガイドをご参照ください。

1. 以下のShellスクリプトを使用して、テスト用ネームスペースci、ServiceAccountタイプのテストユーザーjenkinsを作成し、クラスターアクセス証明(token)認証を取得します。



```
# テスト用ネームスペースciの作成
kubectl create namespace ci
# テスト用ServiceAccountアカウントの作成
kubectl create sa jenkins -n ci
# ServiceAccountアカウントが自動的に作成したSecret tokenの取得
kubectl get secret $(kubectl get sa jenkins -n ci -o jsonpath={.secrets[0].name}) -
```

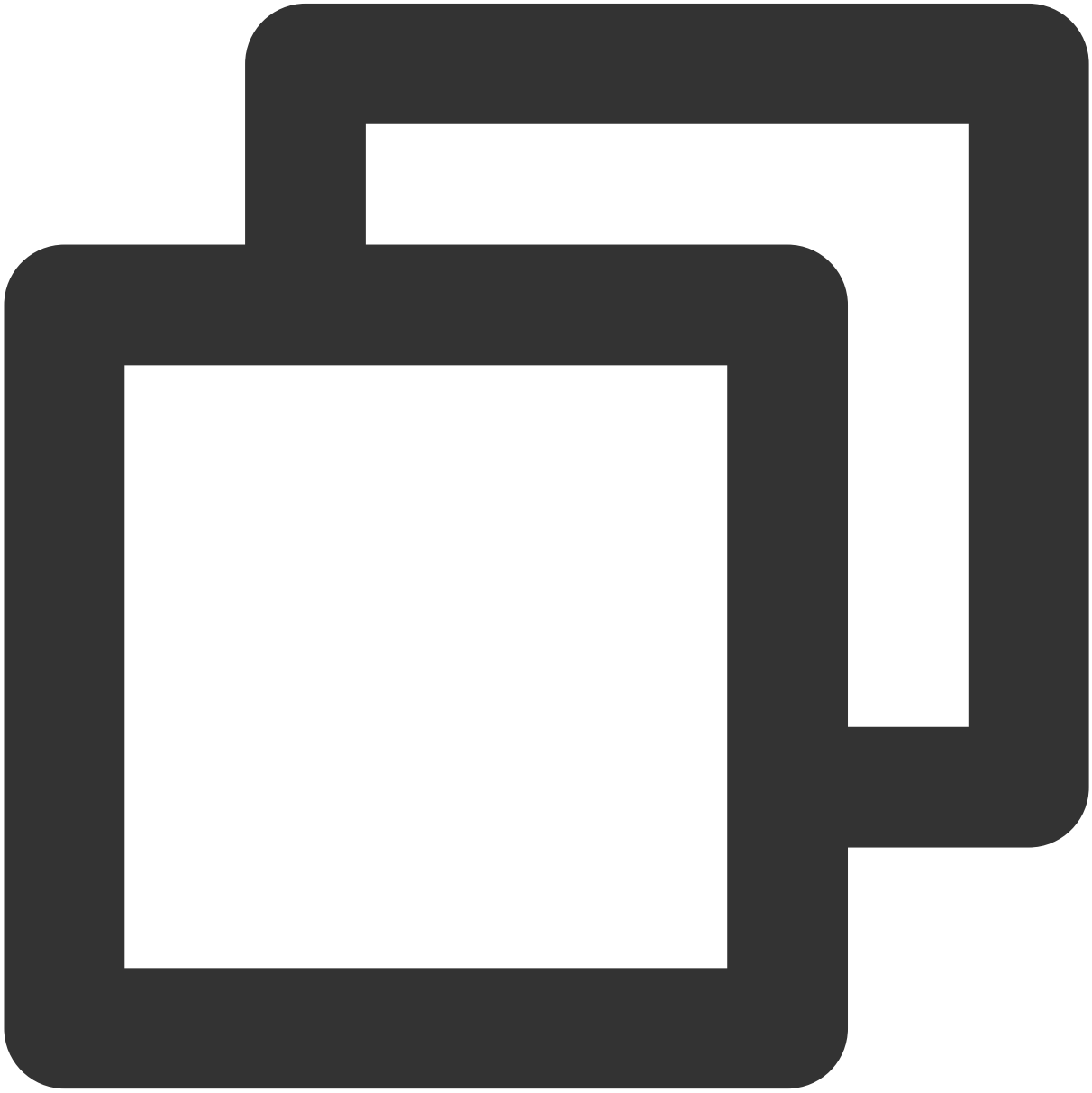
2. テスト用ネームスペースciで、Role権限オブジェクトリソースjenkins-role.yamlファイルを作成します。事例は次のとおりです。



```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: jenkins
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["create","delete","get","list","patch","update","watch"]
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["create","delete","get","list","patch","update","watch"]
```

```
- apiGroups: [""]
  resources: ["pods/log"]
  verbs: ["get","list","watch"]
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
```

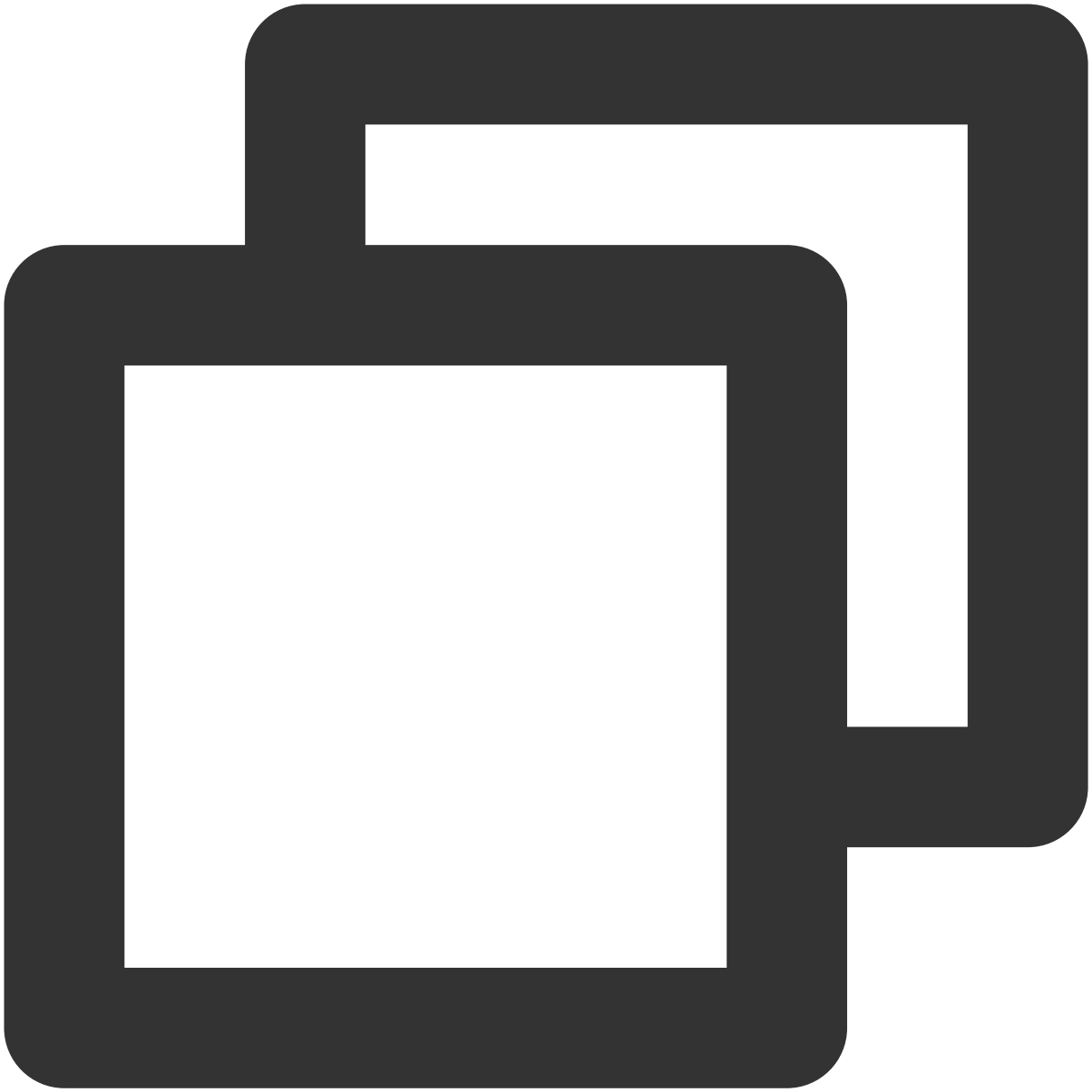
3. RoleBindingオブジェクトリソースjenkins-rolebinding.yamlファイルを作成します。以下の権限バインディングは、ServiceAccountタイプを追加するjenkinsユーザーが、ciネームスペースでjenkins（Roleタイプ）の権限を有することを示しています。事例は次のとおりです。



```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: jenkins
  namespace: ci
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: jenkins
subjects:
- kind: ServiceAccount
  name: jenkins
```

クラスターCA証明書の取得

1. 標準ログイン方式を使用してLinuxインスタンスにログイン（推奨）を参照して、ターゲットクラスターのノードにログインします。
2. 以下のコマンドを実行し、クラスターCA証明書を確認します。

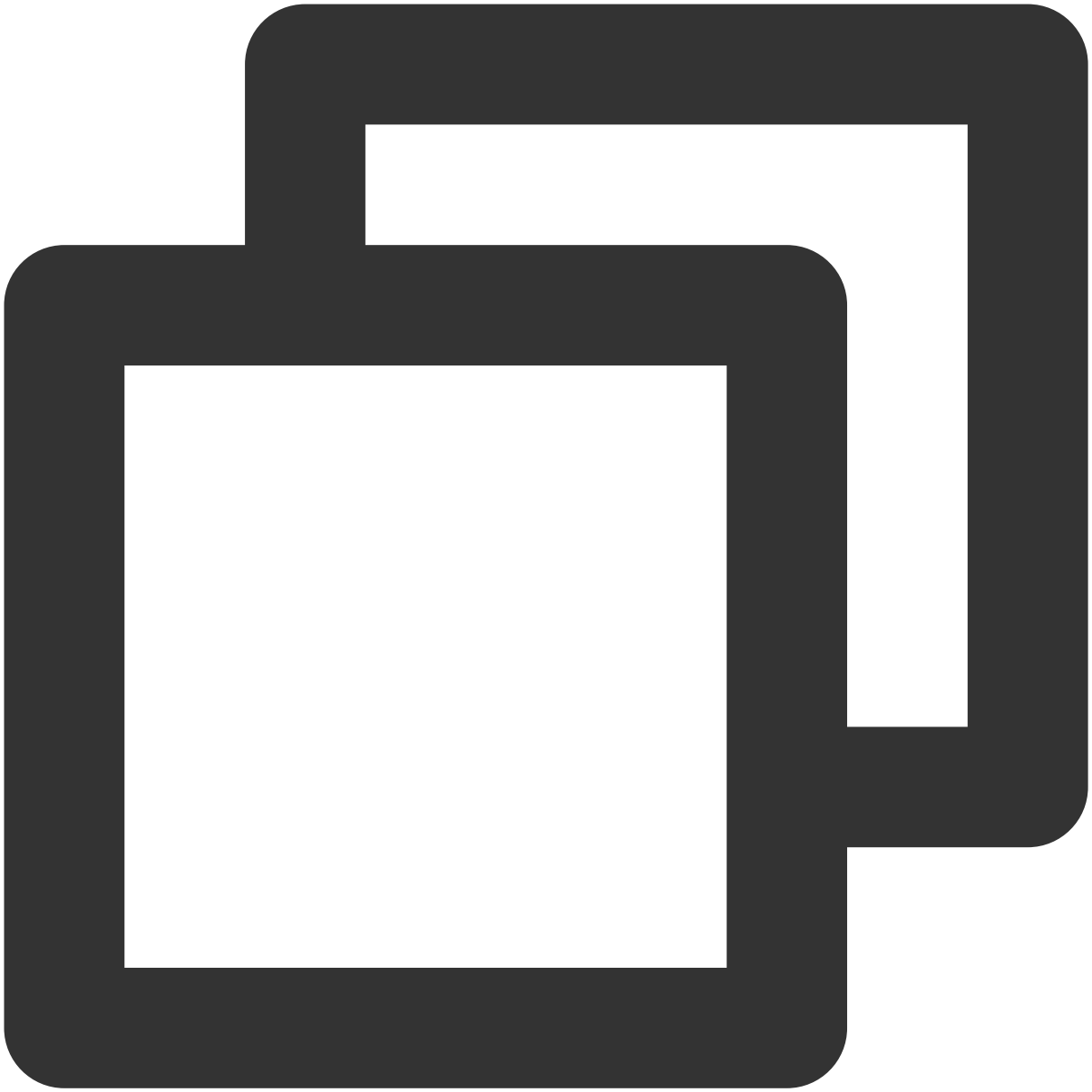


```
cat /etc/kubernetes/cluster-ca.crt
```

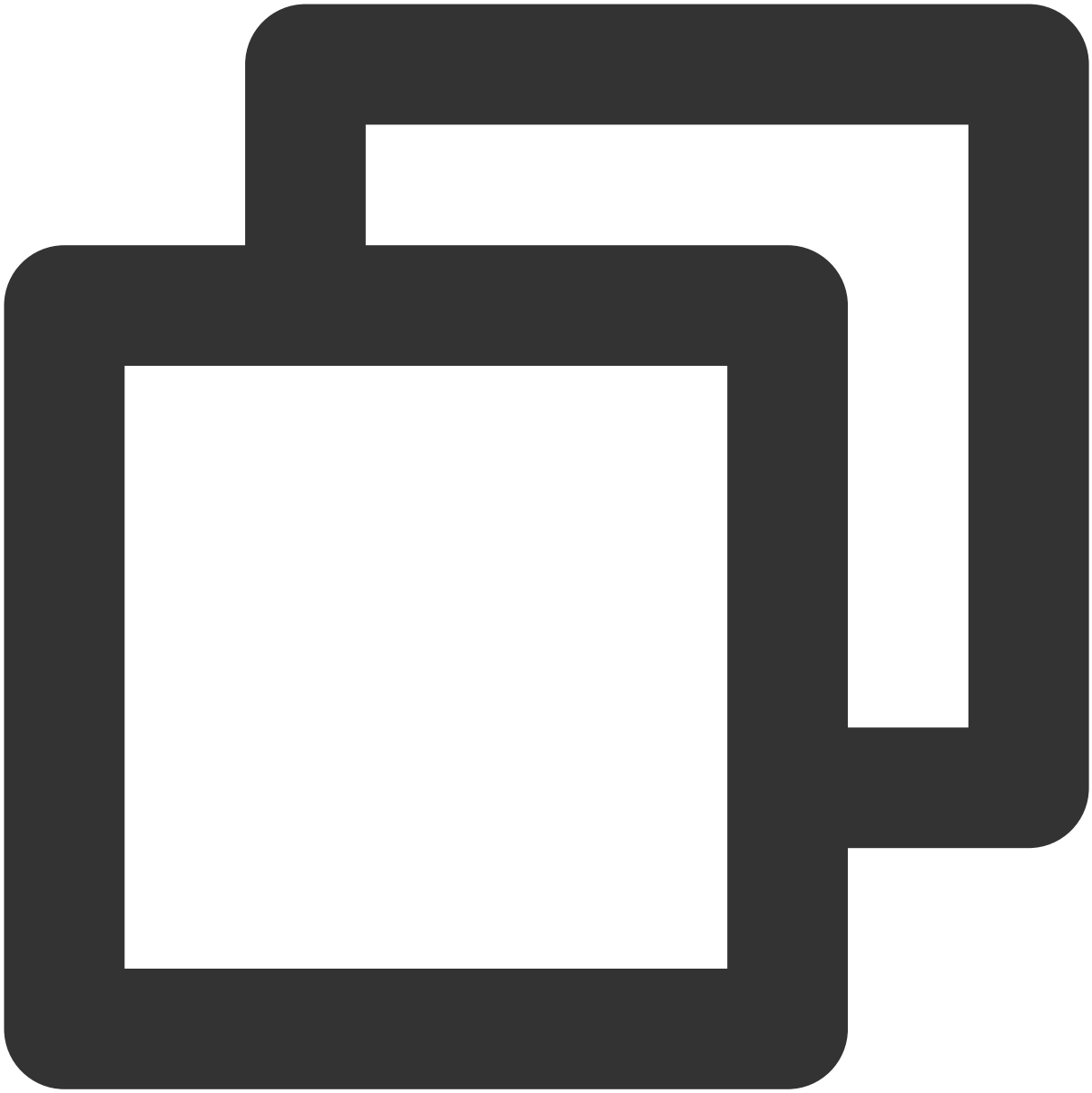
3. 照会によって得た証明書情報を記録、保存してください。下図に示すとおりです。

docker.sockの承認

TKEクラスターの各nodeは、そのノードシステムに `docker.sock` ファイルがあります。slave podが `docker build` を実行すると、このファイルに接続します。その前に、各ノードに個別にログインし、以下のコマンドを順番に実行して `docker build` を承認する必要があります。



```
chmod 666 /var/run/docker.sock
```



```
ls -l /var/run/docker.sock
```

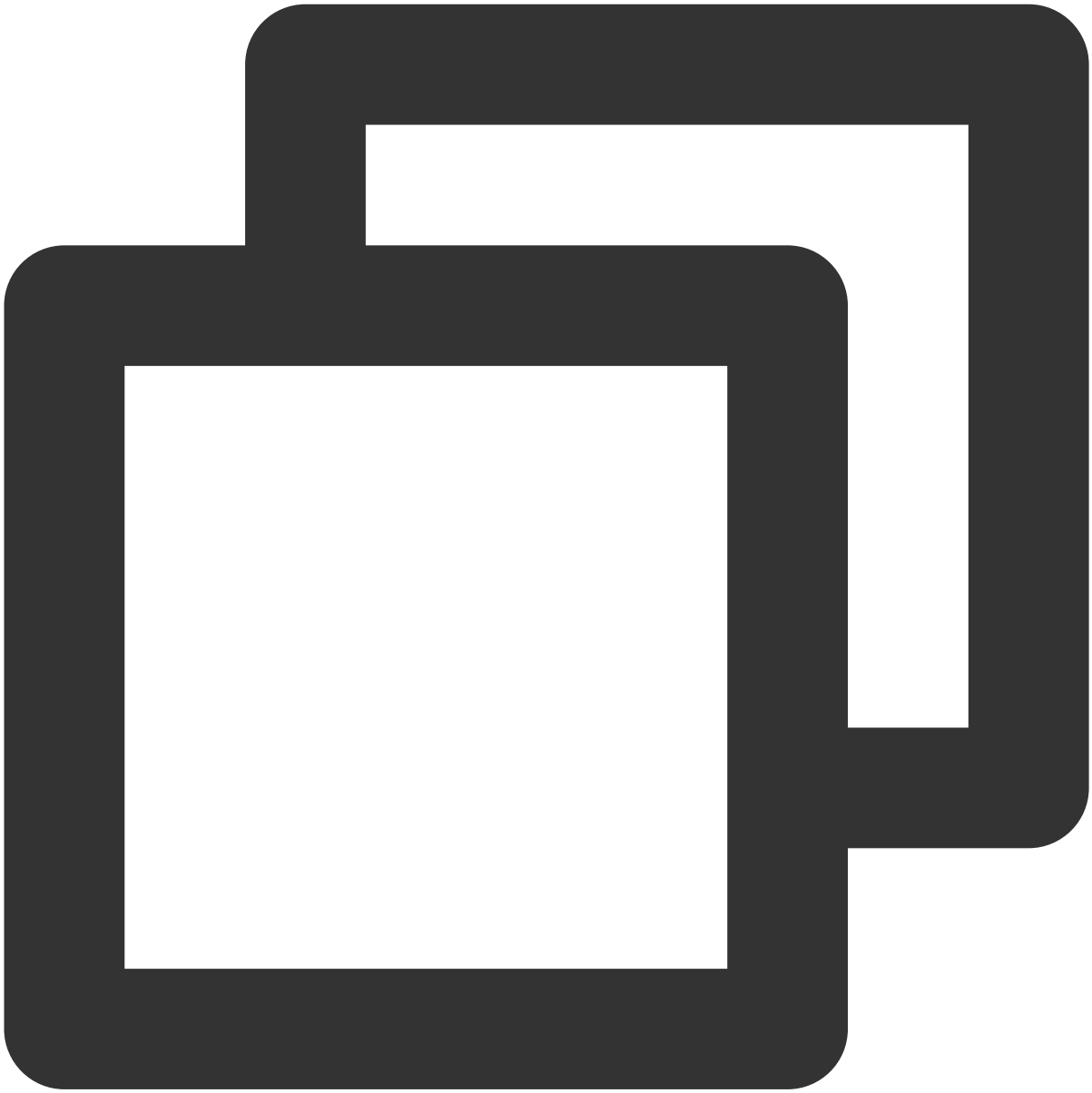
Jenkins側の設定

説明

Jenkinsのバージョンによって UIの使い方に違いがあります。ビジネスニーズに応じて選択できます。

TKEプライベートネットワークアドレスの追加

1. 標準ログイン方式を使用してLinuxインスタンスにログイン（推奨）を参照して、Jenkins Masterのノードにログインします。
2. 以下のコマンドを実行し、ドメイン名へのアクセスを設定します。

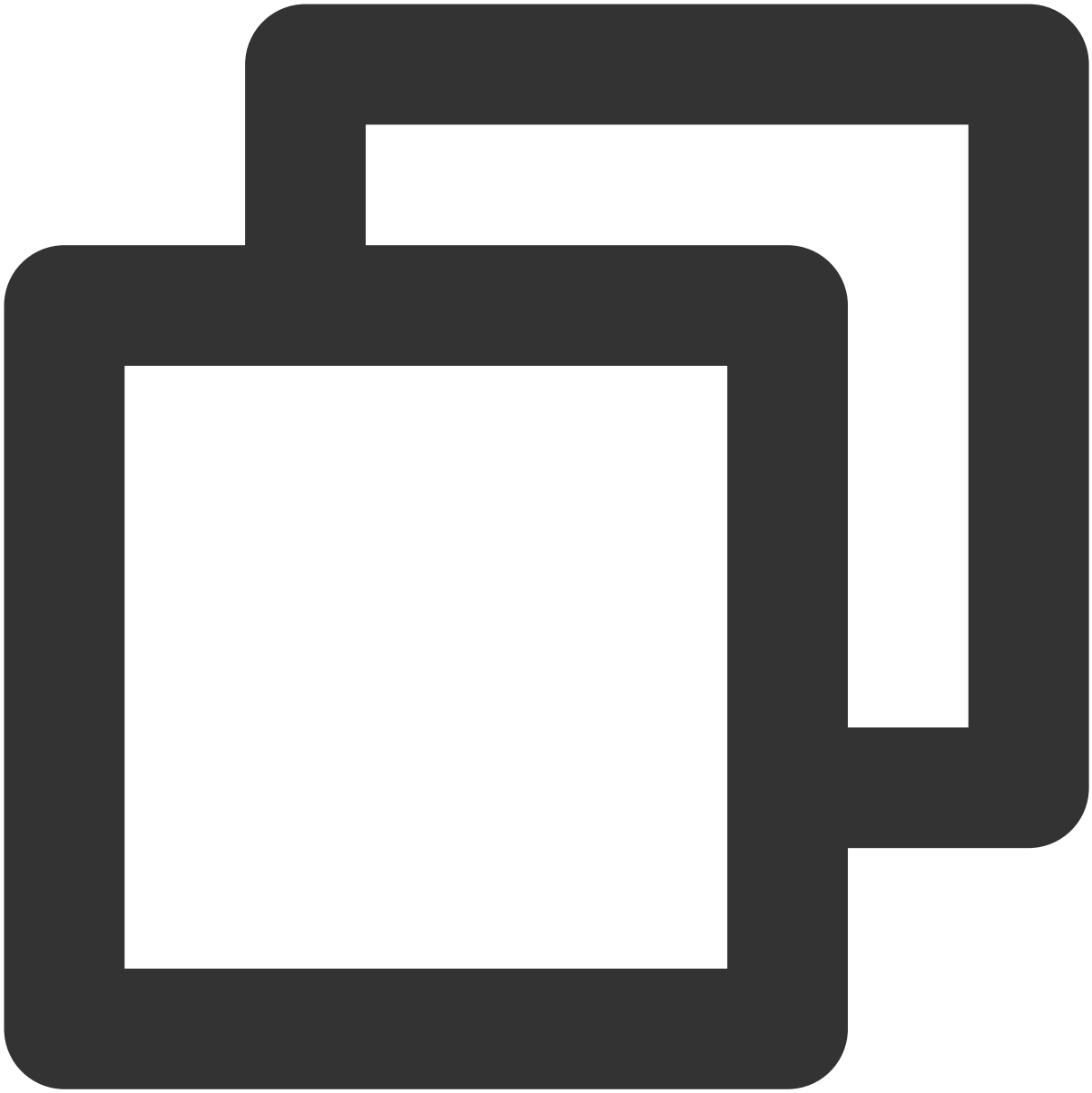


```
sudo sed -i '$a 10.x.x.x cls-ixxxelli.ccs.tencent-cloud.com' /etc/hosts
```

説明

このコマンドは、クラスターがプライベートネットワークアクセスを有効にした後、クラスターの基本情報ページの「クラスターAPIServer」から取得できます。詳細については、クラスター証明書の取得をご参照ください。

3. 以下のコマンドを実行して、設定が成功したかどうかを確認します。



```
cat /etc/hosts
```

下図のようになれば、設定は成功しています。

```
[root@VM_0_7_centos ~]# sudo sed -i '$a 10 .ccs.tencent-clou
[root@VM_0_7_centos ~]# cat /etc/hosts
127.0.0.1 VM_0_7_centos VM_0_7_centos
127.0.0.1 localhost.localdomain localhost
127.0.0.1 localhost4.localdomain4 localhost4

::1 VM_0_7_centos VM_0_7_centos
::1 localhost.localdomain localhost
::1 localhost6.localdomain6 localhost6

10. .ccs.tencent-cloud.com
```

Jenkinsによる必須プラグインのインストール

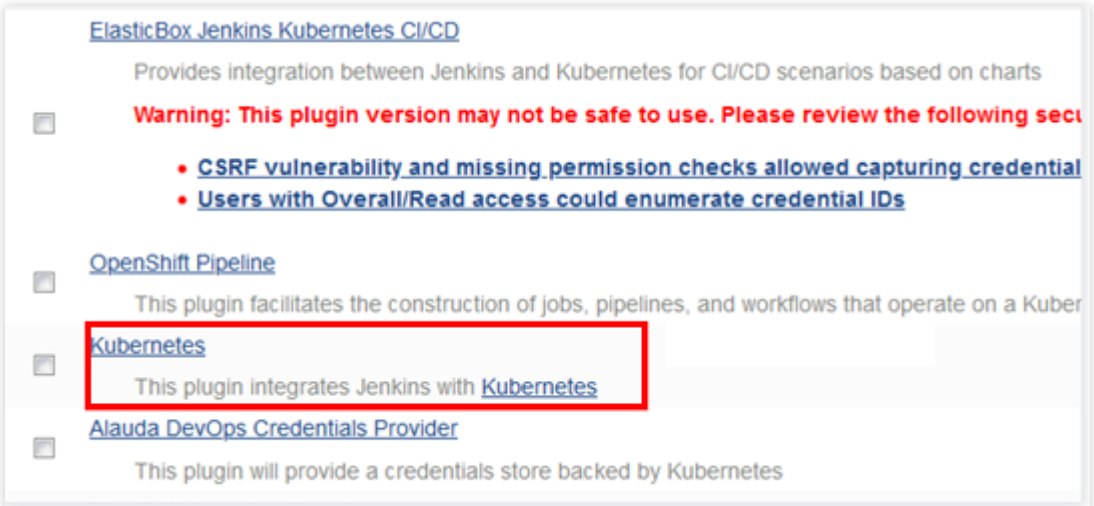
1. Jenkinsのバックエンドにログインし、左側ナビゲーションバーで**システム管理**を選択します。
2. 開いた「Jenkins管理]パネルで、**プラグイン管理**をクリックします。
3. プラグイン管理ページで**選択可能なプラグイン**を選択し、Locale、Kubernetes、Git ParameterとExtended Choice Parameterにチェックを入れます。

Locale：中国語のプラグインです。このプラグインをインストールすると、Jenkinsインターフェースをデフォルトで中国語版にできます。

Kubernetes：Kubernetes-pluginプラグイン。

Git Parameterと**Extended Choice Parameter**：パッケージのビルド時にパラメータを渡すために使用します。

Kubernetesプラグインを例とした場合、下図のようになります。



4. 上記のプラグインにチェックを入れ、**ダイレクトインストール**をクリックし、Jenkinsを再起動すれば完了です。

jnlpポートの開放

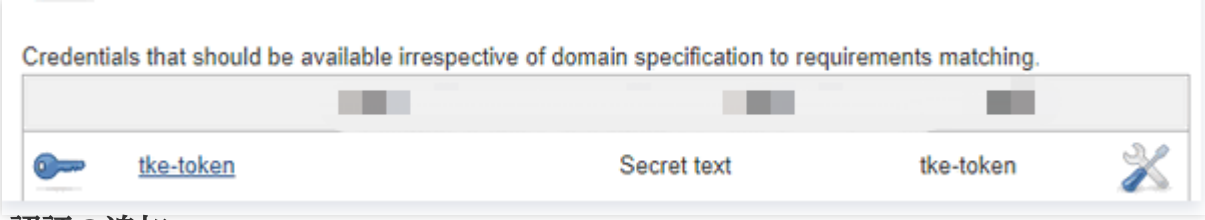
1. Jenkinsのバックエンドにログインし、左側ナビゲーションバーで**システム管理**を選択します。
2. 開いた「Jenkins管理」パネルで、**グローバルセキュリティコンフィグレーション**をクリックします。
3. ローバルセキュリティコンフィグレーションページで、インバウンドプロキシのTCPポートを「ポート50000を指定する」に設定します。
4. その他の設定項目はデフォルトのままにして、ページ下部の**保存**をクリックします。

TKEクラスターtokenの追加

1. Jenkinsのバックエンドにログインし、左側ナビゲーションバーで**認証情報 > システム**を選択します。
2. 開いた「システム」パネルで、**グローバル認証情報(unrestricted)**を選択します。
3. 「グローバル認証情報(unrestricted)ページで、左側メニューバーの**認証情報の追加**をクリックし、以下のプロンプトに従って基本的な認証情報を設定します。

タイプ：**Secret text**を選択します。
範囲：デフォルトでは**グローバル(Jenkins、nodes、items、all child items、etc)**となります。
Secret：クラスター証明書の取得のステップで取得したServiceAccount jenkinsの**Token**を入力します。
ID：デフォルトでは入力されていません。
説明：この認証情報に関する情報を入力します。この内容は、認証情報名と説明情報として表示されます。ここでは例として、 `tke-token` とします。

4. **OK**をクリックするとすぐ追加されます。追加に成功すると、この認証情報が認証情報リストに表示されます。下図に示すとおりです。



gitlab認証の追加

1. 「グローバル認証情報(unrestricted)ページで、左側メニューバーの**認証情報の追加**をクリックし、以下のプロンプトに従って基本的な認証情報を設定します。

タイプ：**Username with password**を選択します。
範囲：デフォルトでは**グローバル(Jenkins、nodes、items、all child items、etc)**となります。
ユーザー名：gitlabユーザー名です。
パスワード：gitlabログインパスワードです。
ID：デフォルトでは入力されていません。
説明：この認証情報に関する情報を入力します。この内容は、認証情報名と説明情報として表示されます。ここでは例として、 `gitlab-password` とします。

2. **OK**をクリックすると、追加が成功します。

slave podテンプレートの設定

1. Jenkinsのバックエンドにログインし、左側ナビゲーションバーで**システム管理**を選択します。
2. 開いた「Jenkins管理」パネルで、**システム設定**をクリックします。
3. 「システム設定」パネルの下部、「クラウド」モジュールの下にある**新しいクラウドの追加 > Kubernetes**を選択します。
4. **Kubernetes Cloud details...**をクリックして、以下のKubernetesの基本情報を設定します。

主要パラメータの設定は次のとおりです。その他のオプションはデフォルトを維持してください。

名前：ご自身で定義します。ここでは例として、 `kubernetes` とします。
Kubernetesアドレス：TKEクラスターアクセスアドレス、クラスター証明書の取得手順を参照して取得できます。
Kubernetesサービス証明書Key：クラスターCA証明書、クラスターCA証明書の取得手順を参照して取得できます。
認証情報：TKEクラスターtokenの追加のステップで作成した認証情報 `tke-token` を選択して**接続テスト**をクリックします。接続が成功すると、Connection test succeessfulというメッセージが表示されます。
Jenkinsアドレス：Jenkinsプライベートネットワークアドレスを入力します。
例： `http://10.x.x.x:8080` 。
5. **Pod Templates** > **Podテンプレートの追加** > **Pod Templates details...**を選択してPodテンプレートの基本情報を設定します。
主要パラメータの情報は次のとおりです。その他のオプションはデフォルトを維持してください。
名前：ご自身で定義します。ここでは例として、 `jnlp-agent` とします。
タグリスト：タグ名を定義します。作成するときにこのタグに基づいてPodを選択できます。ここでは例として、 `jnlp-agent` とします。
使用法：**可能な限りこのノードを使用する**を選択します。
6. 「コンテナリスト」で**コンテナの追加** > **Container Template**を選択して以下のコンテナ関連情報を設定します。
名前：ご自身でコンテナ名を定義します。ここでは例として、 `jnlp-agent` とします。
Dockerイメージ：イメージアドレス `jenkins/jnlp-slave:alpine` を入力します。
作業ディレクトリ：デフォルト設定のままにし、shellスクリプトでパッケージのビルドに使用される作業ディレクトリを記録しておいてください。
残りのオプションは初期設定のままにしておいてください。
7. 「ボリューム」で以下の手順に従ってボリュームを追加し、slave podにdockerコマンドを設定します。
7.1 **ボリュームの追加** > **Host Path Volume**を選択してホストとマウントパスに `/usr/bin/docker` を入力します。
7.2 **ボリュームの追加** > **Host Path Volume**を選択してホストとマウントパスに `/var/run/docker.sock` を入力します。
7.3 ページ下部の**保存**をクリックすれば、slave podテンプレートの設定は完了です。

次のステップ

ステップ2：Slave pod作成の設定に移動して新しいタスクを作成し、タスクパラメータを設定してください。

ステップ2：Slave podのビルド設定

最終更新日：：2023-04-28 11:08:19

このステップでは、Jekinsで新しいタスクを作成し、タスクのパラメータを設定して、slave podをビルドする方法についてご説明します。

説明

Jenkinsのバージョンによって UIの使い方に違いがあります。ビジネスニーズに応じて選択できます。

タスクの作成

1. Jenkinsコンソールにログインし、**タスクの新規作成**または**新しいタスクの作成**をクリックします。
2. タスクの新規作成ページで、タスクの基本情報を設定します。

タスク名を入力：ご自身で定義します。ここでは例として、 `test` とします。

タイプ：**フリースタイルのソフトウェアプロジェクトのビルド**を選択します。

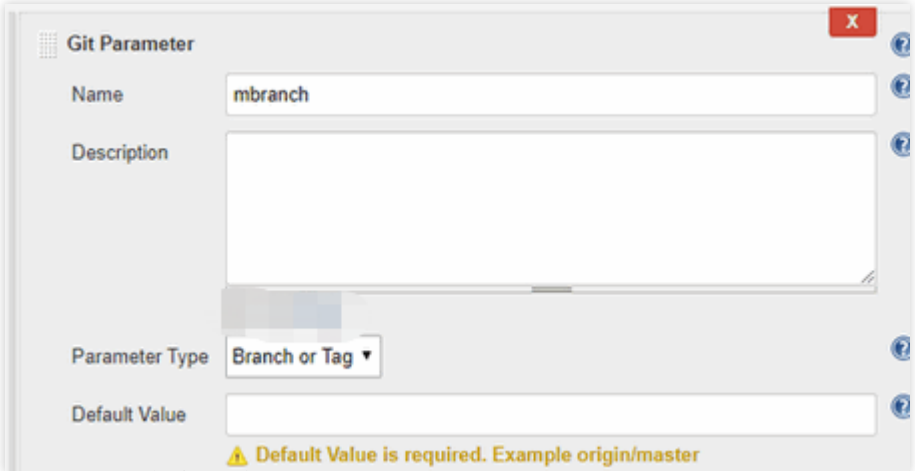
3. **OK**をクリックすると、タスクパラメータ設定画面に進みます。
4. タスクパラメータ設定ページで、基本情報を設定します。

説明：タスクに関する情報を定義して入力します。ここでは例として、「slave pod test」とします。

ビルドプロセスのパラメータ化：この項目にチェックを入れ、**パラメータの追加** > **Git Parameter**を選択します。

タスクパラメータの設定

1. 開いた「Git Parameter」パネルで、以下のパラメータを順番に設定します。下図に示すとおりです。

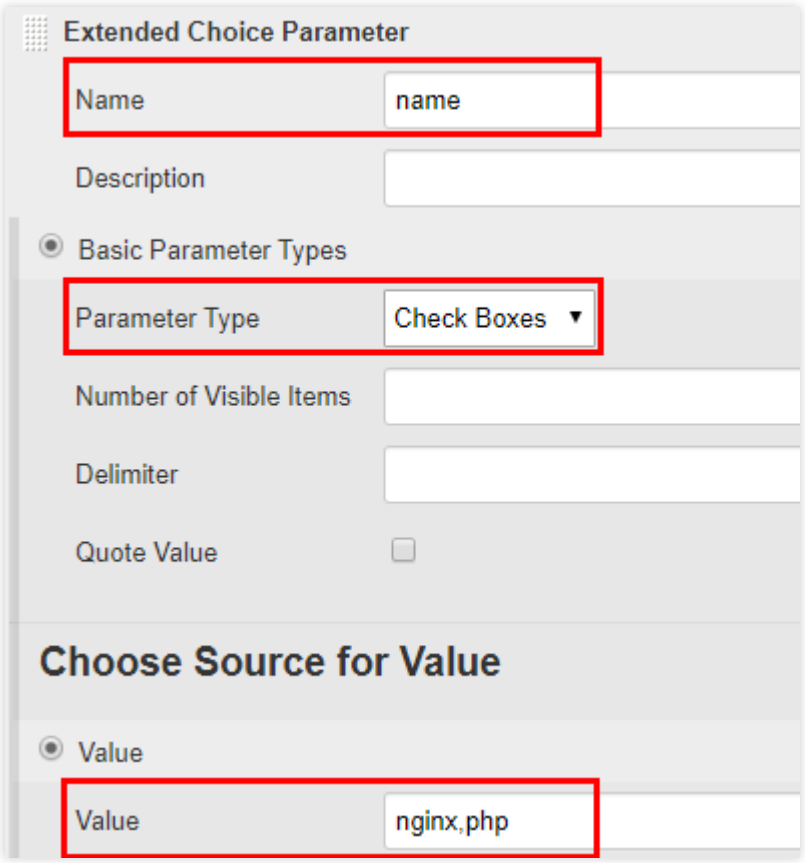


主要パラメータの情報は次のとおりです。その他のオプションはデフォルトを維持してください。

Name： `mbranch` を入力します。このパラメータは、取得したブランチのマッチングのために使用されます。

Parameter Type：**Branch or Tag**を選択します。

2. **パラメータの追加** > **Extended Choice Parameter**を選択し、開いた「Extended Choice Parameter」パネルで以下のパラメータを設定します。下図に示すとおりです。



主要パラメータの情報は次のとおりです。その他のオプションはデフォルトを維持してください。

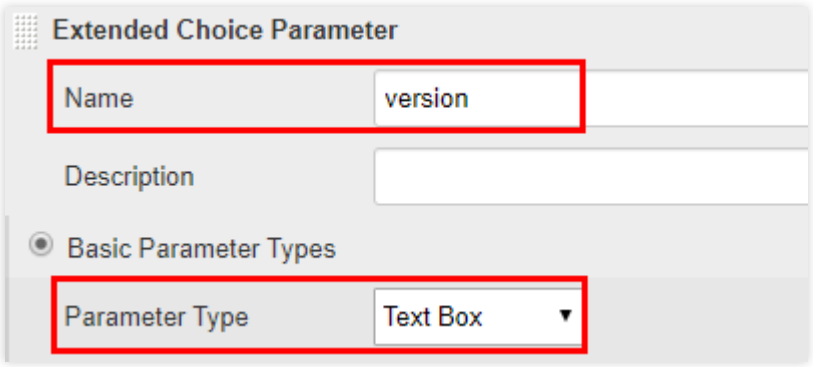
Name： `name` を入力します。このパラメータは、イメージを取得するために使用します。

Basic Parameter Types：この項目を選択します。

Parameter Type：**Check Boxes**を選択します。

Value：この項目を選択してカスタムイメージを入力します。この値は変数 `name` に渡されます。ここでは例として、 `nginx,php` とします。

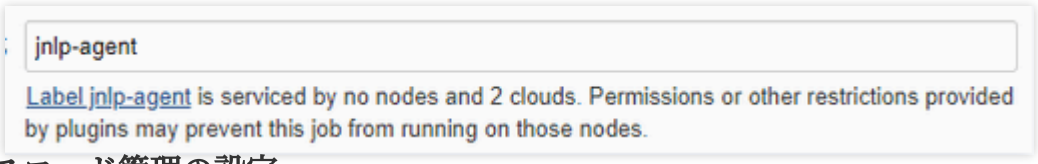
3. **パラメータの追加** > **Extended Choice Parameter**を選択し、開いた「Extended Choice Parameter」パネルで以下のパラメータを設定します。下図に示すとおりです。



主要パラメータの情報は次のとおりです。その他のオプションはデフォルトを維持してください。

Name： `version` を入力します。このパラメータはイメージバージョン変数を取得するために使用します。

Basic Parameter Types：この項目を選択します。
Parameter Type：**Text Box**を選択すると、テキスト形式でイメージ値を取得し、変数 `version` に渡します。
4. **プロジェクトの実行ノードを制限する**にチェックを入れ、タグ式にslave podテンプレートの設定ステップで設定済みのPodタグ `jnlp-agent` を入力します。下図に示すとおりです。

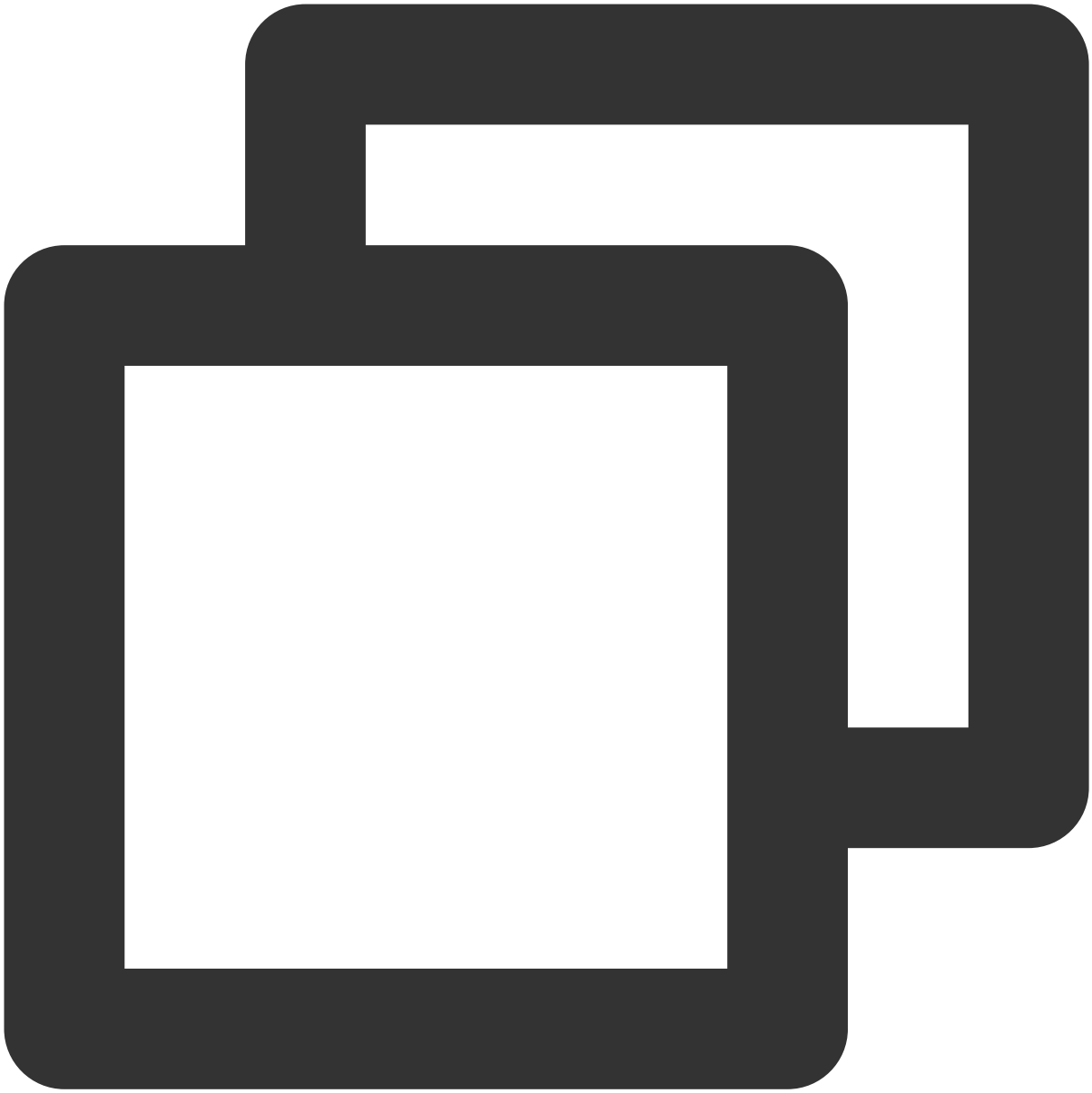


ソースコード管理の設定

「ソースコード管理」モジュールで、**Git**を選択して以下の情報を設定します。
Repositories:
Repository URL：お客様のgitlabアドレスを入力します。例： `https://gitlab.com/user-name/demo.git` 。
Credentials：gitlab認証の追加のステップで作成した認証情報を選択します。
Branches to build:
ブランチの指定（空の場合はany）：ブランチを動的に取得するために使用する、 `$mbranch` を入力します。この値は、Git Parameterパラメータで定義された `mbranch` 値に対応します。

Shellパッケージスクリプトの設定

1. 「作成」モジュールで、**作成ステップの追加 > shellの実行**を選択します。
2. 以下のスクリプトをコピーして、「コマンド」入力ボックスに貼り付け、**保存**をクリックします。
注意
スクリプト内のgitlabアドレス、TKEイメージアドレス、イメージウェアハウスのユーザーとパスワードなどの情報は例として使用しているものですので、必要に応じて変更してください。
ソースコードDocker buildをベースとするパッケージの作成場所を確保してください。作業ディレクトリ `/home/Jenkins/agent` が「コンテナリスト」のContainer Template作業ディレクトリと同じである必要があります。



```
echo " gitlabアドレスは、https://gitlab.com/[user]/[project-name]].git"です
echo "選択したブランチ（イメージ）は"$mbranch、"設定したブランチ（イメージ）のバージョンは"$ve
echo " TKEイメージアドレス：hkccr.ccs.tencentyun.com/[namespace]/[ImageName]"

echo "1.TKEイメージウェアハウスにログイン"
docker login --username=[username] -p [password] hkccr.ccs.tencentyun.com

echo "2.ソース コードDocker buildベースのパッケージビルド："
cd /home/Jenkins/agent/workspace/[project-name] && docker build -t $name:$version

echo "3.DockerイメージをTKEリポジトリにアップロード"
```

```
  docker tag $name:$version hkccr.ccs.tencentyun.com/[namespace]/[ImageName]:$name-$
  docker push hkccr.ccs.tencentyun.com/[namespace]/[ImageName]:$name-$version
```

このスクリプトは、以下の機能を提供します
選択したブランチ、イメージ名およびイメージバージョンを取得します。
コードとのマージ後にビルドしたdockerイメージをTKEイメージウェアハウスにプッシュします。

次の操作

この時点でslave podのビルドは成功しています。ビルドテストに移動してイメージをプッシュし、操作を検証してください。

TKEでJenkinsをデプロイする

最終更新日：：2023-04-28 11:08:19

概要

多くのDevOpsの要件は、Jenkinsの機能を借りて実装する必要があります。ここでは、TKEにJenkinsをデプロイする方法についてご説明します。

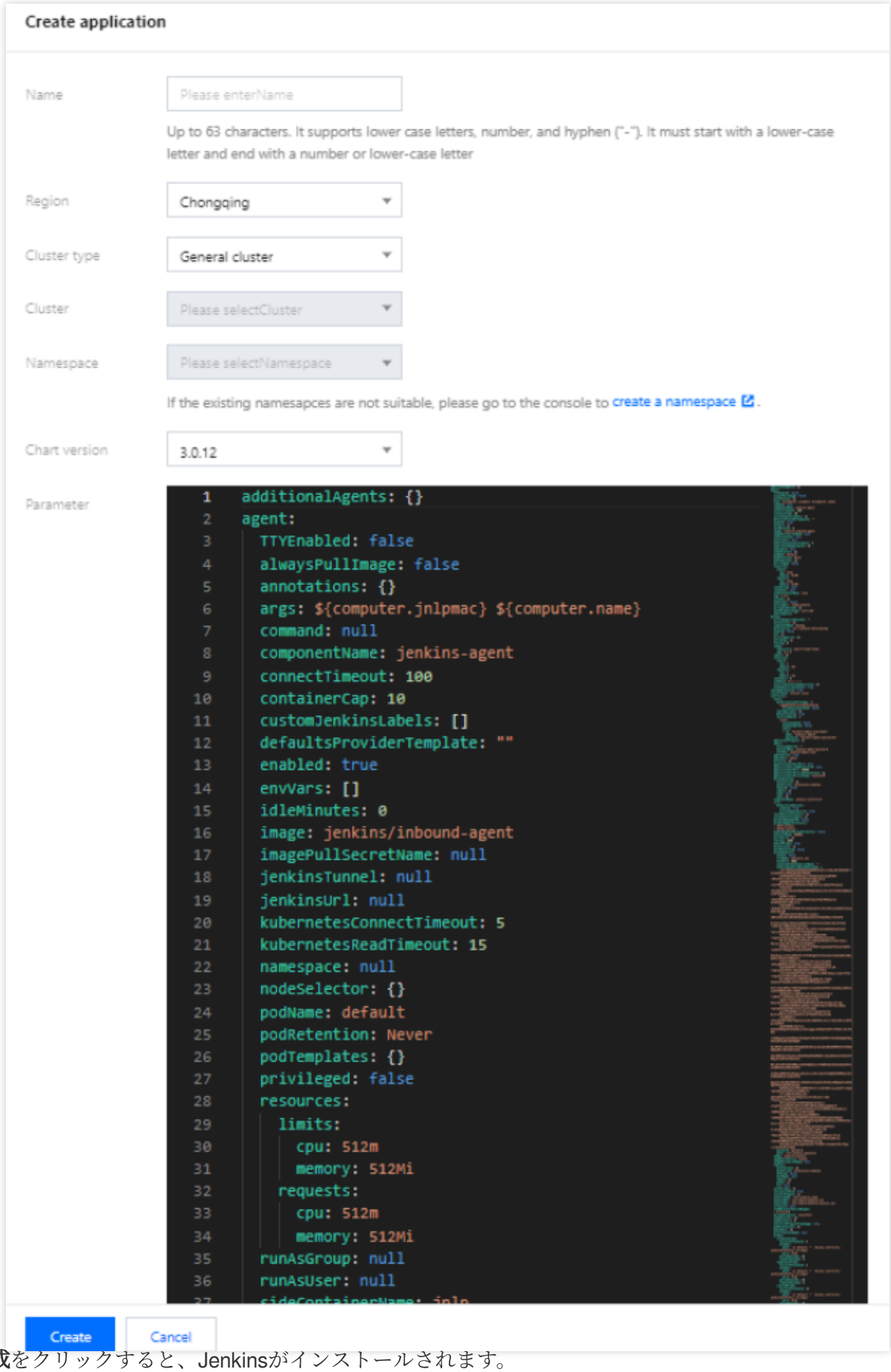
前提条件

TKEクラスターが作成済みであること。

操作手順

Jenkinsのインストール

1. TKEコンソールにログインし、左側ナビゲーションバーのアプリケーション市場を選択します。
2. **アプリケーション市場**ページでJenkinsを検索し、Jenkinsのアプリケーションページに進みます。
3. **アプリケーションの作成**をクリックすると、アプリケーションの作成ウィンドウの「パラメータ」のvalues.yamlの部分を、ニーズに合わせて微調整することができます。



4. **作成**をクリックすると、Jenkinsがインストールされます。

Jenkins UIの公開

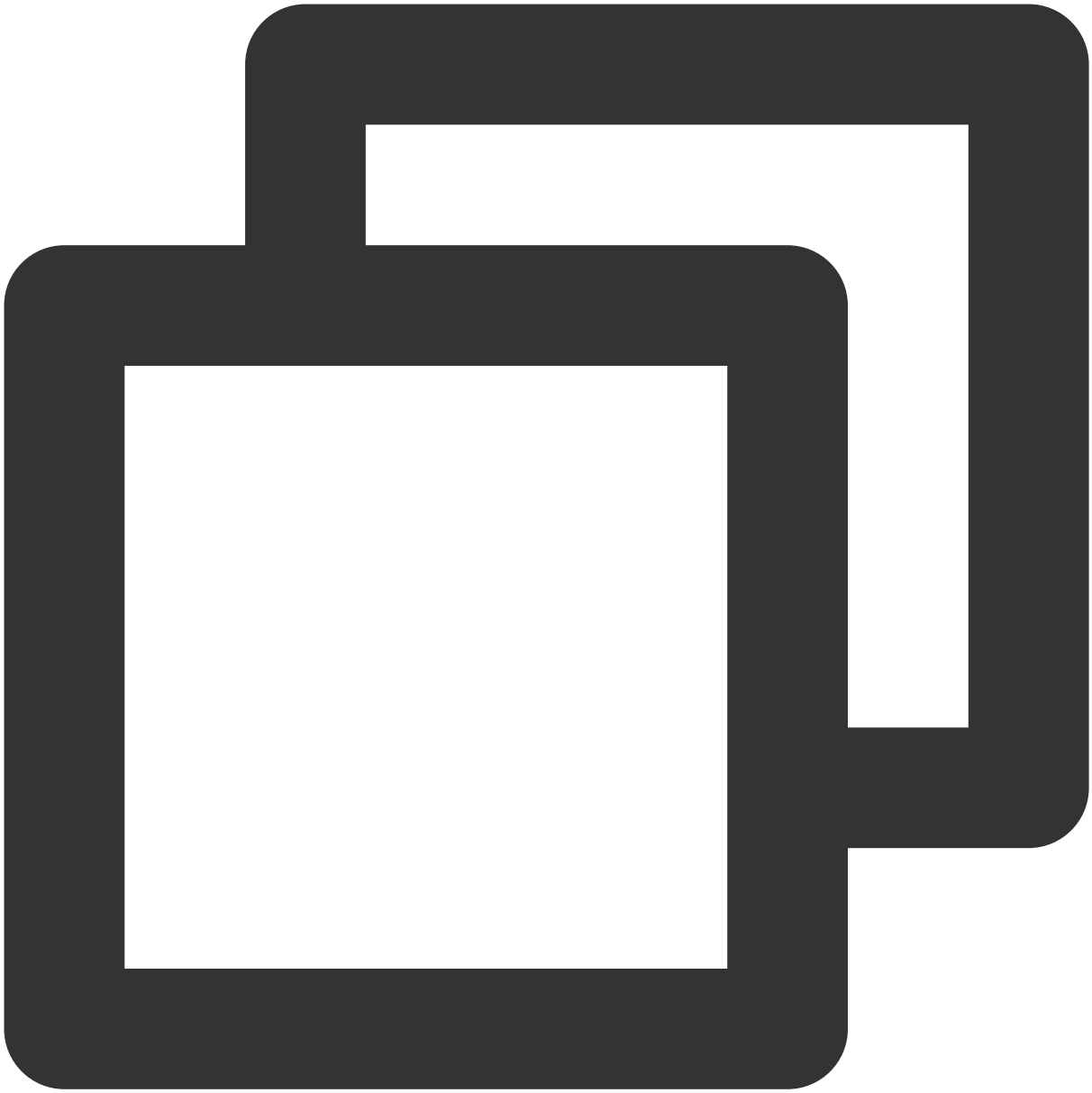
デフォルトでは、クラスター外ではJenkins UIにアクセスできません。Jenkins UIにアクセスする場合、通常はIngressを使用してアクセスを公開します。TKEは、CLBタイプIngressとNginxタイプIngressという2つのIngressを提供しています。ドキュメントを参照して選択してください。

説明

以下の例では、Jenkinsのバージョン2.263を使用しています。Jenkinsのバージョンによって UIの使い方に違いがあります。ビジネスニーズに応じて選択できます。

Jenkinsのログイン

Jenkins UIインターフェースに進み、初期ユーザー名とパスワードを入力してJenkinsバックエンドにログインします。ユーザー名はadminとし、初期パスワードは以下のコマンドで取得する必要があります。



```
kubectl -n devops get secret jenkins -o jsonpath='{.data.jenkins-admin-password}' |
```

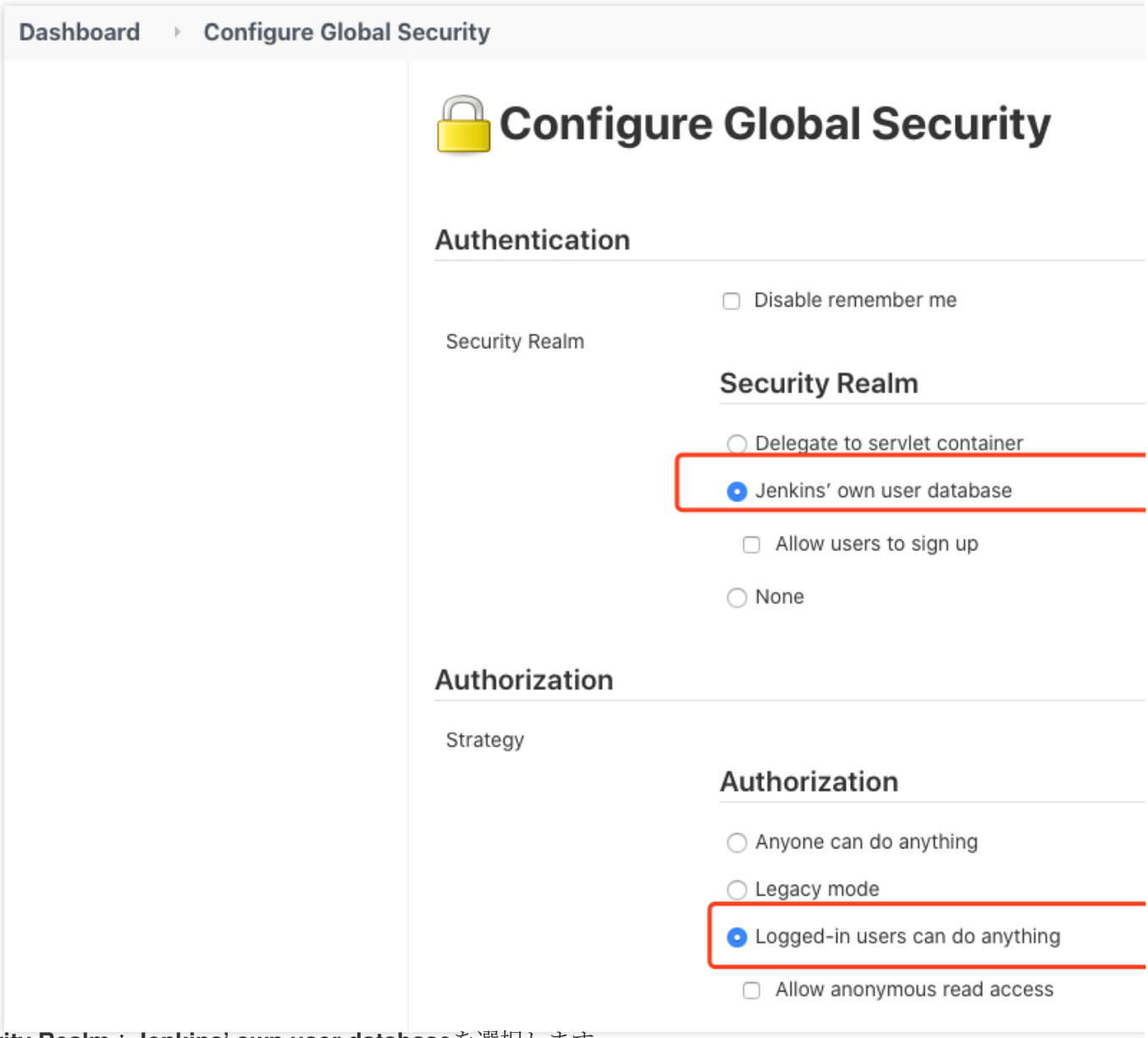
注意

上記コマンドを実行する場合、実際の環境にインストールされているネームスペースに置き換える必要があります。

ユーザーの作成

Jenkinsは一般ユーザーで管理することをお勧めします。一般ユーザーを作成する前に認証と権限付与のポリシーを設定する必要があります。

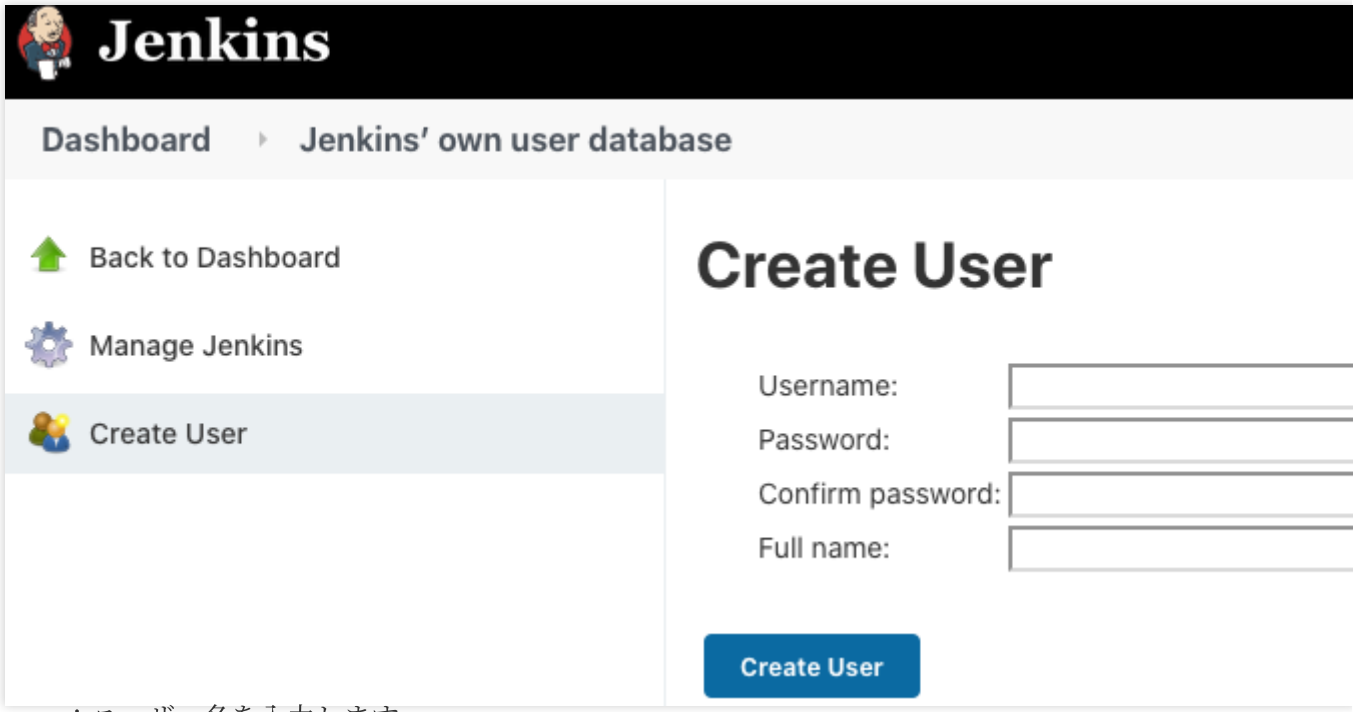
1. Jenkinsバックエンドにログインし、**Dashboard** > **Manage Jenkins** > **Security** > **Configure Global Security**を選択して認証・権限付与ポリシーページに進みます。下図に示すとおりです。



Security Realm：**Jenkins' own user database**を選択します。

Authorization：**Logged-in users can do anything**を選択します。

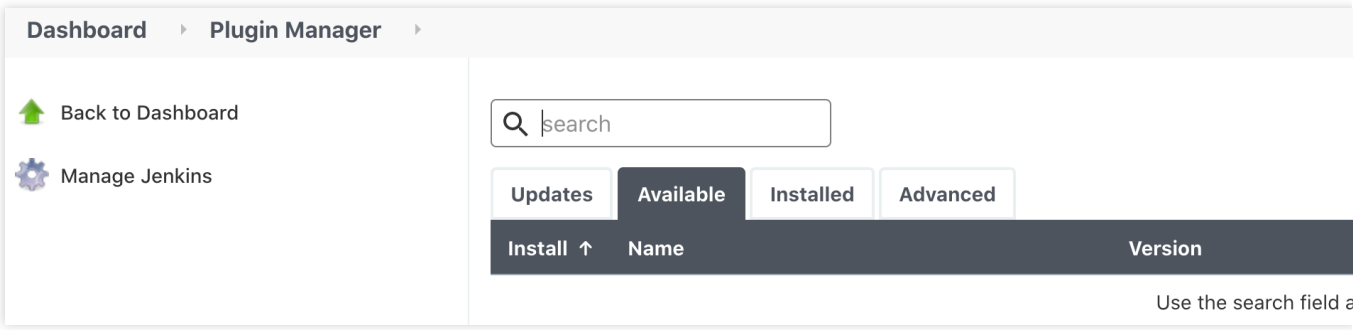
2. **Dashboard** > **Manage Jenkins** > **Security** > **Manage Users** > **Create User**を選択してユーザー作成インターフェースに進み、以下の手順に従ってユーザーを作成します。下図に示すとおりです。



Username：ユーザー名を入力します。
Password：ユーザーパスワードを入力します。
Confirm password：ユーザーパスワードを確認します。
Full name：ユーザーのフルネームを入力します。
3. **Create User** をクリックすると、ユーザーを作成できます。

プラグインのインストール

Jenkinsのバックエンドにログインし、**Dashboard** > **Manage Jenkins** > **System Configuration** > **Manage Plugins**を選択してプラグイン管理ページに進みます。

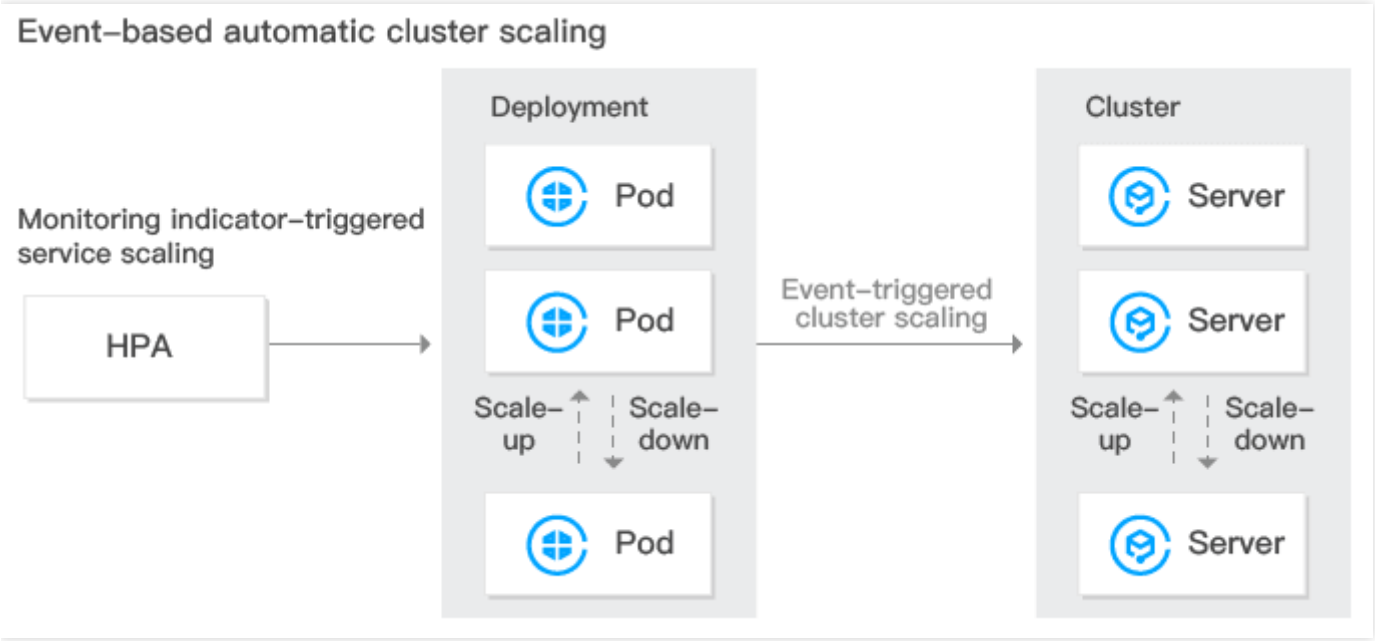


以下の一般的なプラグインをインストールすることができます。
kubernetes
pipeline
git
gitlab
github

オートスケーリング
クラスターオートスケーリング実践

最終更新日：：2022-09-22 17:00:19

テンセントクバネティスエンジン（Tencent Kubernetes Engine、TKE）はクラスターとサービスという2レベルのAuto Scaling機能を提供し、サービスの実行状況によって、コンテナのCPU、メモリ、帯域幅などの指標を監視してAuto Scalingサービスを行います。また、次の図に示すように、コンテナのデプロイ状況により、コンテナのリソースアサインが不足、あるいは余裕がある場合に自動でクラスターをスケーリングすることができます：



クラスターのオートスケーリング特徴

TKEでは、コンピューティングリソースの効率的な管理のため、クラスターのオートスケーリングを有効にすることができます。これにより、ユーザーは、業務に応じてスケーラビリティポリシーを設定できます。クラスターのスケーラビリティポリシーには、次のような特徴があります：

- CVM（Cloud Virtual Machine）を、業務の負荷に応じて動的かつリアルタイムに自動的に作成および解放し、最適なインスタンス数で業務に対応できるようにします。全過程に人の介入が不要で、ユーザーの人の手によるデプロイの負担がなくなります。
- ユーザーが業務に最適なノードリソースを使用できるように支援します。業務ニーズが増加した場合は、コンテナクラスターに適切な量のCVMをシームレスに自動的にユーザに追加することができます。業務ニーズが減少した場合は、不要なCVMを自動的に削減し、デバイスの使用率を向上させ、導入コストとインスタンスコストを削減します。

クラスターオートスケーリング機能説明

Kubernetes cluster autoscaling基本機能

- マルチスケーリンググループの設定をサポートします。
- 容量拡張と容量圧縮ポリシーの設定をサポートします。詳細については、Cluster Autoscalerをご参照ください。

TKEスケーリンググループ拡張機能

- 新しいモデルのスケーリンググループの新規作成をサポートします（推奨）。
- クラスター内のノードからテンプレートとしてスケーリンググループを新規作成できます。
- スポットインスタンスのスケーリンググループをサポートしています（推奨）。
- 対応モデルが売り切れた場合は自動的に適切なスケーリンググループを適合させます。
- アベイラビリティーゾーン間のスケーラグループの設定をサポートします。

クラスターオートスケーリング制限

- クラスターオートスケーリングの拡張可能なノード数は、プライベートネットワーク、コンテナネットワーク、TKEクラスターノードのクォータ、および購入可能なCVMのクォータによって制限されます。
- 拡張ノードは、モデルの現在の販売状況によって制限されます。モデルが売り切れになった場合は、ノードを拡張できないため、複数のスケーラグループを構成することをお勧めします。
- ワークロードにおけるコンテナのrequest値を設定してください。自動拡張のトリガーはクラスター内にリソース不足のためにスケジューリングできないPodが存在することであり、リソースが十分かどうかの判断はPodのrequestに基づいて行われます。
- 監視メトリックに基づくノードのオートスケーリングを有効にしないことをお勧めします。
- スケーラグループを削除すると、スケーラグループ内のCVMも破棄されますので、十分注意してください。

クラススケーラグループの構成

- マルチスケーラグループ構成（推奨）
  クラスターに複数のスケーラグループが存在する場合は、自動拡張圧縮コンポーネントは、選択した拡張アルゴリズムに従ってスケーラグループを選択して拡張します。一度に1つのスケーラグループを選択します。ターゲットのスケーラグループが売り切れなどの理由で拡張に失敗した場合は、一定期間スリープ状態にします。同時に二番目にマッチしたスケーラグループを再選択して拡張するようにトリガーします。

- ランダム：拡張するスケーラグループをランダムに選択します。

- Most-pods：現在pendingしているPodとスケーラグループのモデルに応じて、より多くのPodをスケジューリングできるスケーラグループを選択して、拡張を実行します。

- Least-waste：現在pendingしているPodとスケーラグループのモデルに応じて、Podスケジューリング後のリソース残量がより少ないスケーラグループを選択して拡張を実行します。

  特定のモデルが売り切れにならないように、クラスター内に複数の異なるモデルのスケーラグループを構成することをお勧めします。入札モデルと通常モデルを併用してコストを削減できます。

- クラスターのシングルスケーラブルグループ構成
  クラスターの拡張モデルとしてシングルモデルのみを受け入れる場合は、複数のサブネットの複数の異なるアベイラビリティーゾーンにスケーラグループを構成することをお勧めします。

TKE上でカスタム指標を使用して自動スケーリングします

最終更新日：：2023-04-27 18:15:01

ユースケース

Tencent Kubernetes Engine（TKE）はCustom MetricsAPIに基づいて複数の自動スケーリングに使用される指標をサポートします。CPU、メモリ、ハードディスク、ネットワークおよびGPUに関連する指標をカバーし、大部分のHPA自動スケーリングのシナリオをカバーします。詳細なリストについては、自動スケーリング指標説明をご参照ください。例えば業務の単一レプリカのQPSサイズに基づいて自動スケーリングなどを行う複雑なシナリオに対し、prometheus-adapterをインストールすることによって自動スケーリングを実現します。KubernetesがCustom Metrics APIとExternal Metrics APIを提供することによってHPA指標を拡張し、ユーザーが実際のニーズに応じてカスタマイズできるようにします。prometheus-adapterは上記の2種類のAPIをサポートし、実際の環境では、Custom Metrics APIを使用すれば大部分のシナリオに対応することができます。ここではCustom Metrics APIによってカスタム指標を使用して自動スケーリングを行う方法についてご説明します。

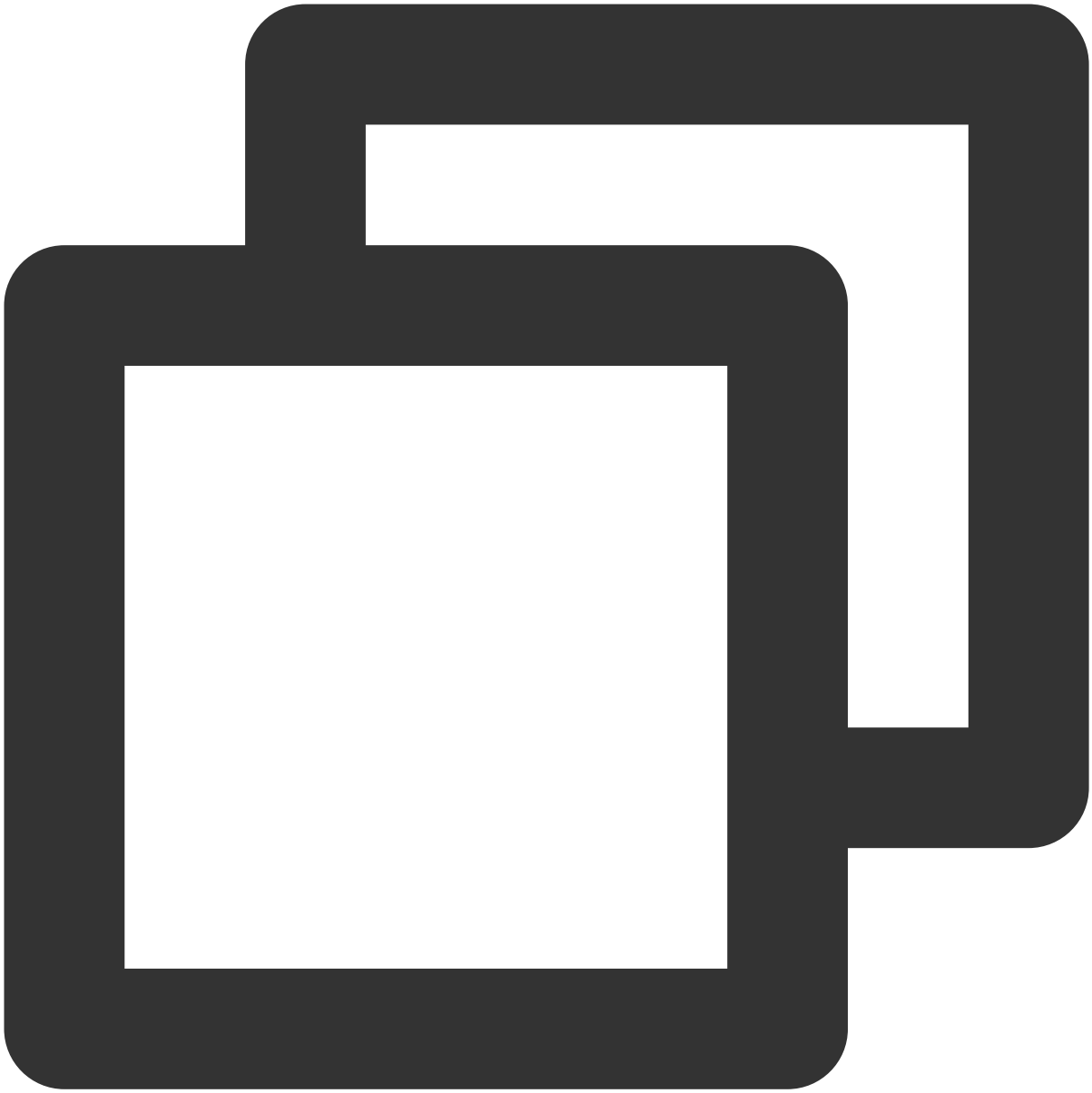
前提条件

バージョン1.12以上のTKEクラスターを作成済みであること。詳細については、クラスターの作成をご参照ください。
Prometheusをデプロイし、対応するカスタム指標を収集済みであること。
Helmをインストール済みであること。

操作手順

監視指標の公開

本文ではGolang業務プログラムを例とします。この例ではプログラムが `httpserver_requests_total` 指標を公開し、HTTPのリクエストを記録し、この指標によって業務プログラムのQPS値を算出することができます。次のとおりです。

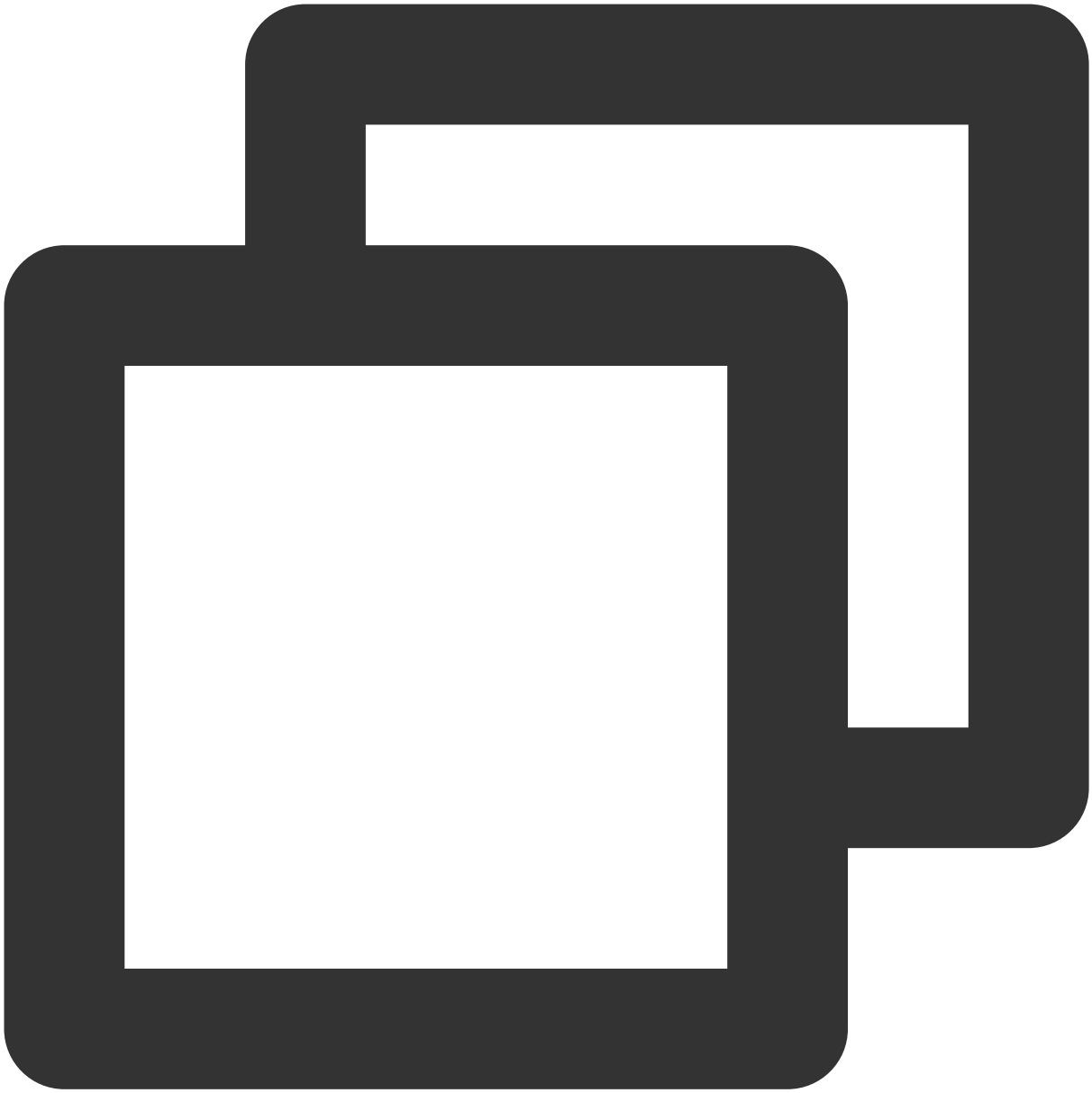


```
package main

import (
        "github.com/prometheus/client_golang/prometheus"
        "github.com/prometheus/client_golang/prometheus/promhttp"
        "net/http"
        "strconv"
)

var (
HTTPRequests = prometheus.NewCounterVec(
```

```
        prometheus.CounterOpts{
            Name: "httpserver_requests_total",
            Help: "Number of the http requests received since the server started",
        },
        []string{"status"},
    )
)

func init() {
    prometheus.MustRegister(HTTPRequests)
}

func main(){
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        path := r.URL.Path
        code := 200
        switch path {
        case "/test":
            w.WriteHeader(200)
            w.Write([]byte("OK"))
        case "/metrics":
            promhttp.Handler().ServeHTTP(w, r)
        default:
            w.WriteHeader(404)
            w.Write([]byte("Not Found"))
        }
        HTTPRequests.WithLabelValues(strconv.Itoa(code)).Inc()
    })
    http.ListenAndServe(":80", nil)
}
```

業務プログラムのデプロイ

前記のプログラムをコンテナイメージにパッケージ化し、その後クラスターにデプロイします。例えばDeploymentを使用してデプロイします。



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpserver
  namespace: httpserver
spec:
  replicas: 1
  selector:
    matchLabels:
      app: httpserver
  template:
```

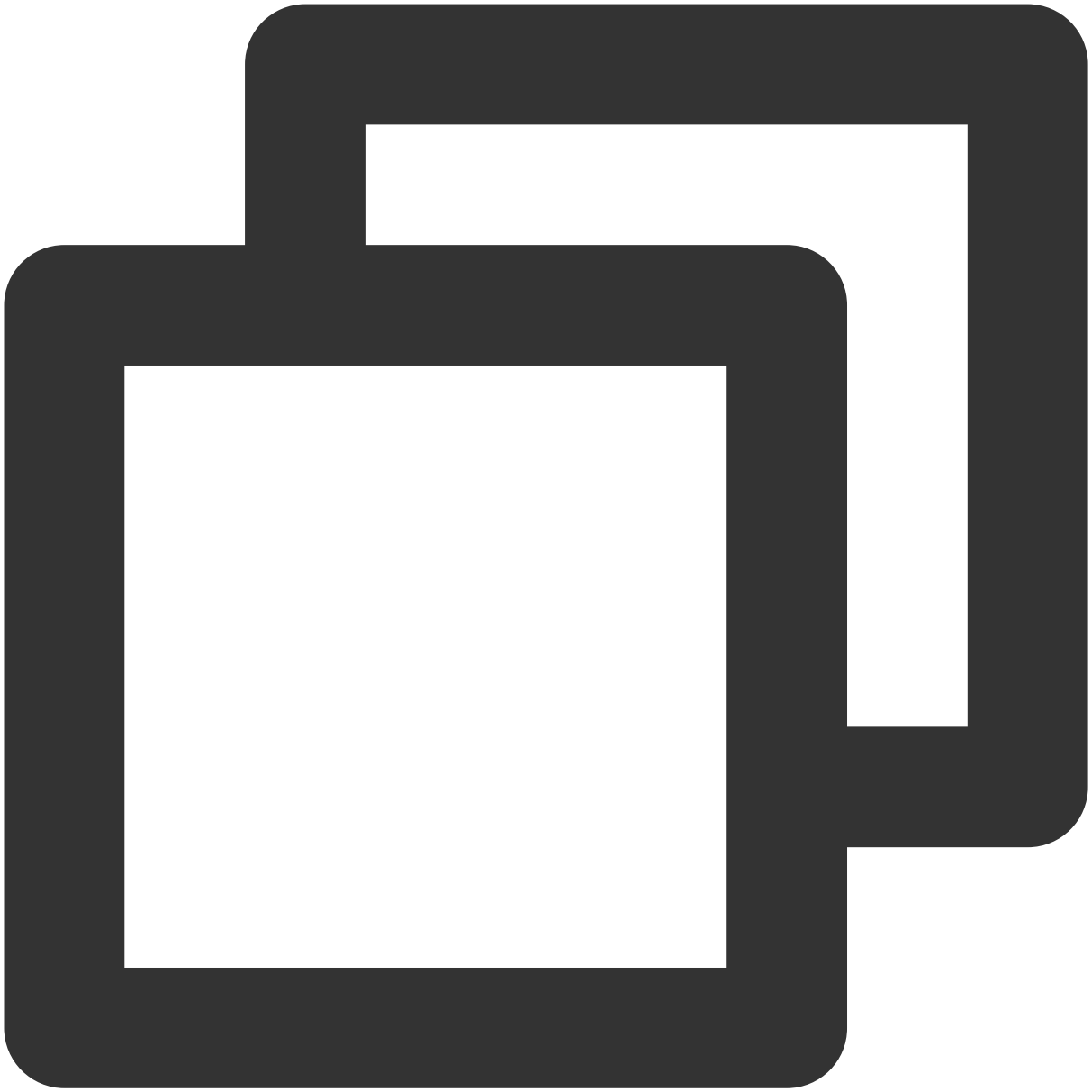
```
    metadata:
      labels:
        app: httpserver
    spec:
      containers:
      - name: httpserver
        image: registry.imroc.cc/test/httpserver:custom-metrics
        imagePullPolicy: Always

---

apiVersion: v1
kind: Service
metadata:
  name: httpserver
  namespace: httpserver
  labels:
    app: httpserver
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/path: "/metrics"
    prometheus.io/port: "http"
spec:
  type: ClusterIP
  ports:
  - port: 80
    protocol: TCP
    name: http
  selector:
    app: httpserver
```

Prometheusによる業務モニタリングの収集

Promtheus収集ルールまたはServiceMonitorによってPromtheusを設定して業務が公開する監視指標を収集することができます。

方式1：Promtheus収集ルールを設定する

Promtheusの収集ルール設定ファイル内に以下の収集ルールを追加します。次のとおりです。

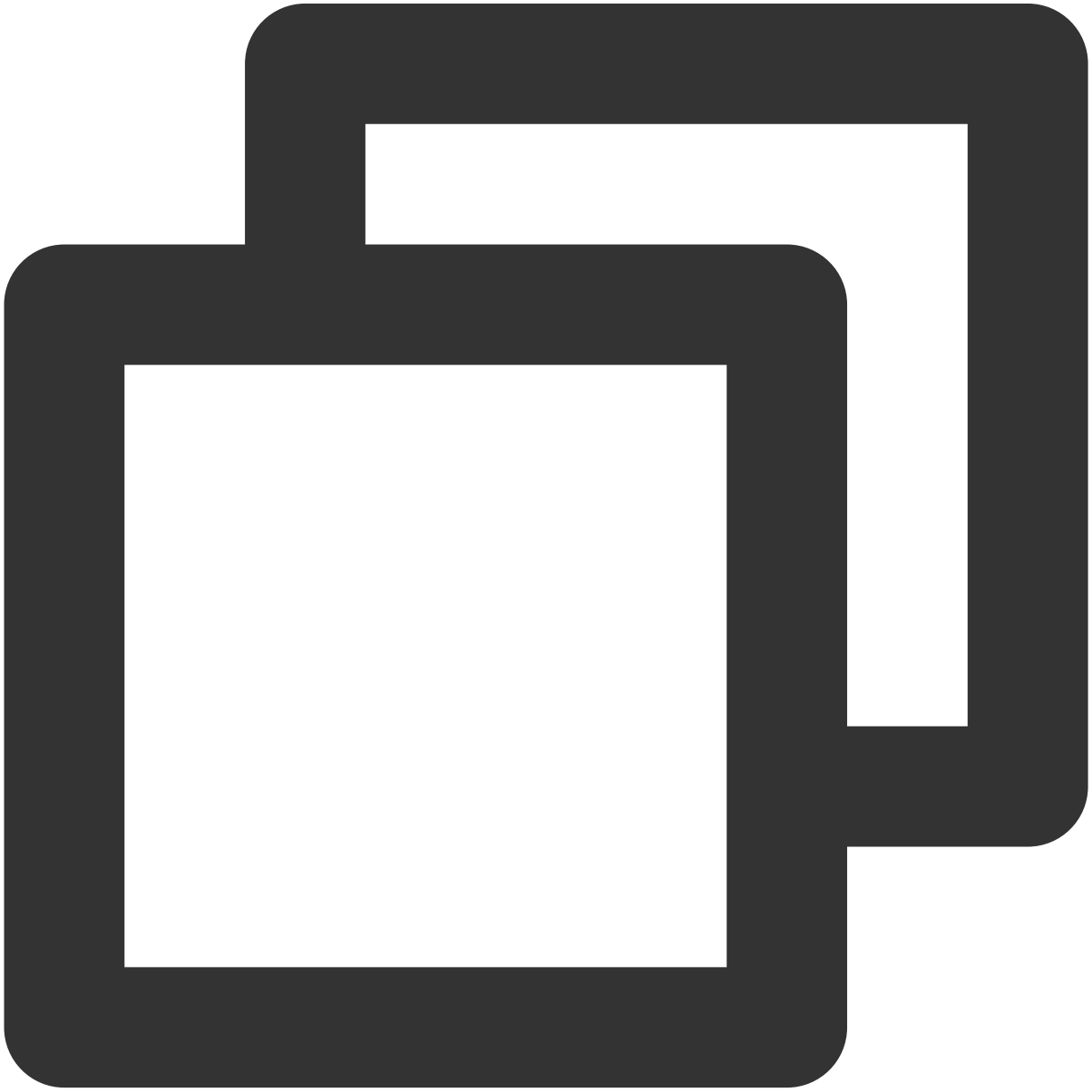


```
- job_name: httpserver
    scrape_interval: 5s
    kubernetes_sd_configs:
    - role: endpoints
        namespaces:
            names:
            - httpserver
    relabel_configs:
    - action: keep
        source_labels:
        - __meta_kubernetes_service_label_app
```

```
            regex: httpserver
        - action: keep
            source_labels:
            - __meta_kubernetes_endpoint_port_name
            regex: http
```

方式2：ServiceMonitorを設定する

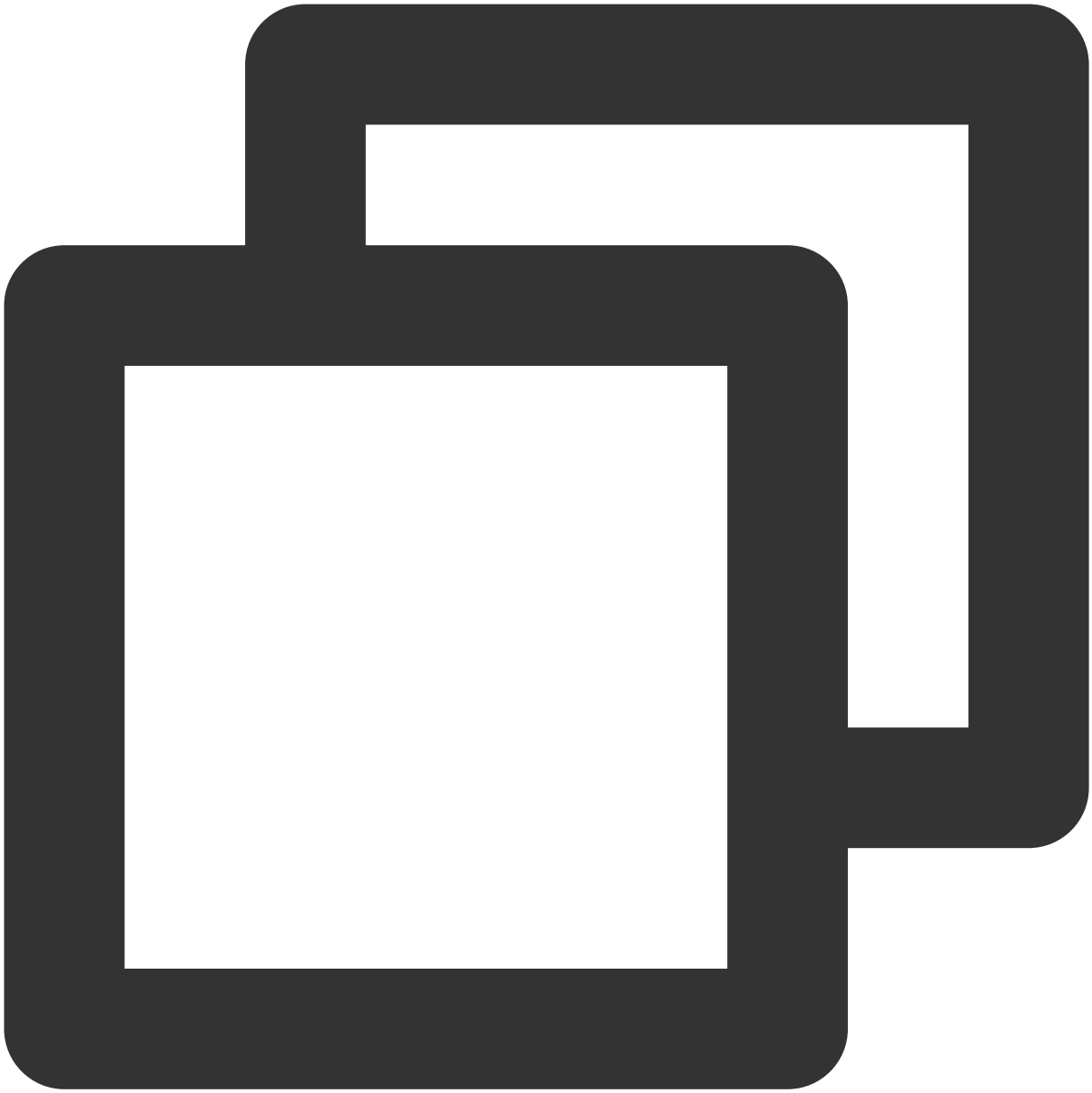
prometheus-operatorがインストールされている場合、ServiceMonitorのCRDのオブジェクトを作成することによってPrometheusを設定することができます。次のとおりです。



```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: httpserver
spec:
  endpoints:
  - port: http
    interval: 5s
  namespaceSelector:
    matchNames:
    - httpserver
  selector:
    matchLabels:
      app: httpserver
```

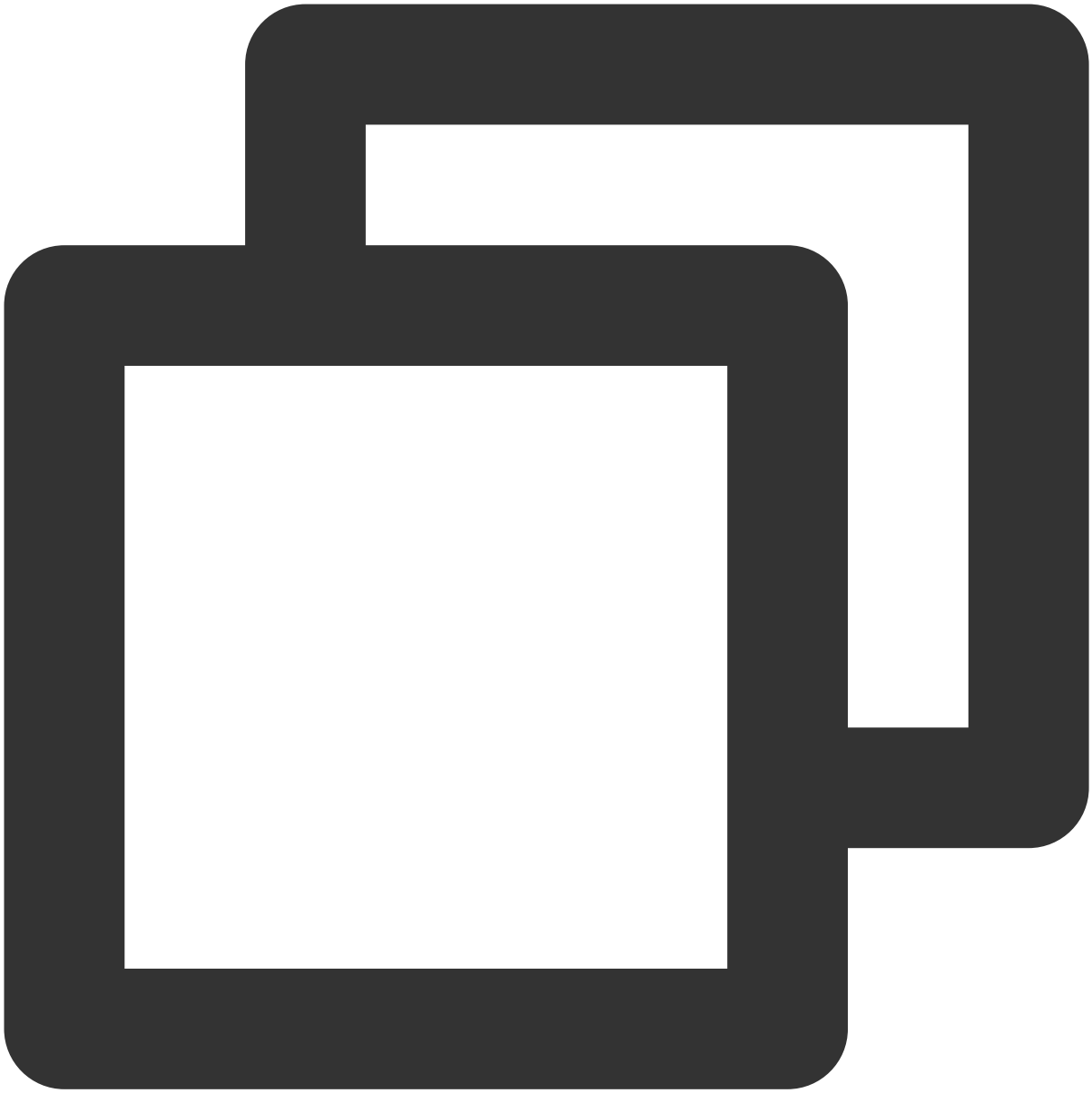
prometheus-adapterをインストールする

1. Helmを使用してprometheus-adapterをインストールし、インストール前にカスタム指標を確認して設定してください。上記の監視指標の公開の例に従い、業務内で `httpserver_requests_total` 指標を使用してHTTPリクエストを記録するため、以下のようなPromQLによって各業務のPodのQPSモニタリングを算出することができます。次のとおりです。



```
sum(rate(http_requests_total[2m])) by (pod)
```

2. それをprometheus-adapterの設定に変換し、 `values.yaml` を作成します。内容は以下のとおりです。



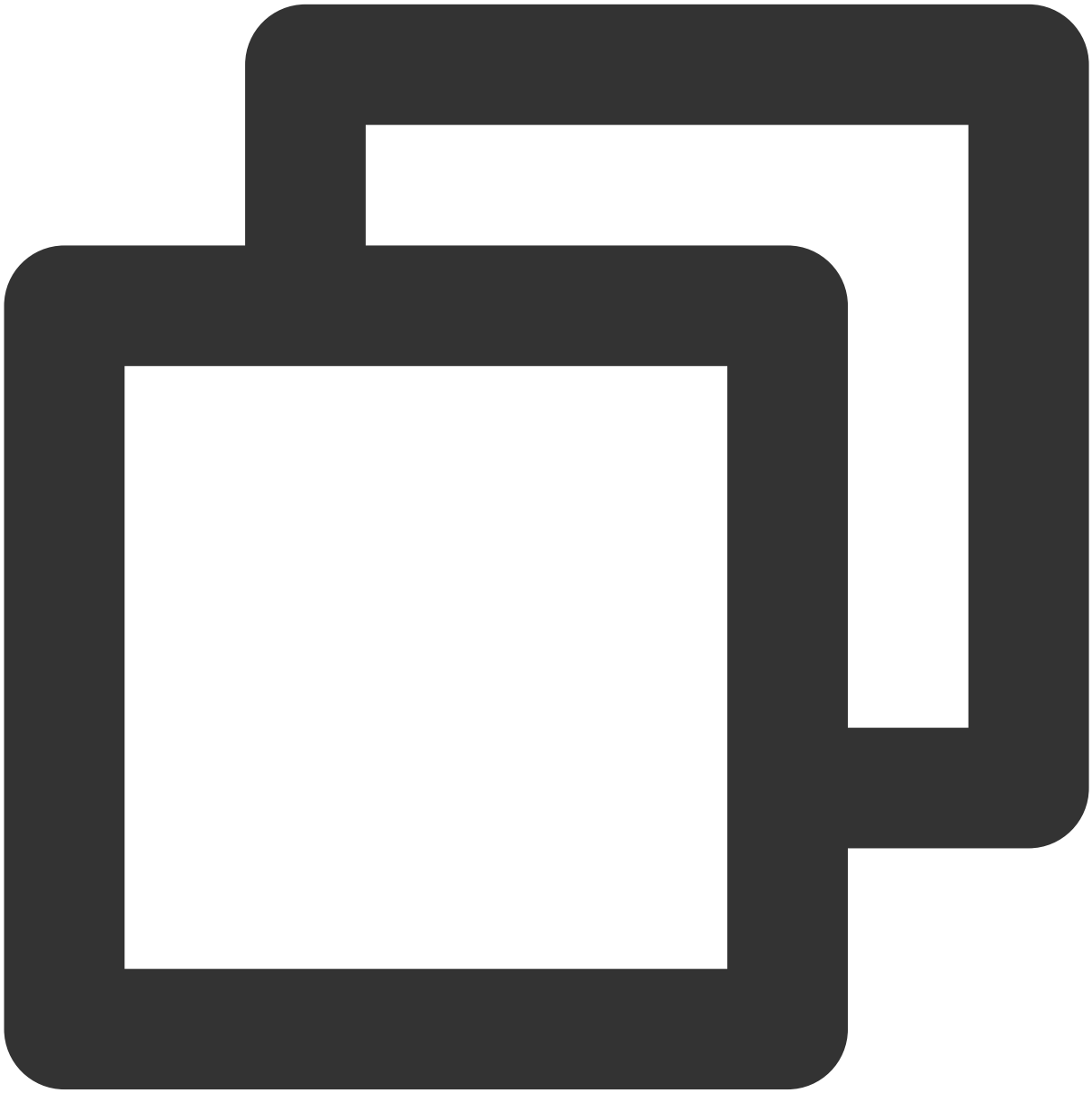
```
rules:
   default: false
   custom:
   - seriesQuery: 'httpserver_requests_total'
     resources:
       template: <<.Resource>>
     name:
       matches: "httpserver_requests_total"
       as: "httpserver_requests_qps" # PromQLが算出したQPS指標
     metricsQuery: sum(rate(<<.Series>>{<<.LabelMatchers>>}[1m])) by (<<.GroupBy>>)
prometheus:
```

```
    url: http://prometheus.monitoring.svc.cluster.local # Prometheus APIのアドレスを置
    port: 9090
```

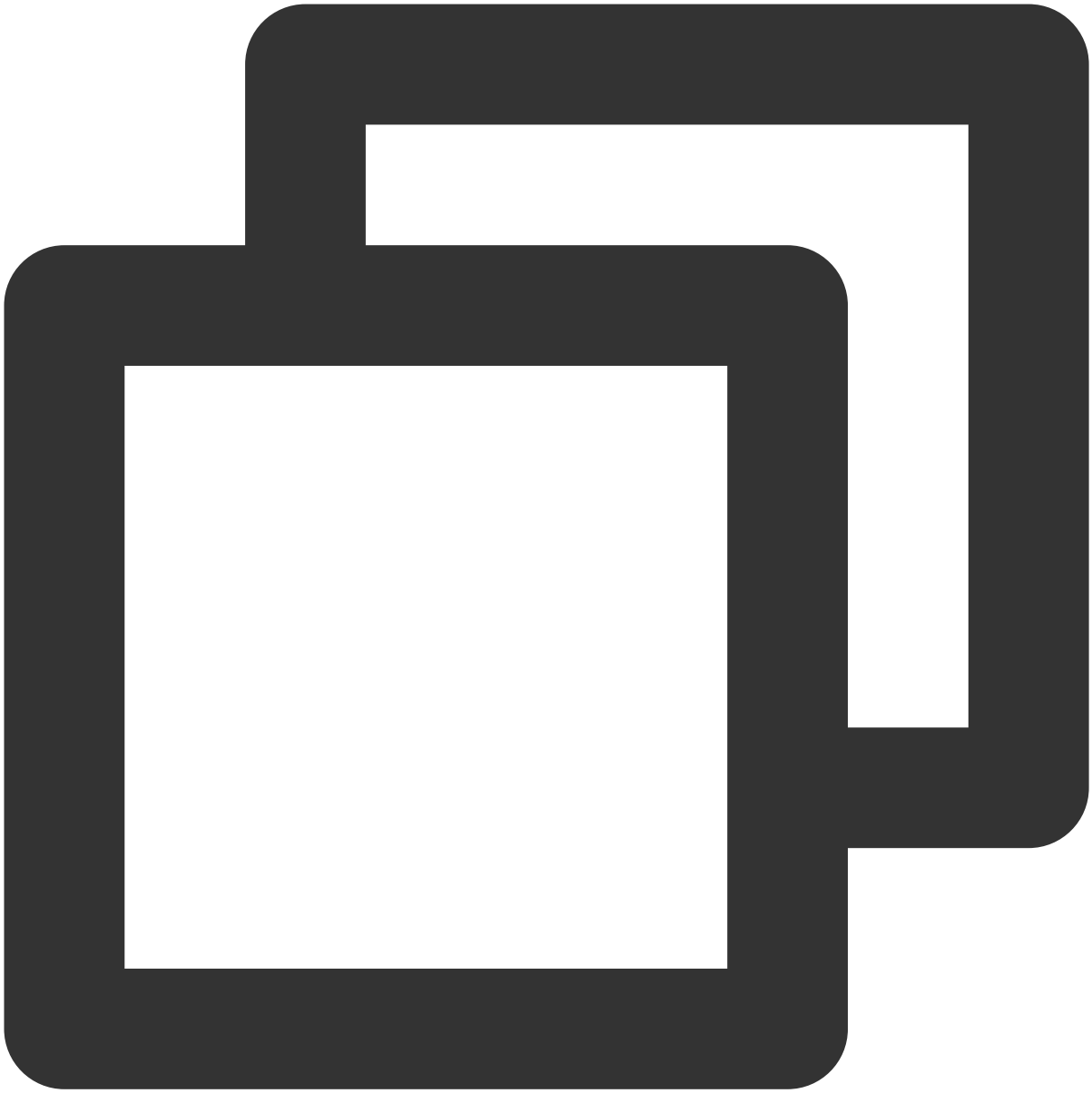
3. 以下のHelmコマンドを実行して prometheus-adapterをインストールします。次のとおりです。

注意

インストール前にTKEが登録したCustom Metrics APIを削除する必要があります。削除コマンドは次のとおりです。



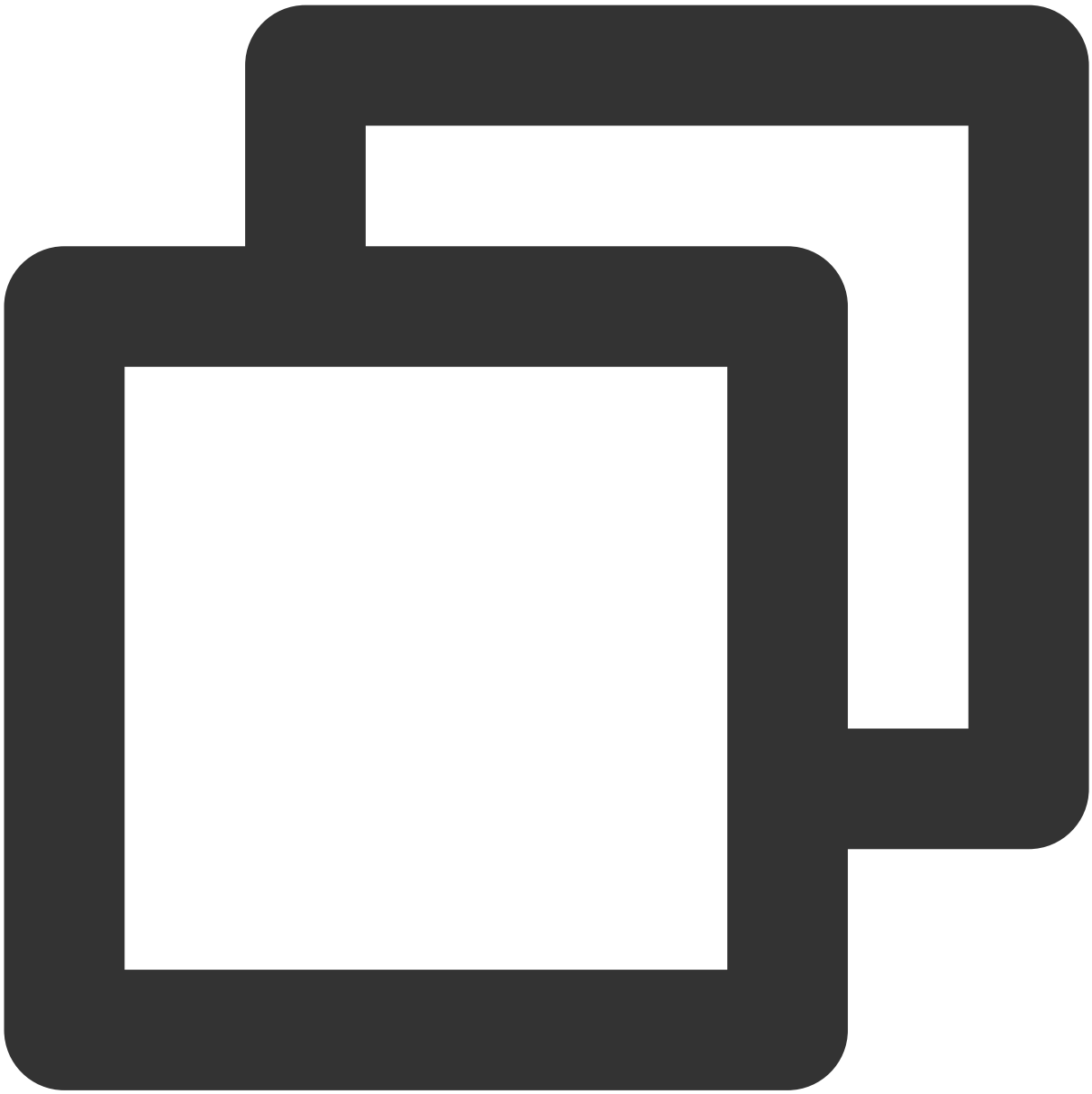
```
kubectl delete apiservice v1beta1.custom.metrics.k8s.io
```



```
helm repo add prometheus-community https://prometheus-community.github.io/helm-char
helm repo update
# Helm 3
helm install prometheus-adapter prometheus-community/prometheus-adapter -f values.y
# Helm 2
# helm install --name prometheus-adapter prometheus-community/prometheus-adapter -f
```

テストの検証

正確にインストールされていれば、以下のコマンドを実行し、Custom Metrics APIが返した設定されたQPS関連指標を確認することができます。次のとおりです。



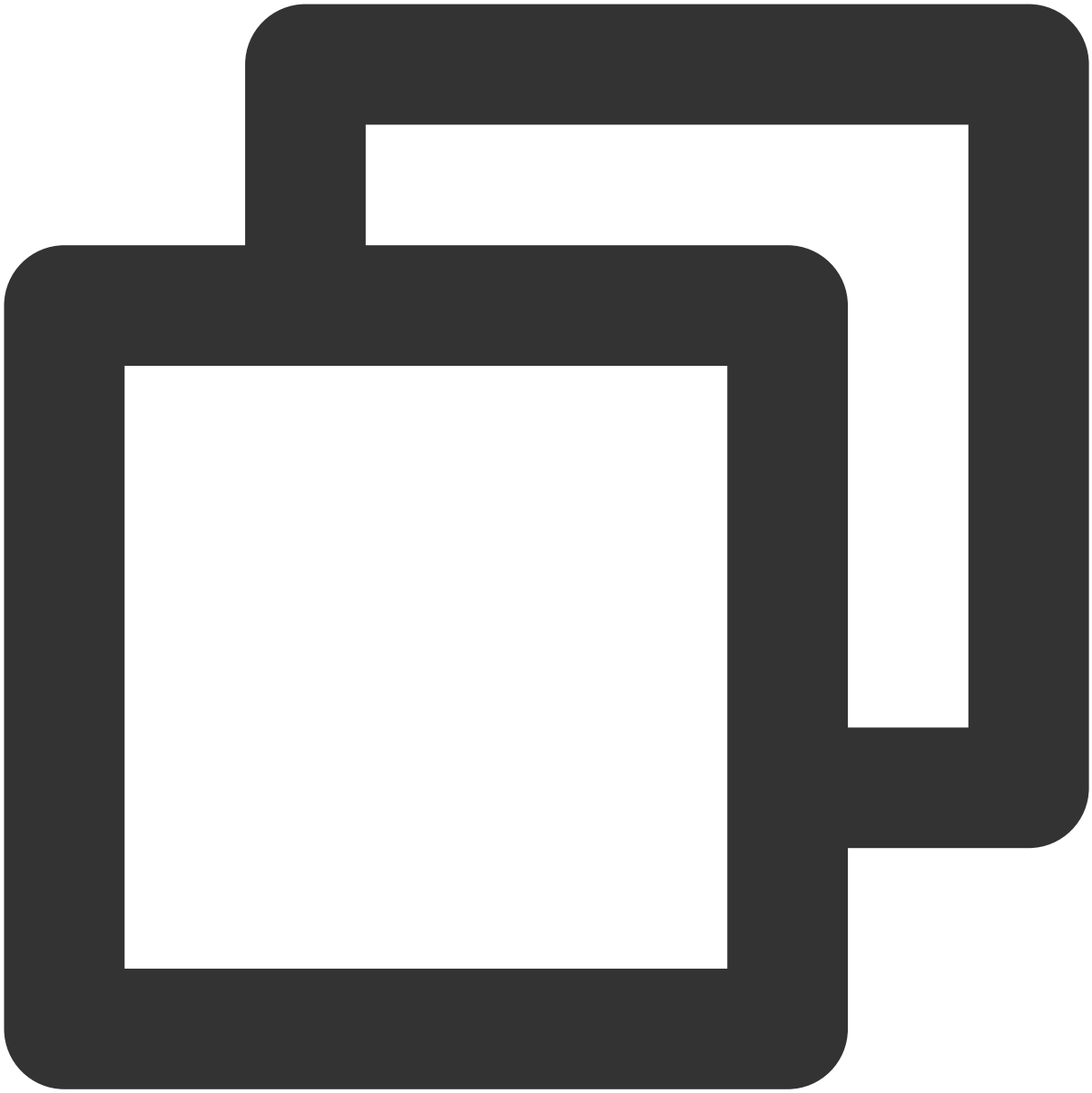
```
$ kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "custom.metrics.k8s.io/v1beta1",
  "resources": [
    {
      "name": "jobs.batch/httpserver_requests_qps",
```

```
      "singularName": "",
      "namespaced": true,
      "kind": "MetricValueList",
      "verbs": [
        "get"
      ]
    },
    {
      "name": "pods/httpserver_requests_qps",
      "singularName": "",
      "namespaced": true,
      "kind": "MetricValueList",
      "verbs": [
        "get"
      ]
    },
    {
      "name": "namespaces/httpserver_requests_qps",
      "singularName": "",
      "namespaced": false,
      "kind": "MetricValueList",
      "verbs": [
        "get"
      ]
    }
  ]
}
```

以下のコマンドを実行すると、PodのQPS値を確認することができます。次のとおりです。

説明

下記の例ではQPSが500mで、QPS値が0.5であることを表します。

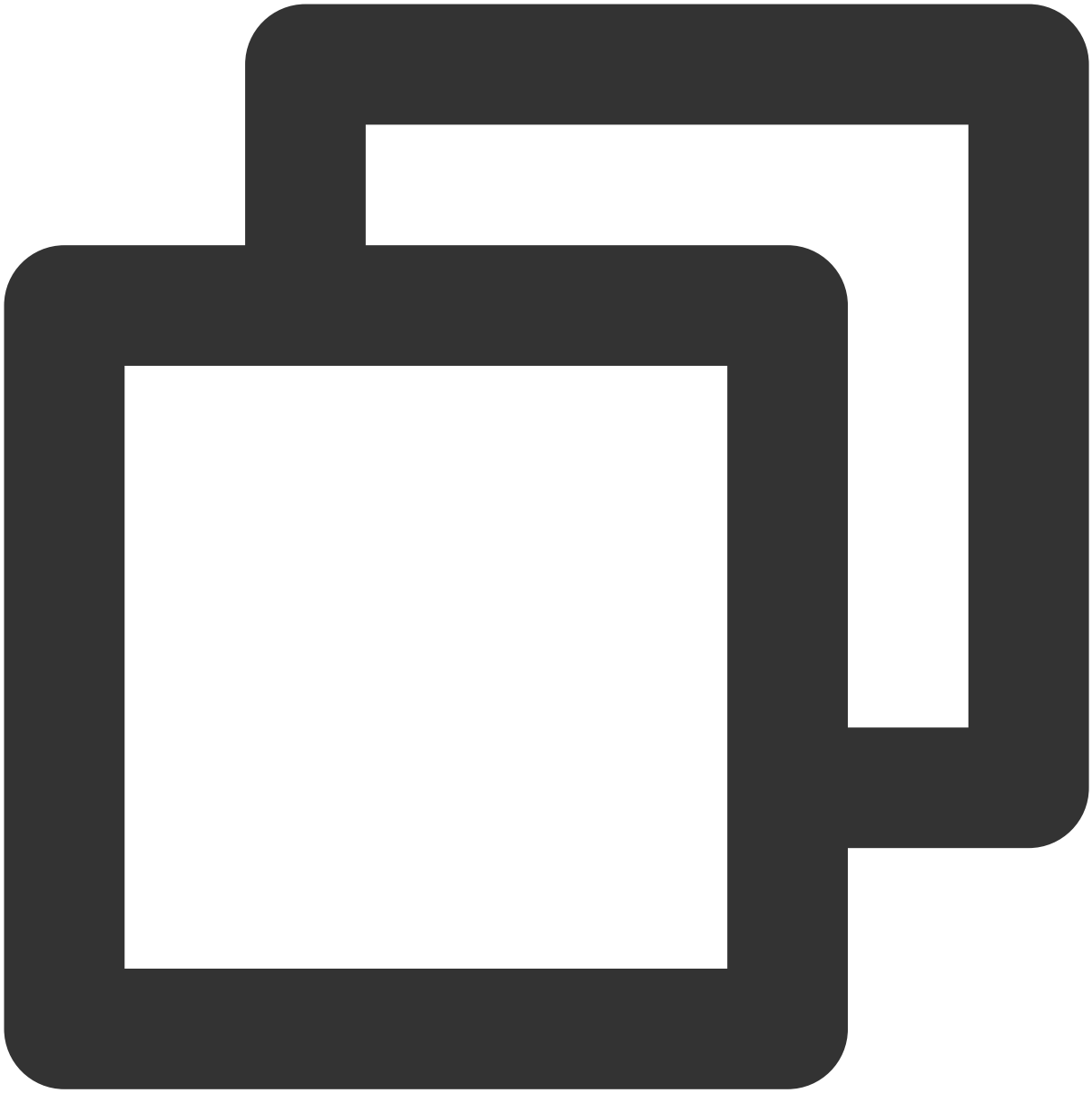


```
$ kubectl get --raw /apis/custom.metrics.k8s.io/v1beta1/namespaces/httpserver/pods/
{
  "kind": "MetricValueList",
  "apiVersion": "custom.metrics.k8s.io/v1beta1",
  "metadata": {
    "selfLink": "/apis/custom.metrics.k8s.io/v1beta1/namespaces/httpserver/pods/%2A
  },
  "items": [
    {
      "describedObject": {
        "kind": "Pod",
```

```
        "namespace": "httpserver",
        "name": "httpserver-6f94475d45-7rln9",
        "apiVersion": "/v1"
      },
      "metricName": "httpserver_requests_qps",
      "timestamp": "2020-11-17T09:14:36Z",
      "value": "500m",
      "selector": null
    }
  ]
}
```

HPAのテスト

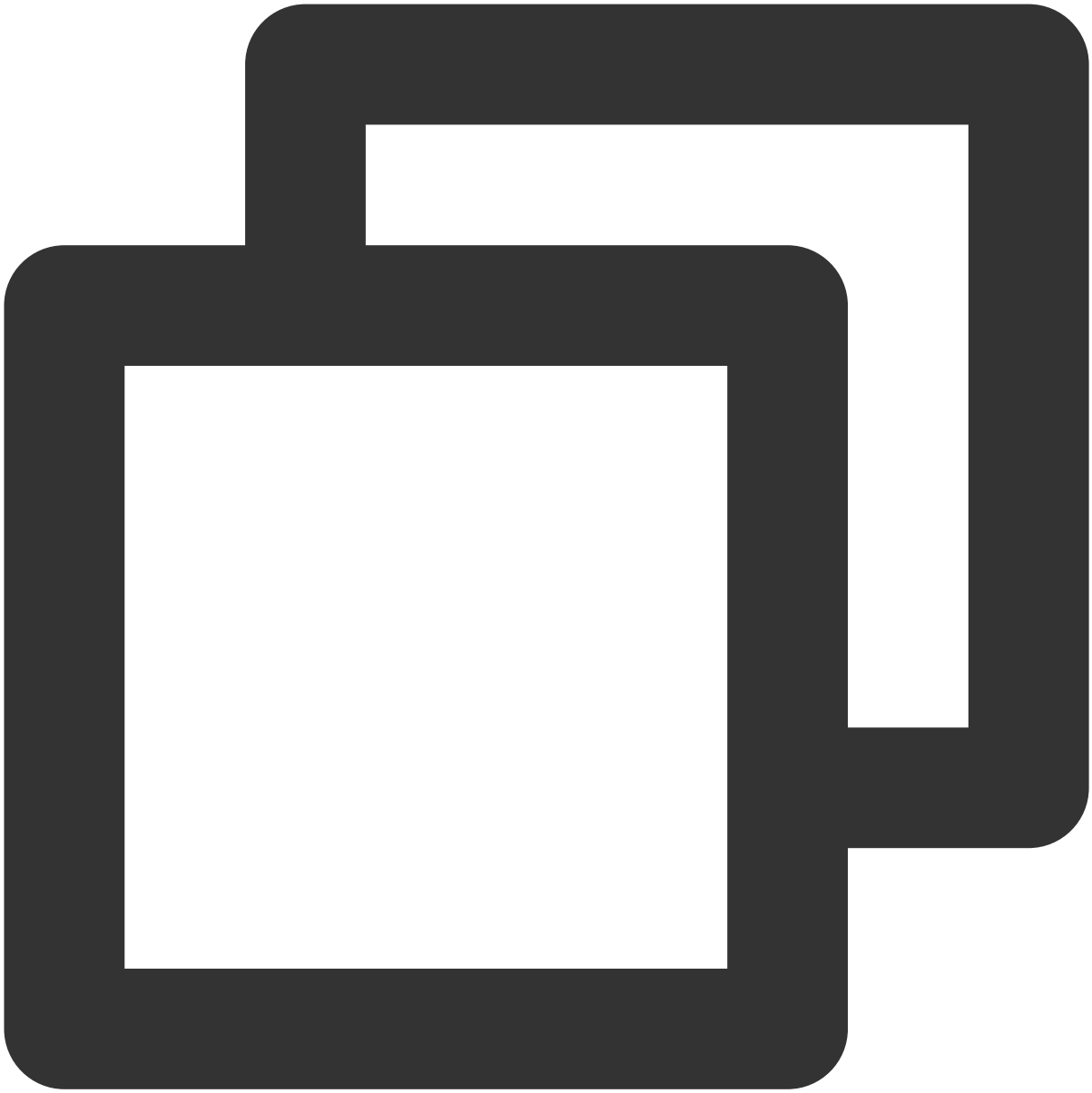
各業務Podの平均QPSが50に達した時にスケーリングをトリガーすると設定した場合、最小レプリカは1個、最大レプリカは1000個です。設定は次のとおりです。



```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: httpserver
  namespace: httpserver
spec:
  minReplicas: 1
  maxReplicas: 1000
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
```

```
    name: httpserver
  metrics:
  - type: Pods
    pods:
      metric:
        name: httpserver_requests_qps
      target:
        averageValue: 50
        type: AverageValue
```

以下のコマンドを実行して業務にストレステストを行い、自動スケーリングされたかどうかを観察します。次のとおりです。



```
$ kubectl get hpa
NAME         REFERENCE                TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
httpserver   Deployment/httpserver   83933m/50   1         1000      2          18h
$ kubectl get pods
NAME                          READY   STATUS              RESTARTS   AGE
httpserver-6f94475d45-47d5w   1/1     Running             0          3m41s
httpserver-6f94475d45-7rln9   1/1     Running             0          37h
httpserver-6f94475d45-6c5xm   0/1     ContainerCreating   0          1s
httpserver-6f94475d45-wl78d   0/1     ContainerCreating   0          1s
```

正常にスケーリングされた場合、HPAが業務カスタム指標に基づいて自動スケーリングを行ったことを表します。

TKEでHPAを使用してサービスのAuto Scalingを実装します

最終更新日：：2023-04-28 15:30:11

概要

Kubernetes Pod水平自動スケーリング（Horizontal Pod Autoscaler、以下HPAと略称）はCPU使用率、メモリ使用率およびその他のカスタマイズされたメトリック指標に基づいてPodのレプリカ数量を自動スケーリングし、それによってワークロードサービスのメトリック全体のレベルとユーザーが設定した目標値を一致させます。ここではTKEのHPA機能を使用してPodの自動水平スケーリングを実現する方法についてご紹介します。

シナリオ

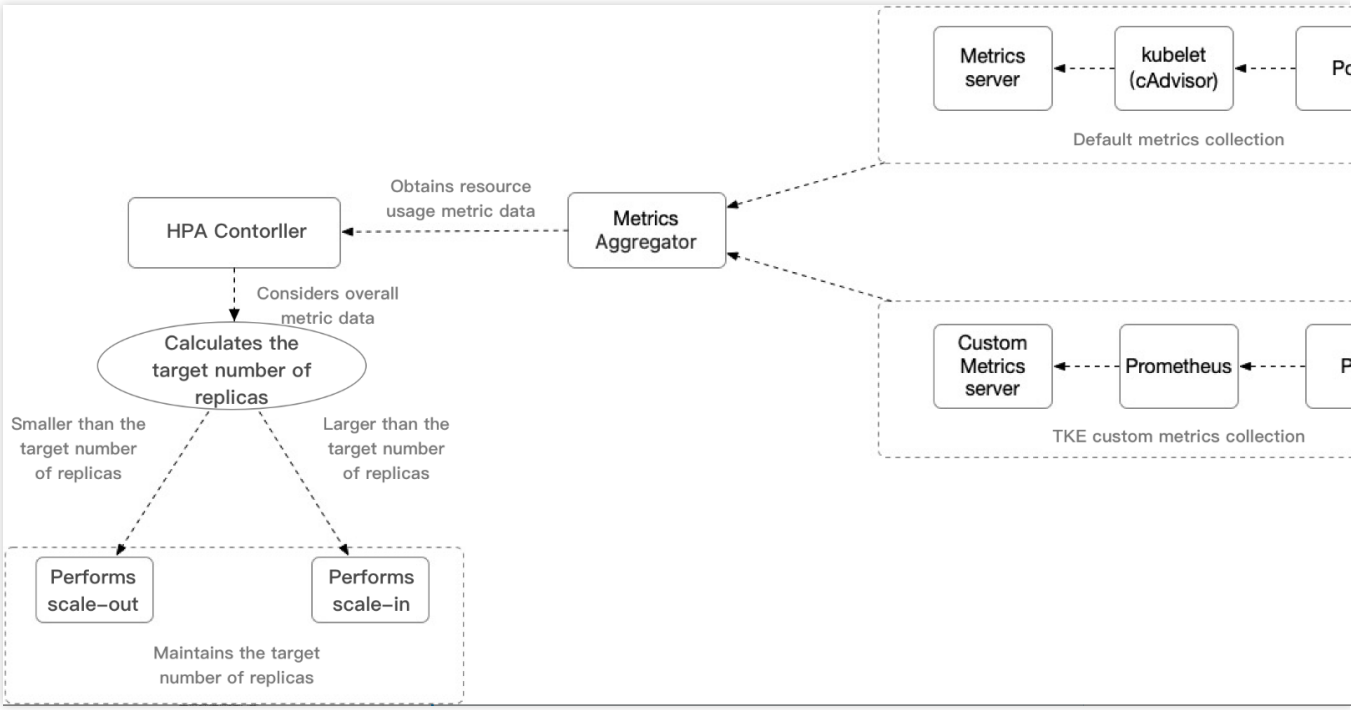
HPA自動スケーリング機能はTKEに非常に柔軟な適応型機能を持たせ、ユーザーが設定内で複数のPodレプリカを迅速にスケーリングして業務負荷の急激な増加に対応できるようにします。業務負荷が小さくなった状況でも実際の状況に応じて適切にスケーリングすることによってその他のサービスへの計算リソースを節約し、プロセス全体の自動化に人手を介する必要がありません。eコマースサービス、オンライン教育、金融サービスなどのようにサービス変動が大きく、サービス数が多く、かつスケーリングを頻繁に行う必要がある業務シナリオに適しています。

原理の概要

Pod水平自動スケーリング機能は Kubernetes APIリソースおよびコントローラによって実現します。リソースの利用指標によってコントローラの行動を決定し、コントローラは定期的にPodリソースの利用状況に基づいてサービスPodのレプリカ数を調整し、それによってワークロードのメトリックレベルとユーザーが設定した目標値を一致させます。そのスケーリングフローは下図に示すとおりです。

注意

Pod自動水平スケーリングはDaemonSetリソースのようにスケーリングできないオブジェクトには適用されません。



重要な内容の説明：

HPA Controller：HPA スケーリングロジックの制御コンポーネントを制御します。

Metrics Aggeregator：メトリック指標アグリゲーター。通常の状態で、コントローラは一連の集約API（`metrics.k8s.io`、`custom.metrics.k8s.io` および `external.metrics.k8s.io`）からメトリック値を取得します。`metrics.k8s.io` APIは通常Metricsサーバーによって提供され、コミュニティ版は基本的なCPU、メモリのメトリックタイプを提供することができます。コミュニティ版に比べ、TKEはカスタムMetrics Serverを使用して、より幅広いHPAをサポートすることができるメトリック指標トリガータイプを収集し、CPU、メモリ、ハードディスク、ネットワークおよびGPUを含む関連指標を提供することができます。より詳細な内容については、TKE自動スケーリング指標の説明をご参照ください。

説明

コントローラもHeapsterから指標を取得することができますが、Kubernetesバージョン1.11からは、Heapsterから指標特性を取得する方式は破棄されました。

HPAが目標レプリカ数を計算するアルゴリズム：TKE HPAスケーリングアルゴリズムについては動作原理をご参照ください。その他の詳細なアルゴリズムについては、アルゴリズム詳細をご参照ください。

前提条件

Tencent Cloudアカウントの登録済みであること。

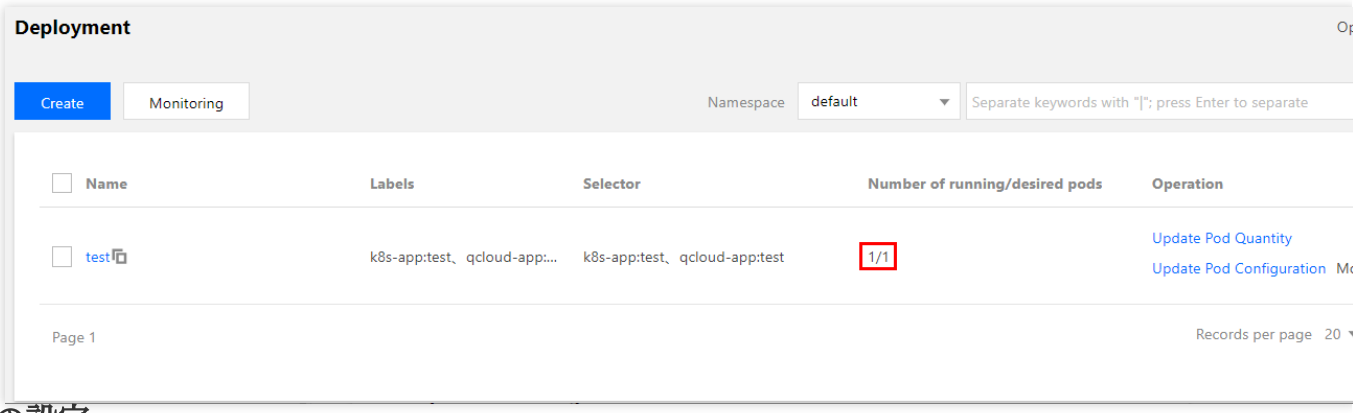
Tencent CloudTKEコンソールにログインしていること。

TKEクラスターを作成済みであること。クラスターの作成に関して、詳細については、クラスターの作成をご参照ください。

操作手順

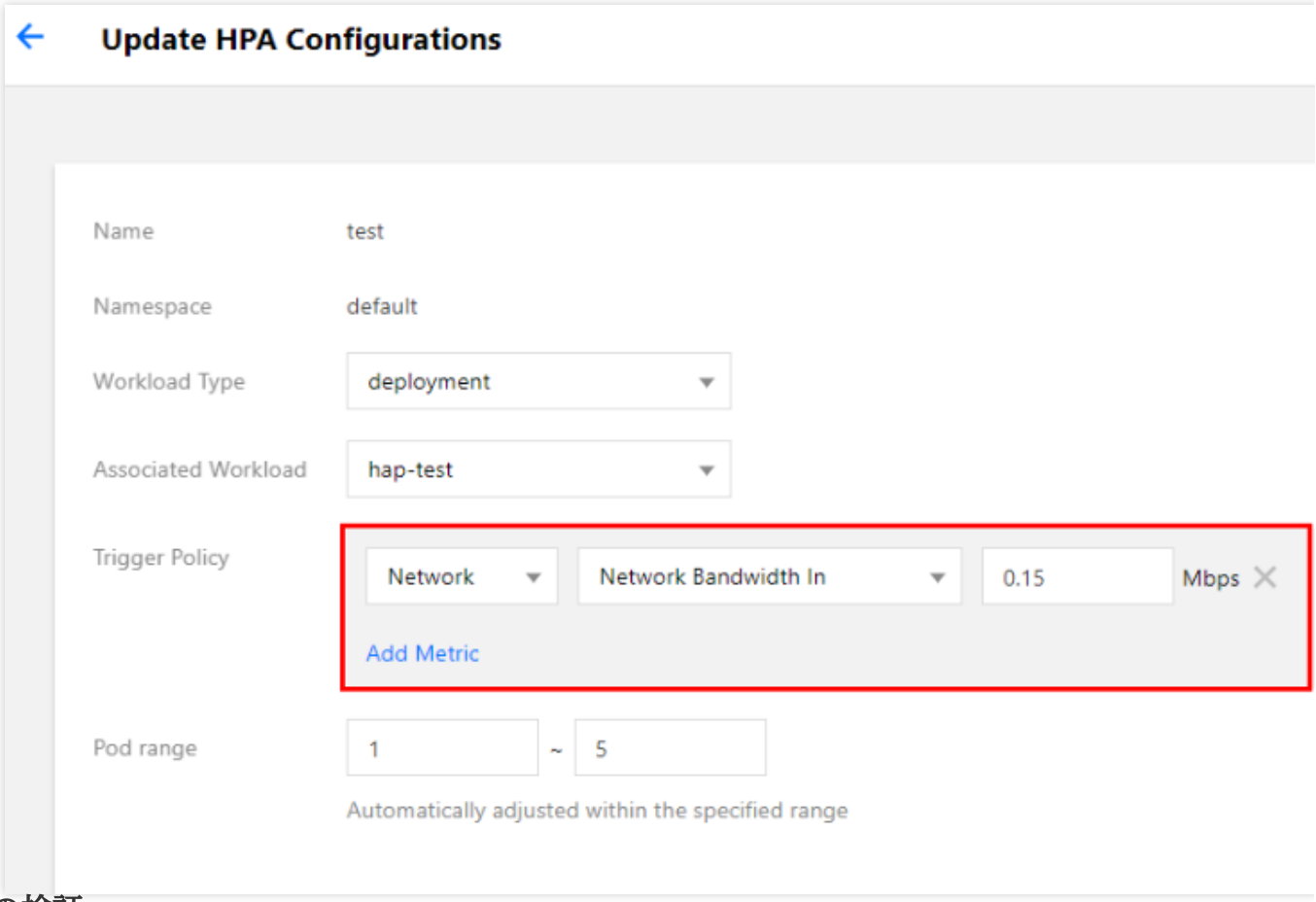
デプロイテストのワークロード

Deploymentリソースタイプのワークロードを例にとり、単一レプリカ数で、サービスタイプがWebサービスの「hpa-test」というワークロードを作成します。TKEコンソールでDeploymentタイプのワークロードを作成する方法はDeployment管理をご参照ください。本事例の作成結果は下図に示すとおりです（このドキュメントのスクリーンキャプチャ情報はコンソールの実際のインターフェースより遅れている可能性があるため、コンソールの実際の表示に準じます）。



HPAの設定

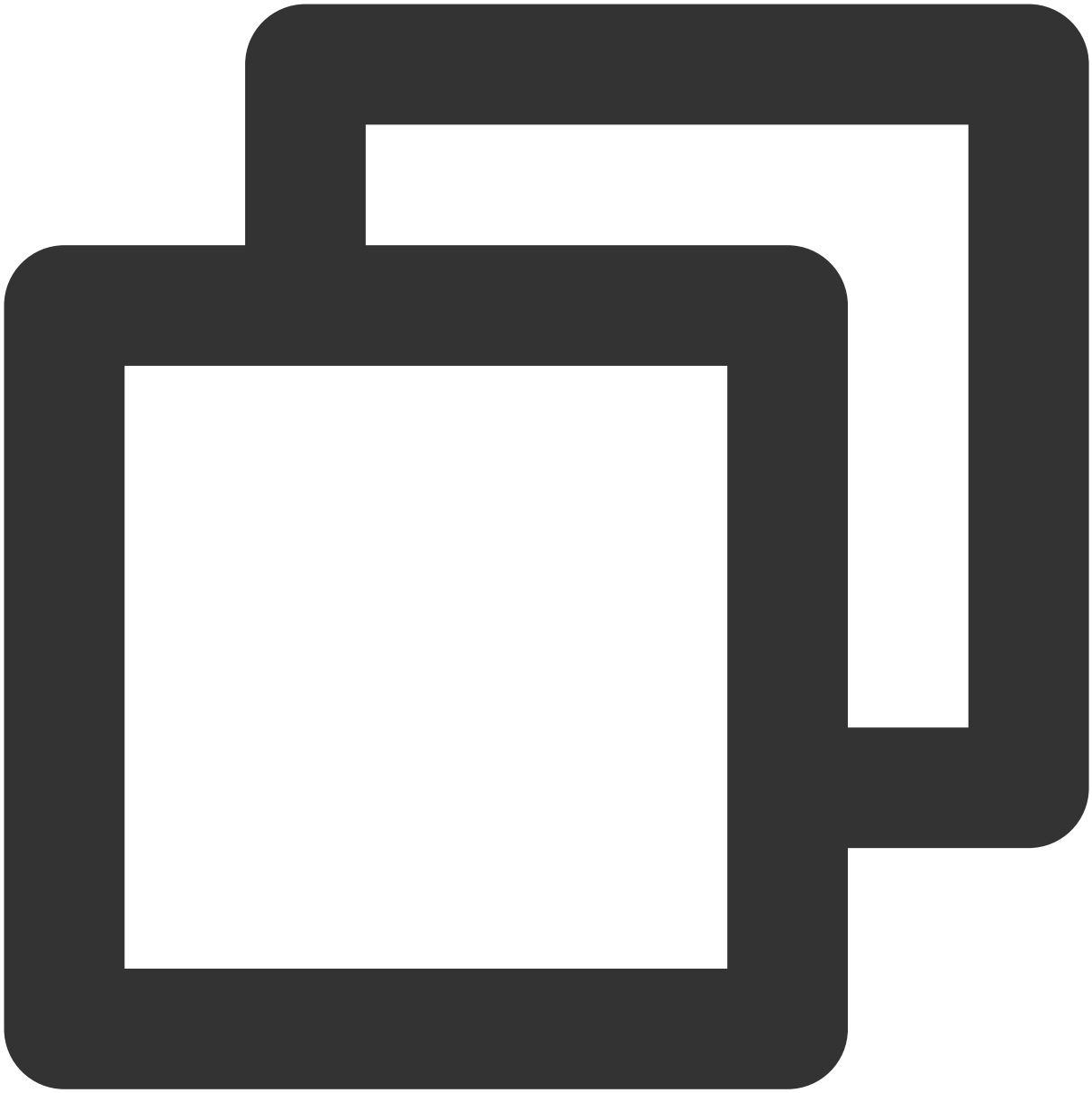
TKEコンソールでテストワークロードにHPA設定をバインドします。バインドの方法およびHPAの設定については、HPA操作ステップをご参照ください。このドキュメントではネットワークのアウトバンド帯域幅が0.15Mbps（150Kbps）に達した時にスケールアウトをトリガーするポリシーの設定を例にとります。下図に示すとおりです。



機能の検証

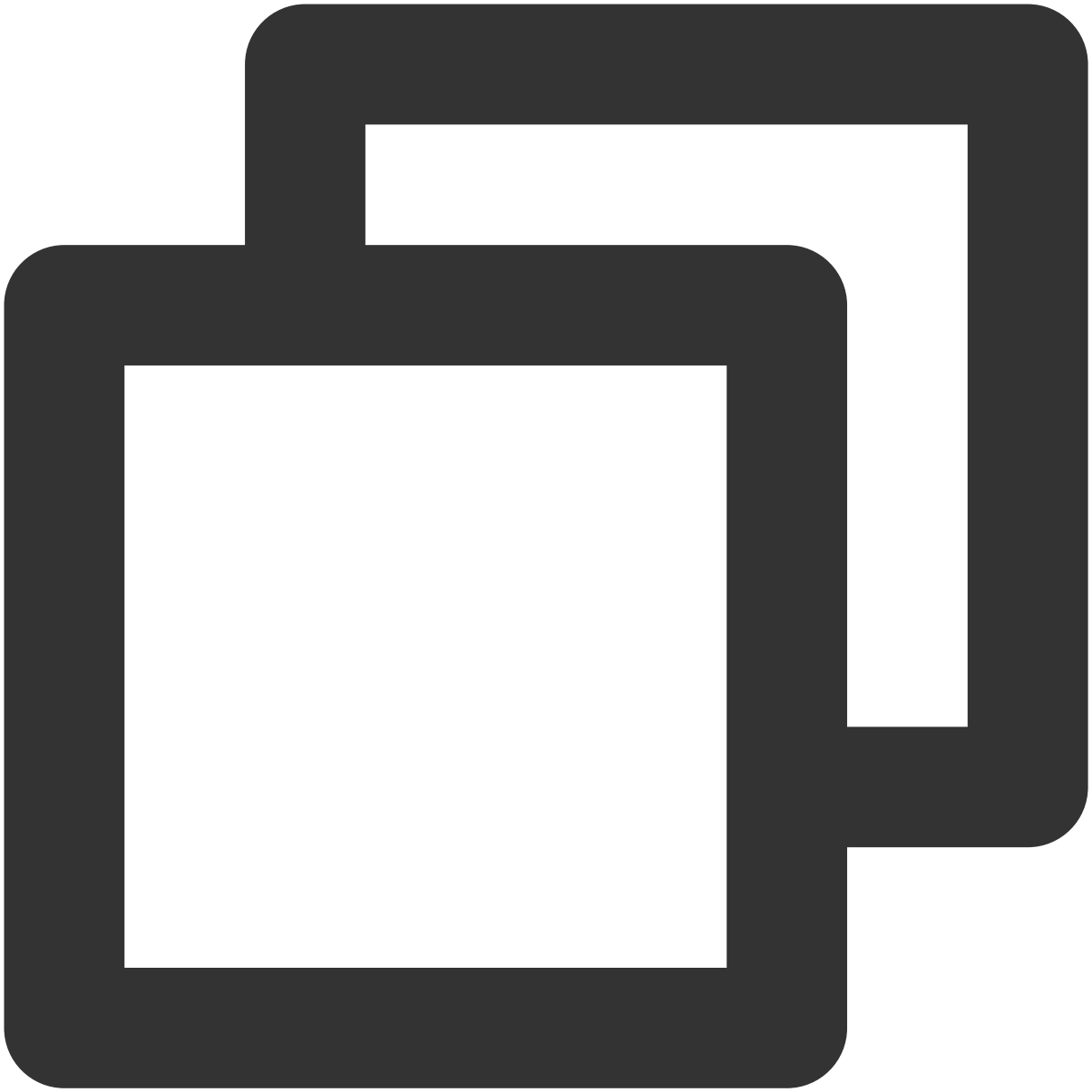
スケールアウトプロセスのシミュレーション

以下のコマンドを実行して、クラスター内で一時的にPodを起動して設定されたHPA機能にテストを行います（クライアントのシミュレーション）。



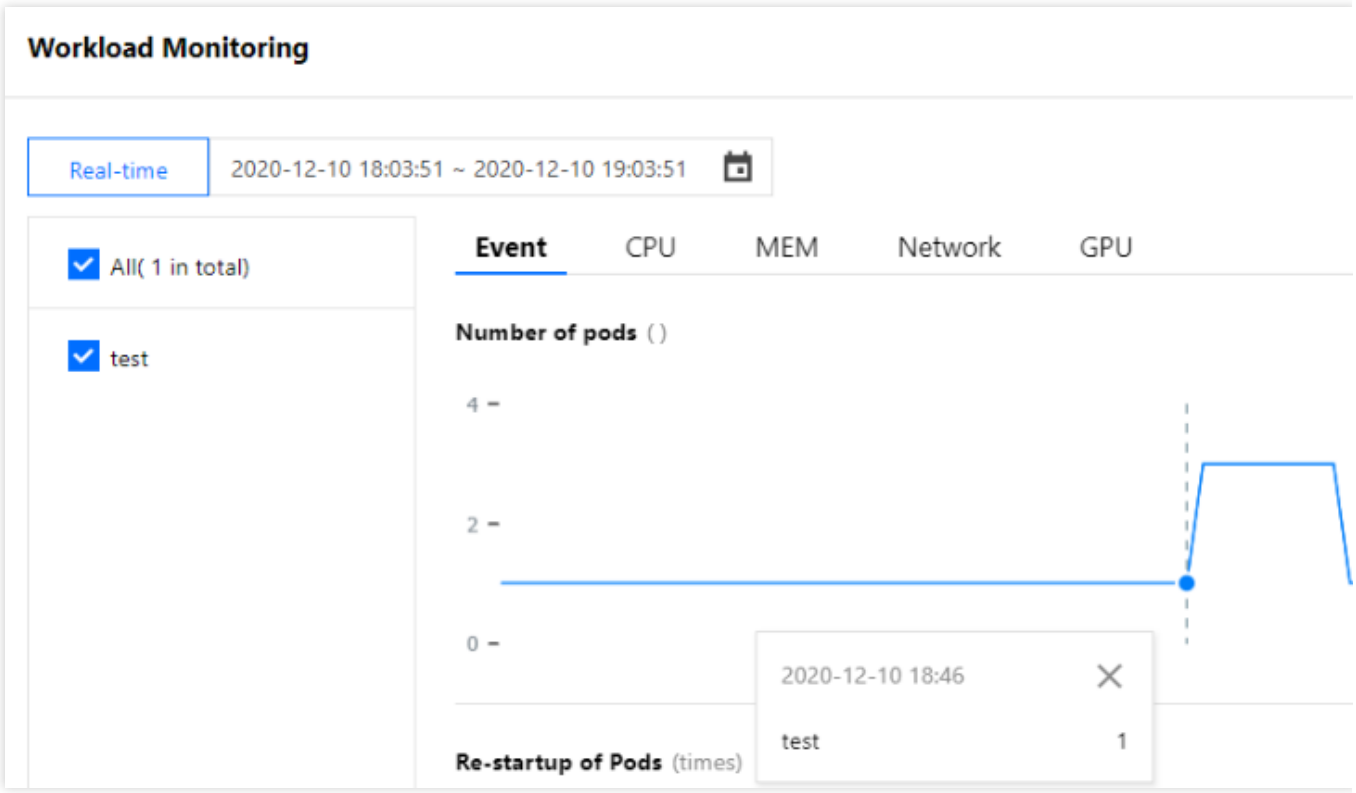
```
kubectl run -it --image alpine hpa-test --restart=Never --rm /bin/sh
```

一時的にPod内で以下のコマンドを実行して、短時間内に大量のリクエストで「hpa-test」というサービスにアクセスすることによってアウトバウンドトラフィックの帯域幅の増加をシミュレートします。



```
# hpa-test.default.svc.cluster.localはサービスのクラスター内でのドメイン名です。スクリプトを
while true; do wget -q -O - hpa-test.default.svc.cluster.local; done
```

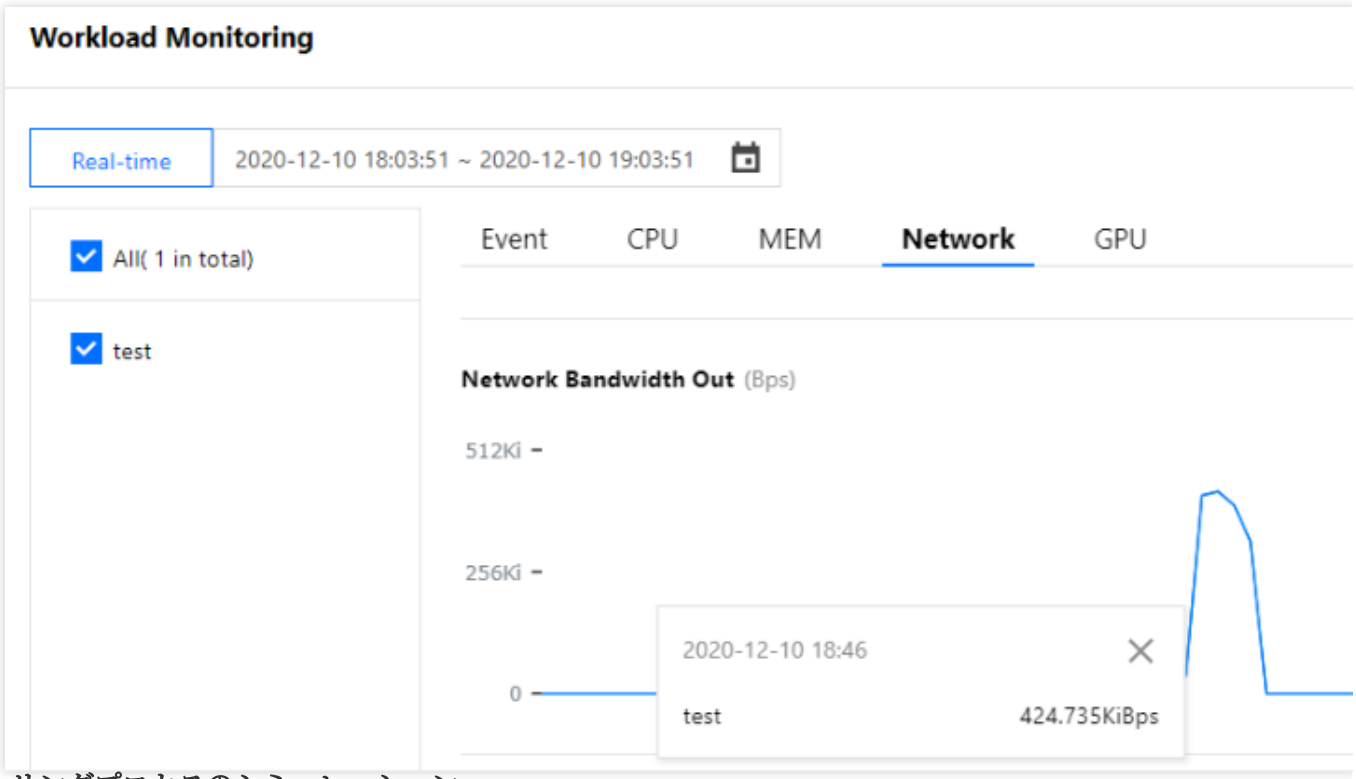
テストPod内でシミュレーションリクエストコマンドを実行した後、ワークロードのPod数を観察することによって監視したところ、16:21にワークロードはレプリカ数が2個にスケールアウトされていることがわかり、これによってHPAのスケールアウトイベントがトリガーされたことが推測できます。下図のように表示されます（このドキュメントのスクリーンキャプチャ情報はコンソール実際のインターフェースより遅れている可能性があるため、コンソールの実際の表示に準じます）。



さらにワークロードのネットワーク送信帯域幅の監視によって16:21にネットワーク送信帯域幅がおよそ196Kbpsに増加していることが観察でき、HPAによって設定されたネットワーク送信帯域幅目標値を超え、さらにこのときHPAスケーリングアルゴリズムがトリガーされたことを証明でき、1つのレプリカ数をスケールアウトすることによって設定された目標値を満たすため、ワークロードのレプリカ数は2個になります。下図のように表示されます（このドキュメントのスクリーンキャプチャ情報はコンソール実際のインターフェースより遅れている可能性があるため、コンソールの実際の表示に準じます）。

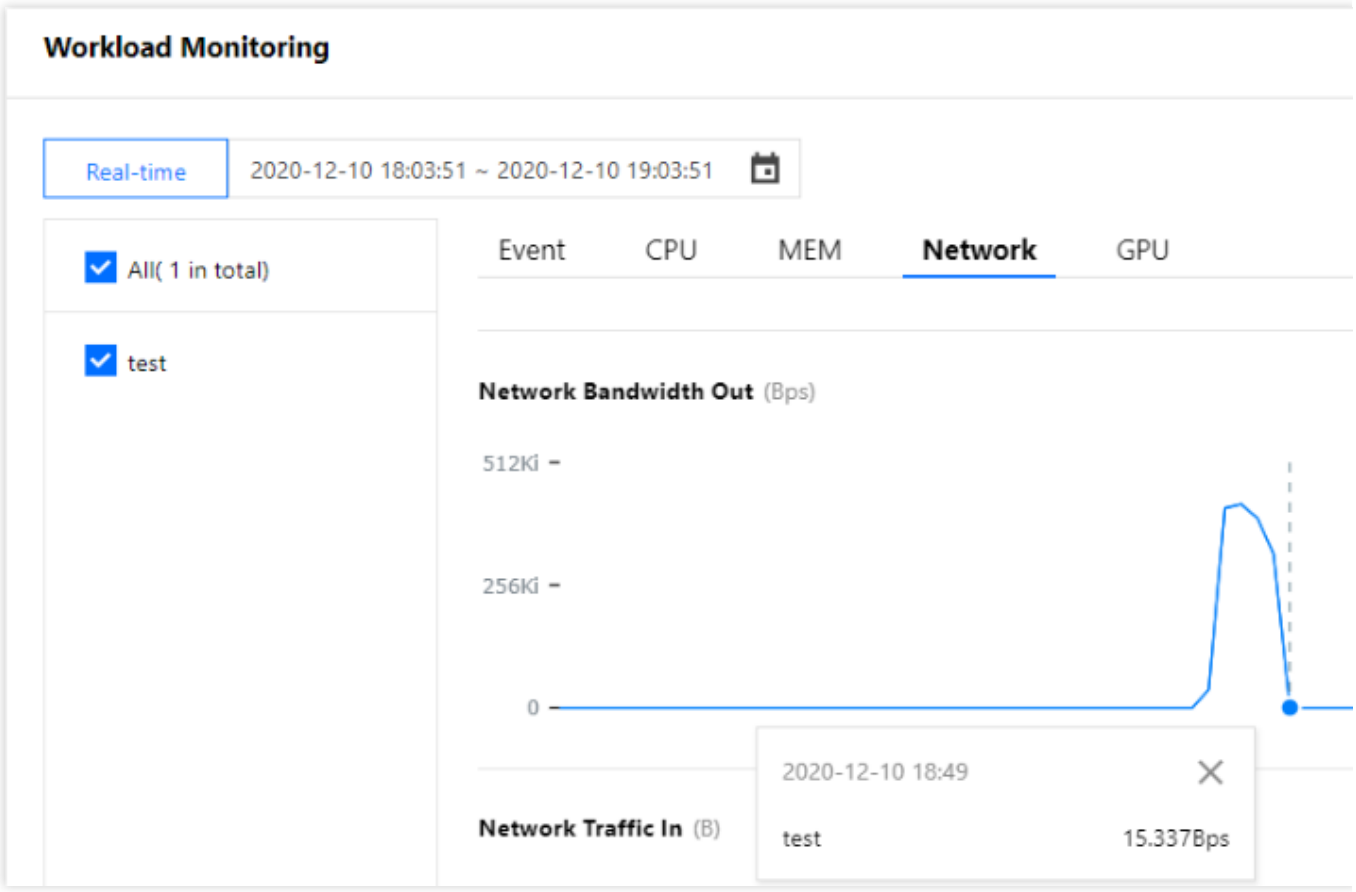
注意

HPAスケーリングアルゴリズムは公式でディメンションを計算してスケーリングロジックを制御するだけでなく、複数のディメンションでスケールアウトまたはスケーリングが必要かどうかを判断するため、実際の状況において予想からわずかにずれる可能性があります。詳細については、アルゴリズム詳細をご参照ください。

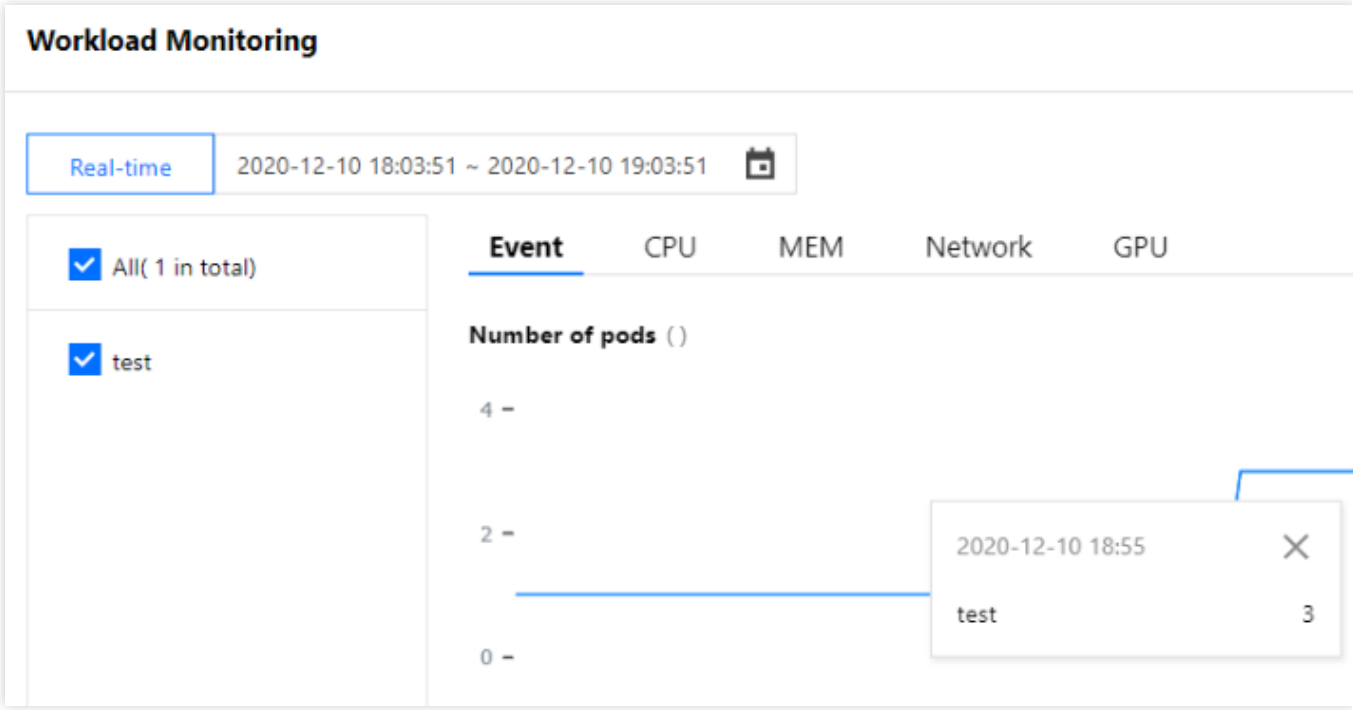


スケーリングプロセスのシミュレーション

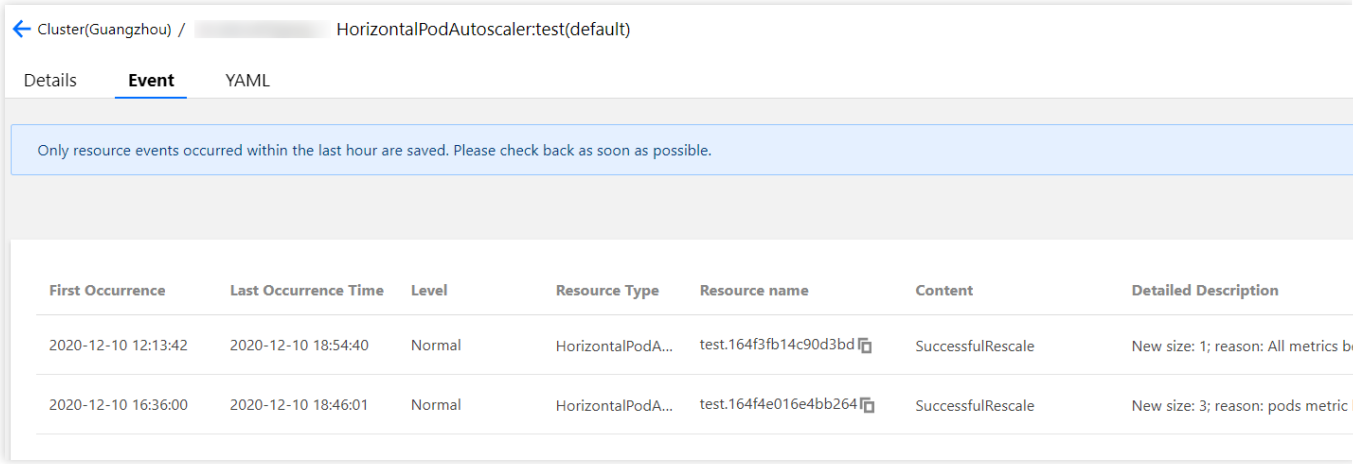
スケーリングプロセスのシミュレーション時、16:24前後に手動でシミュレーションリクエストを実行するコマンドを停止し、監視からこのときまでのネットワーク送信帯域幅値がスケールアウト前の位置に下降していることが観察できます。HPAのロジックによると、このときワークロードスケーリングの条件を満たしています。下図のように表示されます（このドキュメントのスクリーンキャプチャ情報はコンソール実際のインターフェースより遅れている可能性があるため、コンソールの実際の表示に準じます）。



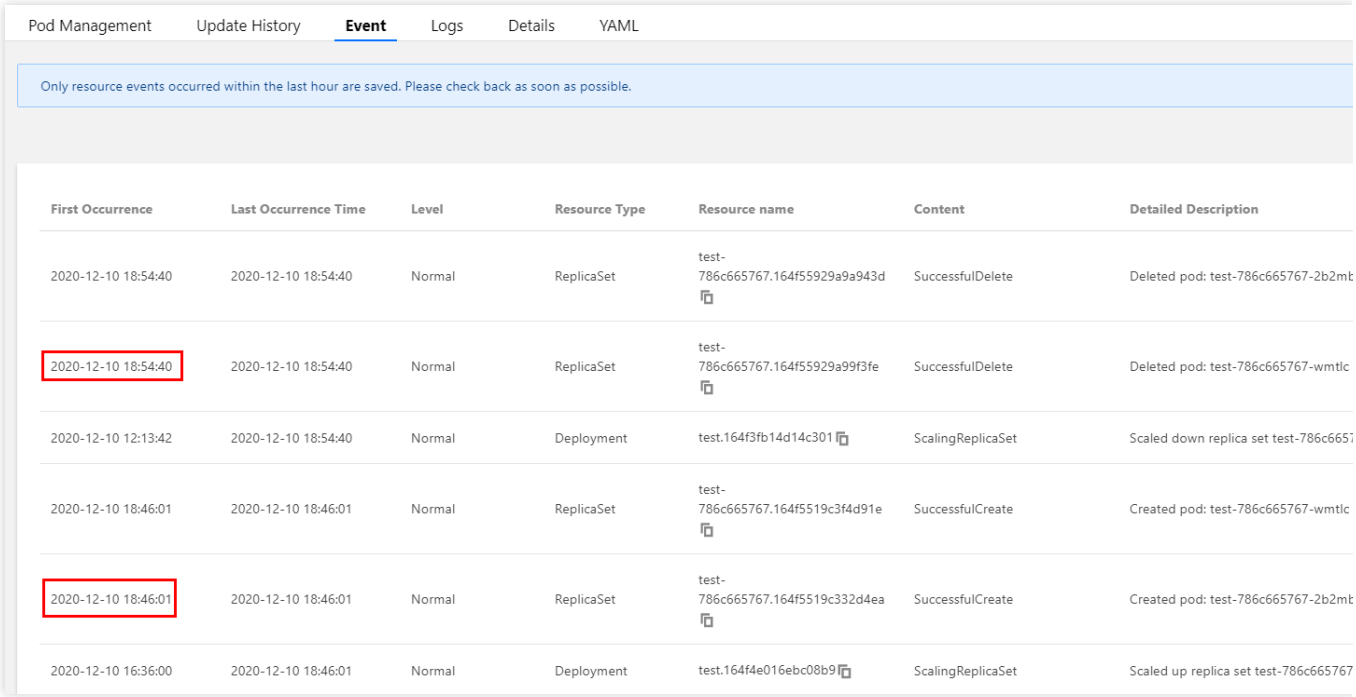
しかし下図のワークロードのPod数の監視からわかるように、ワークロードは16:30にHPAのスケーリングをトリガーします。原因はHPAスケーリングをトリガーした後デフォルトで5分間容認する時間アルゴリズムです。それによってメトリック指標が短時間で変動することによって引き起こされる頻繁なスケーリングを防止します。詳細については、クールダウン/遅延のサポートをご参照ください。下図からワークロードレプリカ数は停止コマンドの5分後にHPA スケーリングアルゴリズムによって最初に設定したレプリカ数にスケーリングされることがわかります。下図のように表示されます（このドキュメントのスクリーンキャプチャ情報はコンソール実際のインターフェースより遅れている可能性があるため、コンソールの実際の表示に準じます）。



TKEにHPAスケーリングイベントが発生した時、対応するHPAインスタンスのイベントリストに表示されます。イベント通知リストの時間はそれぞれ「初回の発生時間」および「最後の発生時間」であることにご注意ください。「初回の発生時間」は同じイベントが初めて発生した時間を表し、「最後の発生時間」は同じイベントが発生した最新の時間を表すため、下図のイベントリストの「最後の発生時間」フィールドからこの事例のスケールアウトイベント時点は16:21:03であることがわかります。スケーリングイベント時間は16.29:42で、時点とワークロードの監視で確認した時点と一致します。下図に示すとおりです。



また、ワークロードイベントリストもHPA発生時のワークロードのレプリカ数追加イベントを記録します。下図からワークロードスケーリング時点とHPAイベントリストの時点も一致し、レプリカ数が増加した時点が16:21:03、レプリカ数が減少した時点が16:29:42であることがわかります。



まとめ

この例では主にTKEのHPA機能をデモンストレーションし、TKEを使用してカスタマイズされたネットワーク送信帯域幅メトリックタイプをワークロードHPAのスケーリングメトリック指標とします。
ワークロードの実際のメトリック値がHPAによって設定されたメトリック目標値を超えた場合、HPAはスケールアウトアルゴリズムに基づいて適切なレプリカ数を計算して水平スケールアウトを実現し、ワークロードのメトリック指標が予測を満たすことおよびワークロードの健全で安定した動作を保証します。
実際のメトリック値がHPAによって設定されたメトリック目標値より大幅に低い場合、HPAは許容時間後に最適なレプリカ数を計算して水平スケーリングを実現し、アイドル状態のリソースを適切に解放し、リソース使用率を向上させるという目的を達成し、かつプロセス全体はHPAおよびワークロードのイベントリストにいずれも対応するイベント記録があり、それによってワークロードの水平スケーリングのプロセス全体を追跡可能にします。

ストレージ
CFS-Turboクラスのファイルシステムを静的にマウントします
TKE ServerlessでCFS-Turboを静的にマウントします

最終更新日：：2023-04-26 19:01:59

シナリオ

TKE ServerlessクラスターのCloud File Storage(CFS)TurboタイプのStorageをマウントします。このコンポーネントは、プライベートプロトコルに基づいてTencent Cloud CFS Turboファイルシステムをワークロードにマウントしており、現在は静的な設定のみをサポートしています。CFSストレージタイプの詳細については、Cloud Fileストレージタイプと性能仕様をご参照ください。

前提条件

TKE Serverlessクラスターを作成済みであり、クラスターのバージョンが>=1.14であること。

使用手順

ファイルシステムの作成

CFS Turboファイルシステムを作成する操作の詳細については、ファイルシステムの作成をご参照ください。

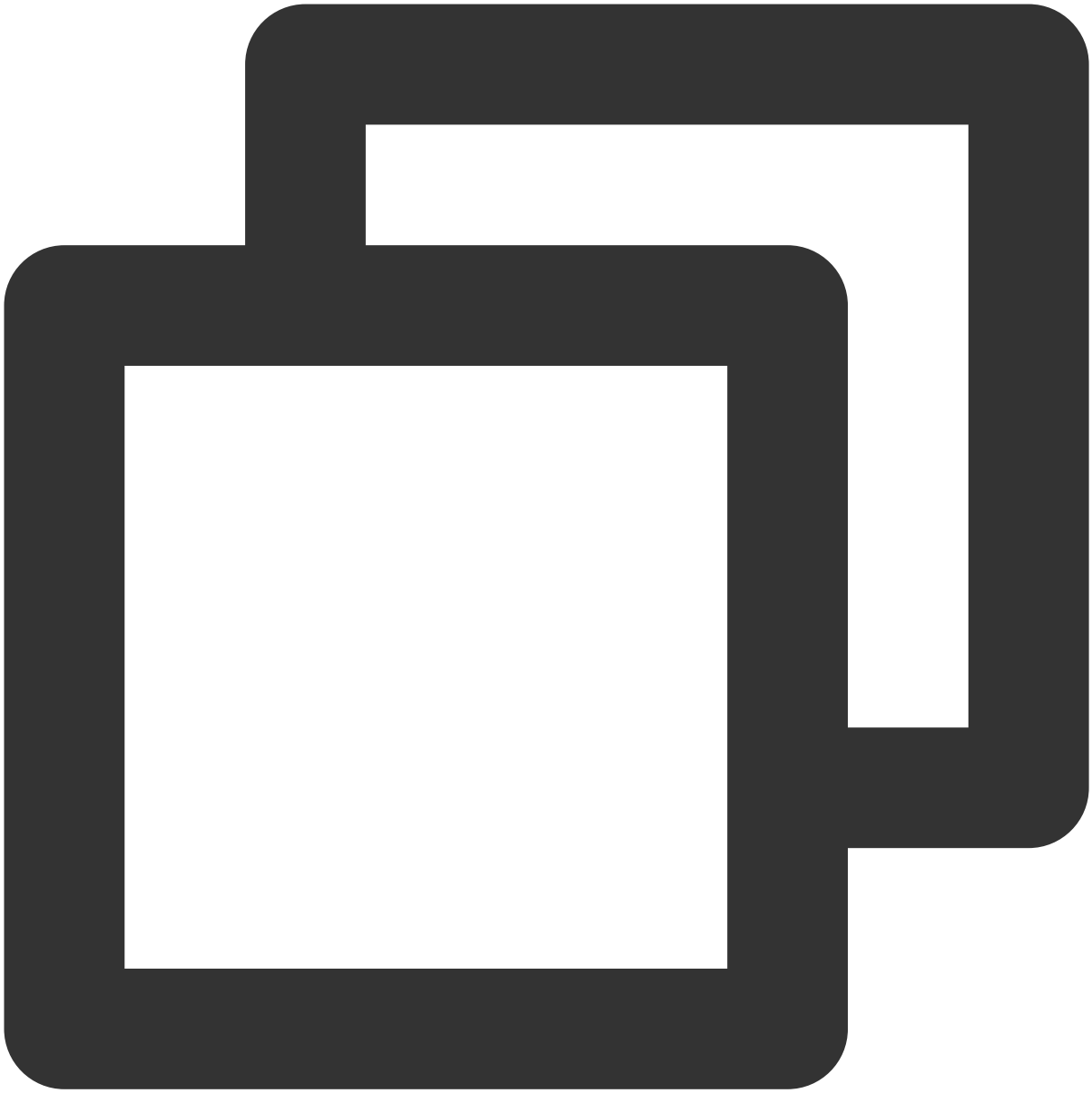
注意

ファイルシステムの作成後、クラスターネットワーク（vpc-xx）をファイルシステムのCCNにバインドする必要があります（ファイルシステムのマウントポイント情報で確認できます）。

Node Pluginのデプロイ

ステップ1：csidriver.yamlファイルの新規作成

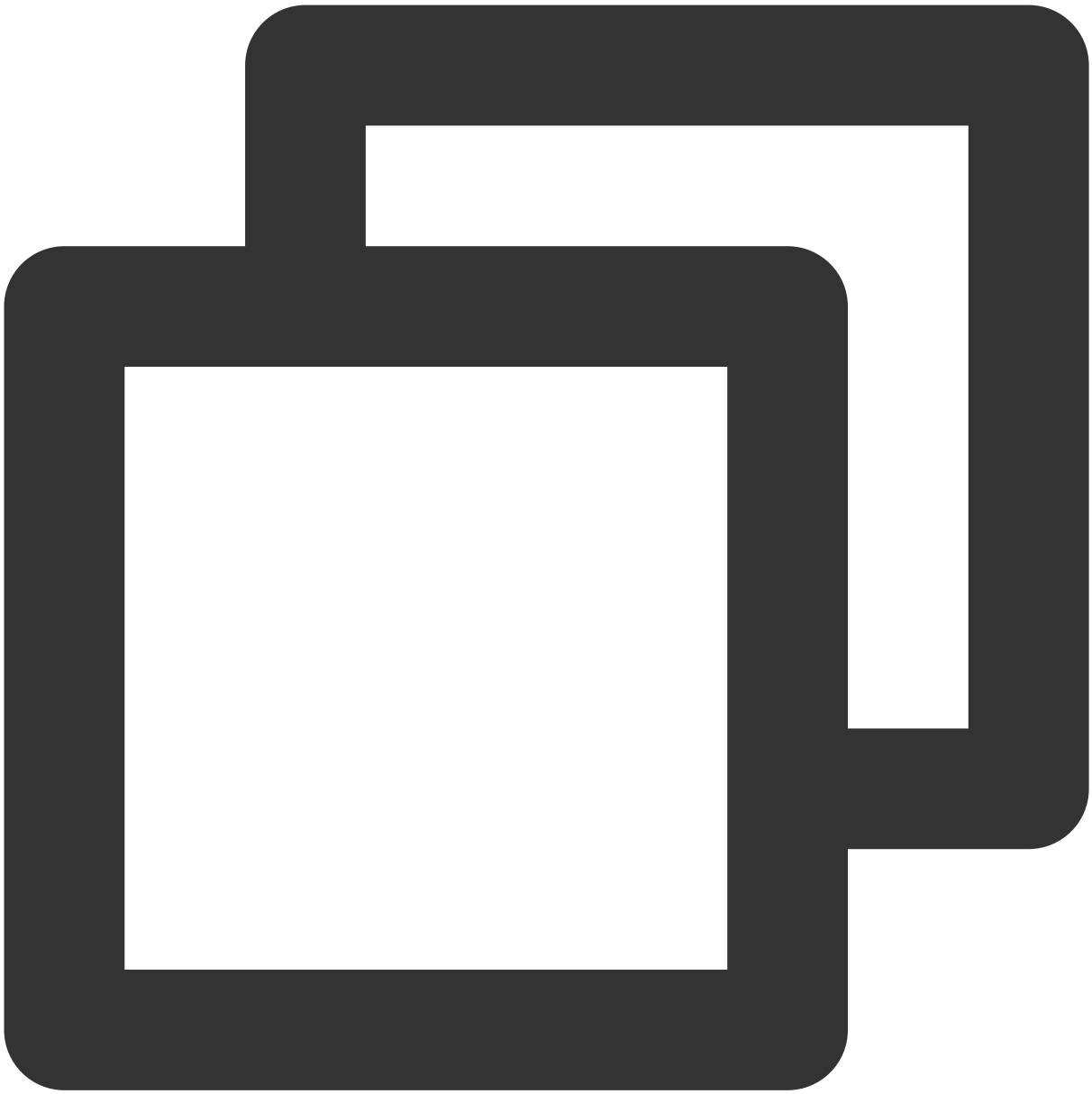
csidriver.yamlファイルの例は次のとおりです。



```
apiVersion: storage.k8s.io/v1
kind: CSIDriver
metadata:
  name: com.tencent.cloud.csi.cfsturbo
spec:
  attachRequired: false
  podInfoOnMount: false
```

ステップ2：csidriverの作成

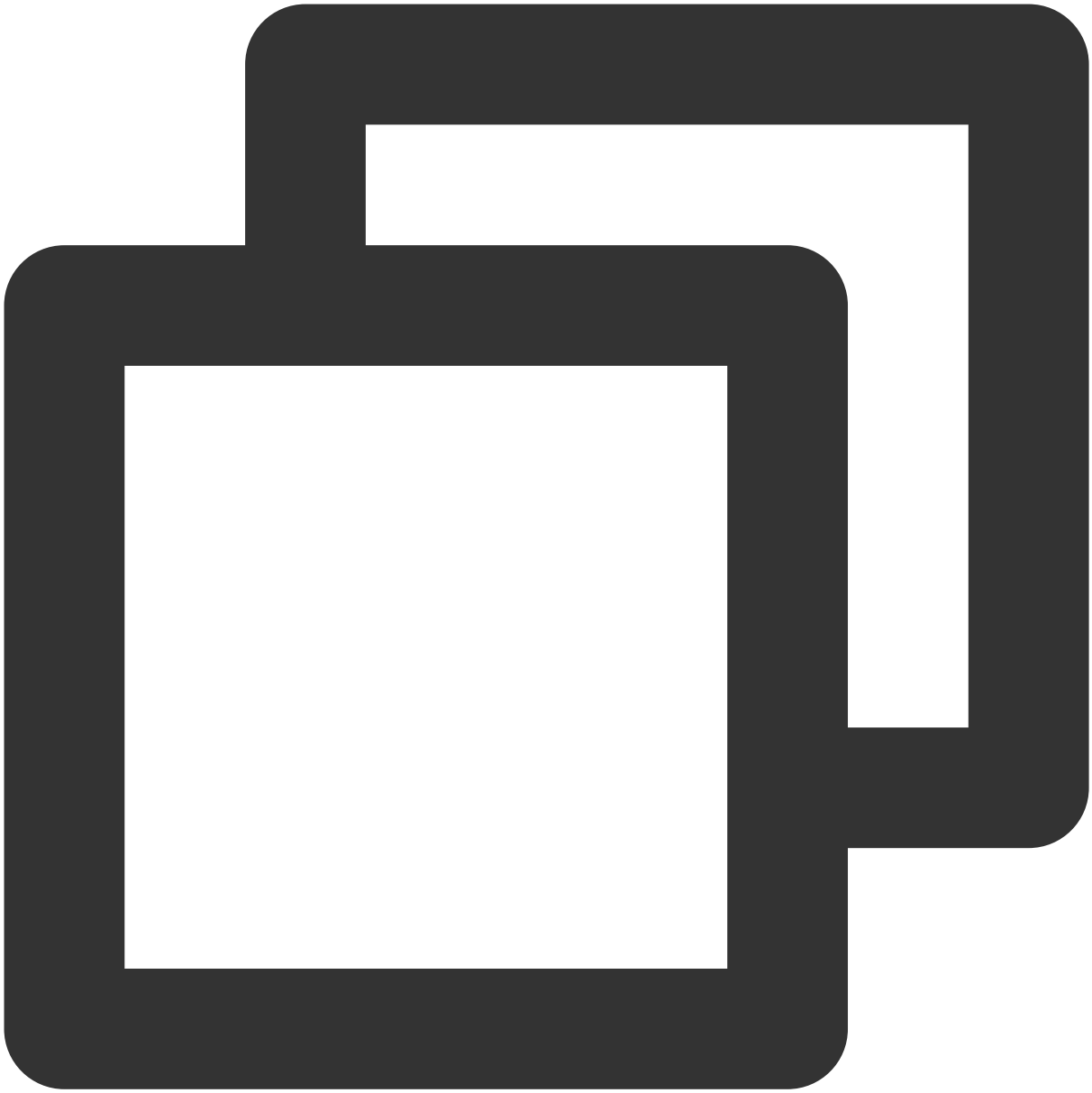
以下のコマンドを実行し、csidriverを作成します。



```
kubectl apply -f csidriver.yaml
```

CFS Turbo Storageボリュームの作成

ステップ1：以下のテンプレートを使用して、CFS TurboタイプPVを作成します



```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-cfsturbo
spec:
  accessModes:
  - ReadWriteMany
  capacity:
    storage: 10Gi
  csi:
    driver: com.tencent.cloud.csi.cfsturbo
```

```
    volumeHandle: pv-cfsturbo
    volumeAttributes:
      host: *.*.*.*
      fsid: ********
      # cfs turbo subPath
      path: /
  storageClassName: ""
```

パラメータの説明：

metadata.name：PV名を作成します。

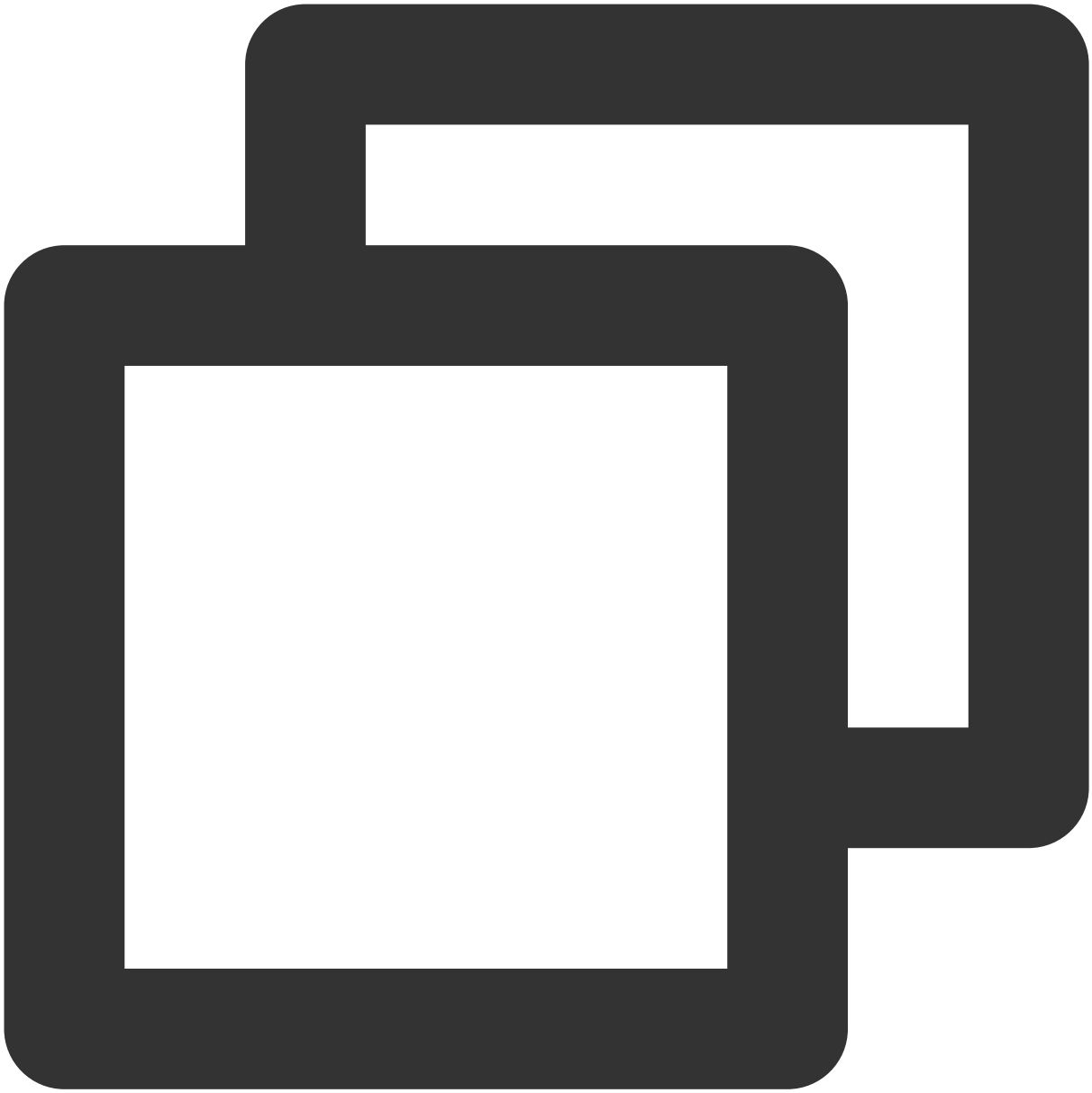
spec.csi.volumeHandle：PV名と同一にします。

spec.csi.volumeAttributes.host：ファイルシステムのIPアドレス。ファイルシステムのマウントポイント情報で確認できます。

spec.csi.volumeAttributes.fsid：ファイルシステムのfsid（ファイルシステムIDではない）。ファイルシステムのマウントポイント情報（マウントコマンドの `tcp0:/` と `/cfs` の間の文字列、下図に示すとおり）で確認できます。

spec.csi.volumeAttributes.path:ファイルシステムのサブディレクトリ。未入力の場合は `/` がデフォルトになります（マウントのパフォーマンスを向上させるため、プラグインバックエンドは実際に `/` ディレクトリの下に `/cfs` ディレクトリを配置します）。サブディレクトリを指定する必要がある場合は、ファイルシステム `/cfs` に存在することを確認する必要があります。マウント後、workloadはこのサブディレクトリの上位階層のディレクトリにアクセスできなくなります。例として、 `path: /test` の場合、ファイルシステムに `/cfs/test` ディレクトリが存在することを確認する必要があります。

ステップ2：以下のテンプレートを使用して、PVCを作成してPVをバインドします



```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-cfsturbo
spec:
  storageClassName: ""
  volumeName: pv-cfsturbo
  accessModes:
  - ReadWriteMany
  resources:
    requests:
```

```
      storage: 10Gi
```

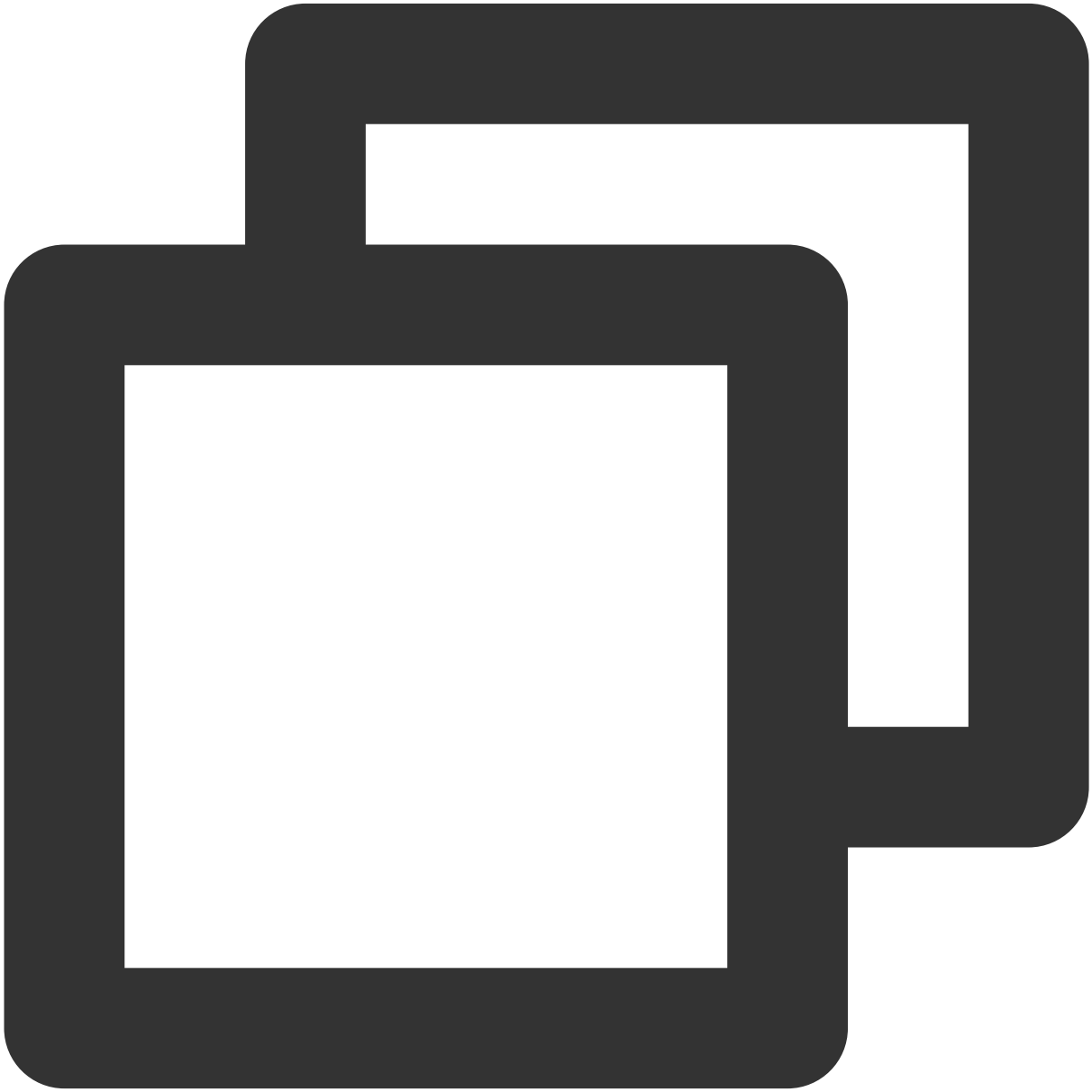
パラメータの説明：

metadata.name：PVC名を作成します。

spec.volumeName：ステップ1で作成したPV名と同一にします。

CFS Turbo Storageボリュームの使用

以下のテンプレートを使用して、Podを作成しPVCをマウントします。



```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: ccr.ccs.tencentyun.com/qcloud/nginx:1.9
    imagePullPolicy: Always
    name: nginx
    ports:
    - containerPort: 80
      protocol: TCP
    volumeMounts:
      - mountPath: /var/www
        name: data
  volumes:
  - name: data
    persistentVolumeClaim:
      claimName: pvc-cfsturbo
```