Red Hat OpenShift Serverless 1.32

Integrations

OpenShift Serverless と Service Mesh の統合、およびコスト管理サービスとの統合

Last Updated: 2024-06-03

Red Hat OpenShift Serverless 1.32 Integrations

OpenShift Serverless と Service Mesh の統合、およびコスト管理サービスとの統合

法律上の通知

概要

本書では、Service Mesh を OpenShift Serverless と統合する方法を説明します。また、コスト管理サービスを使用したコストの理解および追跡や、Serverless アプリケーションでの NVIDIA GPU リソースの使用についても説明します。

目次

第1章 サービスメッシュと OPENSHIFT SERVERLESS の統合

OpenShift Serverless Operator は、Knative のデフォルト Ingress として Kourier を提供します。ただし、Kourier が有効であるかどうかにかかわらず、OpenShift Serverless でサービスメッシュを使用できます。Kourier を無効にして統合すると、mTLS 機能など、Kourier イングレスがサポートしない追加のネットワークおよびルーティングオプションを設定できます。

次の前提条件と制限事項に注意してください。

- すべての Knative 内部コンポーネントと Knative Services は Service Mesh の一部であり、サイドカーインジェクションが有効になっています。これは、メッシュ全体で厳密な mTLS が適用されることを意味します。Knative Services へのすべてのリクエストには mTLS 接続が必要で、OpenShift Routing からの呼び出しを除き、クライアントは証明書を送信する必要があります。

- OpenShift Serverless と Service Mesh の統合では、**1つの** サービスメッシュのみをターゲットにできます。クラスター内には複数のメッシュが存在できますが、OpenShift Serverless はそのうちの1つでのみ使用できます。

- OpenShift Serverless が含まれているターゲット **ServiceMeshMemberRoll** の変更、つまり OpenShift Serverless を別のメッシュに移動することはサポートされていません。ターゲットの Service Mesh を変更する唯一の方法は、OpenShift Serverless をアンインストールして再インストールすることです。

1.1. 前提条件

- クラスター管理者アクセス権を持つ Red Hat OpenShift Serverless アカウントにアクセスできる。

- OpenShift CLI (**oc**) がインストールされている。

- Serverless Operator がインストールされている。

- Red Hat OpenShift Service Mesh Operator がインストールされている。

- 以下の手順の例では、ドメイン **example.com** を使用します。このドメインの証明書のサンプルは、サブドメイン証明書に署名する認証局 (CA) として使用されます。
  お使いのデプロイメントでこの手順を完了し、検証するには、一般に信頼されているパブリック CA によって署名された証明書、または組織が提供する CA のいずれかが必要です。コマンドの例は、ドメイン、サブドメイン、および CA に合わせて調整する必要があります。

- ワイルドカード証明書を OpenShift Container Platform クラスターのドメインに一致するように設定する必要があります。たとえば、OpenShift Container Platform コンソールアドレスが **https://console-openshift-console.apps.openshift.example.com** の場合は、ドメインが ***.apps.openshift.example.com** になるようにワイルドカード証明書を設定する必要があります。ワイルドカード証明書の設定に関する詳細は、**着信外部トラフィックを暗号化する証明書の作成**のトピックを参照してください。

- デフォルトの OpenShift Container Platform クラスタードメインのサブドメインではないものを含むドメイン名を使用する必要がある場合は、これらのドメインのドメインマッピングを設定する必要があります。詳細は、OpenShift Serverless ドキュメントのカスタムドメインマッピングの作成を参照してください。

重要

OpenShift Serverless は、本書で明示的に文書化されている Red Hat OpenShift Service Mesh 機能の使用のみをサポートし、文書化されていない他の機能はサポートしません。

Service Mesh での Serverless 1.31 の使用は、Service Mesh バージョン 2.2 以降でのみサポートされます。1.31 以外のバージョンの詳細と情報は、「Red Hat OpenShift Serverless でサポートされる構成」ページを参照してください。

1.2. 関連情報

- Red Hat OpenShift Serverless でサポートされる設定
- Kourier と Istio Ingress

1.3. 着信外部トラフィックを暗号化する証明書の作成

デフォルトでは、サービスメッシュ mTLS 機能は、Ingress ゲートウェイとサイドカーを持つ個々の Pod 間で、サービスメッシュ自体内のトラフィックのみを保護します。OpenShift Container Platform クラスターに流入するトラフィックを暗号化するには、OpenShift Serverless とサービスメッシュの統合を有効にする前に証明書を生成する必要があります。

前提条件

- OpenShift Container Platform に対するクラスター管理者権限があるか、Red Hat OpenShift Service on AWS または OpenShift Dedicated に対するクラスターまたは専用管理者権限がある。
- OpenShift Serverless Operator および Knative Serving がインストールされている。
- OpenShift CLI (**oc**) がインストールされている。
- アプリケーションおよび他のワークロードを作成するために、プロジェクトを作成しているか、適切なロールおよびパーミッションが割り当てられたプロジェクトにアクセスできる。

手順

1. Knative サービスの証明書に署名する root 証明書と秘密鍵を作成します。

```
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 \
    -subj '/O=Example Inc./CN=example.com' \
    -keyout root.key \
    -out root.crt
```

2. ワイルドカード証明書を作成します。

```
$ openssl req -nodes -newkey rsa:2048 \
    -subj "/CN=*.apps.openshift.example.com/O=Example Inc." \
    -keyout wildcard.key \
    -out wildcard.csr
```

3. ワイルドカード証明書を署名します。

```
$ openssl x509 -req -days 365 -set_serial 0 \
```

```
    -CA root.crt \
    -CAkey root.key \
    -in wildcard.csr \
    -out wildcard.crt
```

4. ワイルドカード証明書を使用してシークレットを作成します。

```
$ oc create -n istio-system secret tls wildcard-certs \
    --key=wildcard.key \
    --cert=wildcard.crt
```

この証明書は、OpenShift Serverless をサービスメッシュと統合する際に作成されるゲートウェイによって取得され、Ingress ゲートウェイはこの証明書でトラフィックを提供します。

1.4. サービスメッシュと OPENSHIFT SERVERLESS の統合

1.4.1. インストールの前提条件の確認

Service Mesh と Serverless の統合をインストールして設定する前に、前提条件が満たされていることを確認してください。

手順

1. 競合するゲートウェイを確認します。

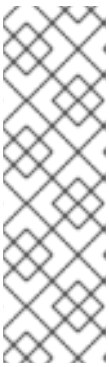
コマンドの例

```
$ oc get gateway -A -o jsonpath='{range .items[*]}{@.metadata.namespace}{"/"}
{@.metadata.name}{" "}{@.spec.servers}{"\n"}{end}' | column -t
```

出力例

```
knative-serving/knative-ingress-gateway  [{"hosts":["*"],"port":
{"name":"https","number":443,"protocol":"HTTPS"},"tls":{"credentialName":"wildcard-
certs","mode":"SIMPLE"}}]
knative-serving/knative-local-gateway    [{"hosts":["*"],"port":
{"name":"http","number":8081,"protocol":"HTTP"}}]
```

このコマンドは、**knative-serving** 内の **Gateways** と別の Service Mesh インスタンスの一部である **Gateways** を除き、**port: 443** と **hosts: ["*"]** をバインドする **Gateway** を返してはなりません。



注記

Serverless が属するメッシュは個別である必要があり、できれば Serverless ワークロード専用に予約されている必要があります。これは、**Gateways** などの追加設定が Serverless ゲートウェイ **knative-local-gateway** および **knative-ingress-gateway** に干渉する可能性があるためです。Red Hat OpenShift Service Mesh では、1 つのゲートウェイのみが同じポート (**port: 443**) 上でワイルドカードホストバインディング (**hosts: ["*"]**) を要求できます。別のゲートウェイがすでにこの設定をバインドしている場合は、Serverless ワークロード用に別のメッシュを作成する必要があります。

2. Red Hat OpenShift Service Mesh の **istio-ingressgateway** がタイプ **NodePort** または **LoadBalancer** として公開されているかどうかを確認します。

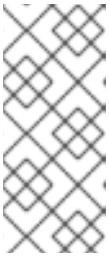
コマンドの例

```
$ oc get svc -A | grep istio-ingressgateway
```

出力例

```
istio-system   istio-ingressgateway  ClusterIP  172.30.46.146 none>
15021/TCP,80/TCP,443/TCP     9m50s
```

このコマンドは、タイプ **NodePort** または **LoadBalancer** の **Service** オブジェクトを返しません。



注記

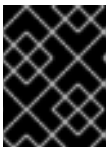
クラスターの外部 Knative サービスは、OpenShift Route を使用して OpenShift Ingress 経由で呼び出されることが想定されています。**NodePort** または **LoadBalancer** タイプの **Service** オブジェクトを使用して **istio-ingressgateway** を公開するなど、Service Mesh に直接アクセスすることはサポートされていません。

1.4.2. Service Mesh のインストールと設定

Serverless を Service Mesh と統合するには、特定の設定で Service Mesh をインストールする必要があります。

手順

1. 次の設定で **istio-system** namespace に **ServiceMeshControlPlane** リソースを作成します。



重要

既存の **ServiceMeshControlPlane** オブジェクトがある場合は、同じ設定が適用されていることを確認してください。

```
apiVersion: maistra.io/v2
kind: ServiceMeshControlPlane
metadata:
  name: basic
  namespace: istio-system
spec:
  profiles:
  - default
  security:
    dataPlane:
      mtls: true (1)
  techPreview:
    meshConfig:
      defaultConfig:
        terminationDrainDuration: 35s (2)
  gateways:
```

```
    ingress:
      service:
        metadata:
          labels:
            knative: ingressgateway 3
  proxy:
    networking:
      trafficControl:
        inbound:
          excludedPorts: 4
          - 8444 # metrics
          - 8022 # serving: wait-for-drain k8s pre-stop hook
```

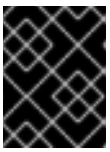
1. メッシュ内で厳密な mTLS を強制します。有効なクライアント証明書を使用した呼び出しのみが許可されます。
2. Serverless では、Knative サービスの正常な終了は 30 秒です。**istio-proxy** は、リクエストが破棄されないように、より長い終了時間を設定する必要があります。
3. Knative ゲートウェイのみをターゲットとする Ingress ゲートウェイの特定のセレクターを定義します。
4. これらのポートは、Kubernetes およびクラスター監視によって呼び出されます。これらはメッシュの一部ではないため、mTLS を使用して呼び出すことはできません。したがって、これらのポートはメッシュから除外されます。

2. サービスメッシュと統合する必要のある namespace をメンバーとして **ServiceMeshMemberRoll** オブジェクトに追加します。

servicemesh-member-roll.yaml 設定ファイルの例

```
apiVersion: maistra.io/v1
kind: ServiceMeshMemberRoll
metadata:
  name: default
  namespace: istio-system
spec:
  members: 1
    - knative-serving
    - knative-eventing
    - your-OpenShift-projects
```

1. サービスメッシュと統合する namespace の一覧。



重要

この namespace のリストには、**knative-serving** namespace と **knative-eventing** namespace が含まれている必要があります。

3. **ServiceMeshMemberRoll** リソースを適用します。

```
$ oc apply -f servicemesh-member-roll.yaml
```

4. サービスメッシュがトラフィックを受け入れることができるように、必要なゲートウェイを作成します。次の例では、**ISTIO_MUTUAL** モード (mTLS) で **knative-local-gateway** オブジェクトを使用します。

istio-knative-gateways.yaml 設定ファイルの例

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: knative-ingress-gateway
  namespace: knative-serving
spec:
  selector:
    knative: ingressgateway
  servers:
    - port:
        number: 443
        name: https
        protocol: HTTPS
      hosts:
        - "*"
      tls:
        mode: SIMPLE
        credentialName: <wildcard_certs> (1)
---
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
 name: knative-local-gateway
 namespace: knative-serving
spec:
 selector:
   knative: ingressgateway
 servers:
   - port:
       number: 8081
       name: https
       protocol: HTTPS (2)
     tls:
       mode: ISTIO_MUTUAL (3)
     hosts:
       - "*"
---
apiVersion: v1
kind: Service
metadata:
 name: knative-local-gateway
 namespace: istio-system
 labels:
   experimental.istio.io/disable-gateway-port-translation: "true"
spec:
 type: ClusterIP
 selector:
   istio: ingressgateway
 ports:
```

```
  - name: http2
    port: 80
    targetPort: 8081
```

1 ワイルドカード証明書を含むシークレットの名前。

2 3 **knative-local-gateway** オブジェクトは HTTPS トラフィックを処理し、すべてのクライアントが mTLS を使用してリクエストを送信することを期待します。これは、Service Mesh 内からのトラフィックのみが可能であることを意味します。Service Mesh の外部からのワークロードは、OpenShift Routing 経由で外部ドメインを使用する必要があります。

5. **Gateway** リソースを適用します。

```
$ oc apply -f istio-knative-gateways.yaml
```

1.4.3. Serverless のインストールと設定

Service Mesh をインストールした後、特定の設定で Serverless をインストールする必要があります。

手順

1. 次の **KnativeServing** カスタムリソースを使用して Knative Serving をインストールします。これにより、Istio 統合が有効になります。

   knative-serving-config.yaml 設定ファイルの例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  ingress:
    istio:
      enabled: true 1
  deployments: 2
  - name: activator
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  - name: autoscaler
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  config:
    istio: 3
      gateway.knative-serving.knative-ingress-gateway: istio-ingressgateway.<your-istio-namespace>.svc.cluster.local
      local-gateway.knative-serving.knative-local-gateway: knative-local-gateway.<your-istio-namespace>.svc.cluster.local
```

1 Istio 統合を有効にします。

2 Knative Serving データプレーン Pod のサイドカーの挿入を有効にします。

3 istio が **istio-system** namespace で実行されていない場合は、これら 2 つのフラグを正しい namespace で設定する必要があります。

2. **KnativeServing** リソースを適用します。

```
$ oc apply -f knative-serving-config.yaml
```

3. 次の **KnativeEventing** オブジェクトを使用して Knative Eventing をインストールします。これにより、Istio 統合が有効になります。

knative-eventing-config.yaml 設定ファイルの例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  config:
    features:
      istio: enabled 1
  workloads: 2
  - name: pingsource-mt-adapter
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  - name: imc-dispatcher
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  - name: mt-broker-ingress
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  - name: mt-broker-filter
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
```

1 Eventing Istio コントローラーが各 **InMemoryChannel** または **KafkaChannel** サービスの **DestinationRule** を作成できるようにします。

2 Knative Eventing Pod のサイドカーインジェクションを有効にします。

4. **KnativeEventing** リソースを適用します。

```
$ oc apply -f knative-eventing-config.yaml
```

5. 次の **KnativeKafka** カスタムリソースを使用して Knative Kafka をインストールします。これにより、Istio 統合が有効になります。

knative-kafka-config.yaml 設定ファイルの例

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  name: knative-kafka
  namespace: knative-eventing
spec:
  channel:
    enabled: true
    bootstrapServers: <bootstrap_servers> 1
  source:
    enabled: true
  broker:
    enabled: true
    defaultConfig:
      bootstrapServers: <bootstrap_servers> 2
      numPartitions: <num_partitions>
      replicationFactor: <replication_factor>
    sink:
      enabled: true
  workloads: 3
  - name: kafka-controller
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  - name: kafka-broker-receiver
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  - name: kafka-broker-dispatcher
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  - name: kafka-channel-receiver
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  - name: kafka-channel-dispatcher
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  - name: kafka-source-dispatcher
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  - name: kafka-sink-receiver
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
```

1 2 Apache Kafka クラスター URL (例: **my-cluster-kafka-bootstrap.kafka:9092**)。

3 Knative Kafka Pod のサイドカーインジェクションを有効にします。

6. **KnativeEventing** オブジェクトを適用します。

```
$ oc apply -f knative-kafka-config.yaml
```

7. **ServiceEntry** をインストールして、**KnativeKafka** コンポーネントと Apache Kafka クラスターとの間の通信を Service Mesh に通知します。

kafka-cluster-serviceentry.yaml 設定ファイルの例

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: kafka-cluster
  namespace: knative-eventing
spec:
  hosts: (1)
    - <bootstrap_servers_without_port>
  exportTo:
    - "."
  ports: (2)
    - number: 9092
      name: tcp-plain
      protocol: TCP
    - number: 9093
      name: tcp-tls
      protocol: TCP
    - number: 9094
      name: tcp-sasl-tls
      protocol: TCP
    - number: 9095
      name: tcp-sasl-tls
      protocol: TCP
    - number: 9096
      name: tcp-tls
      protocol: TCP
  location: MESH_EXTERNAL
  resolution: NONE
```

(1) Apache Kafka クラスターホストのリスト (例: **my-cluster Kafka -bootstrap.kafka**)。

(2) Apache Kafka クラスターリスナーポート。

注記

spec.ports にリストされているポートは、TPC ポートの例です。実際の値は、Apache Kafka クラスターの設定方法によって異なります。

8. **ServiceEntry** リソースを適用します。

```
$ oc apply -f kafka-cluster-serviceentry.yaml
```

1.4.4. 統合の検証

Istio を有効にして Service Mesh と Serverless をインストールした後、統合が機能することを確認できます。

手順

1. サイドカー挿入が有効で、パススルールートを使用する Knative サービスを作成します。

   knative-service.yaml 設定ファイルの例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: <service_name>
  namespace: <namespace> 1
  annotations:
    serving.knative.openshift.io/enablePassthrough: "true" 2
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true" 3
        sidecar.istio.io/rewriteAppHTTPProbers: "true"
    spec:
      containers:
      - image: <image_url>
```

   1 サービスメッシュメンバーロールの一部である namespace。

   2 生成した証明書が Ingress ゲートウェイ経由で直接提供されるように、パススルー対応ルートを生成するように Knative Serving に指示します。

   3 Service Mesh サイドカーは Knative サービス Pod に挿入します。

   重要

   Service Mesh で動作するようにするには、この例のアノテーションをすべての Knative Service に必ず追加してください。

2. **Service** リソースを適用します。

```
$ oc apply -f knative-service.yaml
```

3. CA によって信頼されるようになった安全な接続を使用して、Serverless アプリケーションにアクセスします。

```
$ curl --cacert root.crt <service_url>
```

   たとえば、以下を実行します。

   コマンドの例

```
$ curl --cacert root.crt https://hello-default.apps.openshift.example.com
```

出力例

```
Hello Openshift!
```

1.5. MTLS で SERVICE MESH を使用する場合の KNATIVE SERVING メトリックの有効化

サービスメッシュが mTLS で有効にされている場合は、サービスメッシュが Prometheus のメトリクスの収集を阻止するため、Knative Serving のメトリクスはデフォルトで無効にされます。このセクションでは、Service Mesh および mTLS を使用する際に Knative Serving メトリックを有効にする方法を説明します。

前提条件

- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- mTLS 機能を有効にして Red Hat OpenShift Service Mesh をインストールしている。
- OpenShift Container Platform に対するクラスター管理者権限があるか、Red Hat OpenShift Service on AWS または OpenShift Dedicated に対するクラスターまたは専用管理者権限がある。
- OpenShift CLI (**oc**) がインストールされている。
- アプリケーションおよび他のワークロードを作成するために、プロジェクトを作成しているか、適切なロールおよびパーミッションが割り当てられたプロジェクトにアクセスできる。

手順

1. **prometheus** を Knative Serving カスタムリソース (CR) の **observability** 仕様で **metrics.backend-destination** として指定します。

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    observability:
      metrics.backend-destination: "prometheus"
...
```

この手順により、メトリックがデフォルトで無効になることを防ぎます。

2. 以下のネットワークポリシーを適用して、Prometheus namespace からのトラフィックを許可します。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-monitoring-ns
  namespace: knative-serving
spec:
```

```
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: "openshift-monitoring"
    podSelector: {}
...
```

3. **istio-system** namespace のデフォルトのサービスメッシュコントロールプレーンを変更して再適用し、以下の仕様が含まれるようにします。

```
...
spec:
  proxy:
    networking:
      trafficControl:
        inbound:
          excludedPorts:
          - 8444
...
```

1.6. KOURIER が有効にされている場合のサービスメッシュの OPENSHIFT SERVERLESS との統合

Kourier がすでに有効になっている場合でも、OpenShift Serverless で Service Mesh を使用できます。この手順は、Kourier を有効にして Knative Serving をすでにインストールしているが、後で Service Mesh 統合を追加することにした場合に役立つ可能性があります。

前提条件

- OpenShift Container Platform に対するクラスター管理者権限があるか、Red Hat OpenShift Service on AWS または OpenShift Dedicated に対するクラスターまたは専用管理者権限がある。
- アプリケーションおよび他のワークロードを作成するために、プロジェクトを作成しているか、適切なロールおよびパーミッションが割り当てられたプロジェクトにアクセスできる。
- OpenShift CLI (**oc**) がインストールされている。
- OpenShift Serverless Operator と Knative Serving をクラスターにインストールしている。
- Red Hat OpenShift Service Mesh をインストールしている。OpenShift Serverless with Service Mesh and Kourier は、Red Hat OpenShift Service Mesh バージョン 1.x および 2.x の両方での使用がサポートされています。

手順

1. サービスメッシュと統合する必要のある namespace をメンバーとして **ServiceMeshMemberRoll** オブジェクトに追加します。

```
apiVersion: maistra.io/v1
kind: ServiceMeshMemberRoll
metadata:
  name: default
```

```
  namespace: istio-system
spec:
  members:
    - <namespace> 1
...
```

1 サービスメッシュと統合する namespace の一覧。

2. **ServiceMeshMemberRoll** リソースを適用します。

```
$ oc apply -f <filename>
```

3. Knative システム Pod から Knative サービスへのトラフィックフローを許可するネットワークポリシーを作成します。

   a. サービスメッシュと統合する必要のある namespace ごとに、**NetworkPolicy** リソースを作成します。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-serving-system-namespace
  namespace: <namespace> 1
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          knative.openshift.io/part-of: "openshift-serverless"
  podSelector: {}
  policyTypes:
  - Ingress
...
```

1 サービスメッシュと統合する必要のある namespace を追加します。

注記

knative.openshift.io/part-of: "openshift-serverless" ラベルが OpenShift Serverless 1.22.0 で追加されました。OpenShift Serverless 1.21.1 以前を使用している場合は、**knative.openshift.io/part-of** ラベルを **knative-serving** および **knative-serving-ingress** namespace に追加します。

knative-serving namespace にラベルを追加します。

```
$ oc label namespace knative-serving knative.openshift.io/part-of=openshift-serverless
```

knative-serving-ingress namespace にラベルを追加します。

```
$ oc label namespace knative-serving-ingress knative.openshift.io/part-of=openshift-serverless
```

b. **NetworkPolicy** リソースを適用します。

```
$ oc apply -f <filename>
```

1.7. SERVICE MESH のシークレットフィルタリングを使用した NET-ISTIO のメモリー使用量の改善

デフォルトでは、Kubernetes **client-go** ライブラリーの informers の実装は、特定のタイプのすべてのリソースをフェッチします。これにより、多くのリソースが使用可能な場合にかなりのオーバーヘッドが発生する可能性があり、メモリーリークが原因で大規模なクラスターで Knative **net-istio** イングレスコントローラーが失敗する可能性があります。ただし、Knative **net-istio** Ingress コントローラーではフィルタリングメカニズムを使用できます。これにより、コントローラーは Knative 関連のシークレットのみを取得できます。

シークレットのフィルタリングは、OpenShift Serverless Operator 側でデフォルトで有効化されています。環境変数 **ENABLE_SECRET_INFORMER_FILTERING_BY_CERT_UID=true** は、デフォルトで **net-kourier** コントローラー Pod に追加されます。

重要

シークレットフィルタリングを有効にする場合は、すべてのシークレットに **networking.internal.knative.dev/certificate-uid: "<id>"** というラベルを付ける必要があります。そうしないと、Knative Serving がシークレットを検出しないため、障害が発生します。新規および既存のシークレットの両方にラベルを付ける必要があります。

前提条件

- OpenShift Container Platform に対するクラスター管理者権限があるか、Red Hat OpenShift Service on AWS または OpenShift Dedicated に対するクラスターまたは専用管理者権限がある。
- アプリケーションおよび他のワークロードを作成するために、プロジェクトを作成しているか、適切なロールおよびパーミッションが割り当てられたプロジェクトにアクセスできる。
- Red Hat OpenShift Service Mesh をインストールしている。OpenShift Serverless with Service Mesh は、Red Hat OpenShift Service Mesh バージョン 2.0.5 以降での使用でのみサポートされます。
- OpenShift Serverless Operator および Knative Serving をインストールしている。
- OpenShift CLI (**oc**) がインストールされている。

KnativeServing カスタムリソース (CR) の **workloads** フィールドを使用して、**ENABLE_SECRET_INFORMER_FILTERING_BY_CERT_UID** 変数を **false** に設定することで、シークレットフィルターを無効化できます。

KnativeServing CR の例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
```

```
...
  workloads:
    - env:
        - container: controller
          envVars:
            - name: ENABLE_SECRET_INFORMER_FILTERING_BY_CERT_UID
              value: 'false'
      name: net-istio-controller
```

第2章 SERVICE MESH を使用して OPENSHIFT SERVERLESS でネットワークトラフィックを分離する

重要

Service Mesh を使用して OpenShift Serverless でネットワークトラフィックを分離することは、テクノロジープレビューのみの機能です。テクノロジープレビュー機能は、Red Hat 製品サポートのサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではない場合があります。Red Hat は、実稼働環境でこれらを使用することを推奨していません。テクノロジープレビューの機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

Service Mesh を使用すると、Service Mesh **AuthorizationPolicy** リソースを使用して、共有 Red Hat OpenShift Serverless クラスター上のテナント間のネットワークトラフィックを分離できます。Serverless では、いくつかの Service Mesh リソースを使用して、これを活用することもできます。テナントは、共有クラスター上のネットワークを介して相互にアクセスできる 1 つまたは複数のプロジェクトのグループです。

2.1. 前提条件

- クラスター管理者アクセス権を持つ Red Hat OpenShift Serverless アカウントにアクセスできる。
- Service Mesh と Serverless の統合がセットアップされている。
- 各テナントに対して 1 つ以上の OpenShift プロジェクトを作成している。

2.2. 高レベルのアーキテクチャー

Service Mesh によって提供される Serverless トラフィック分離の高レベルアーキテクチャーは、**knative-serving**、**knative-eventing**、およびテナントの namespace 内の **AuthorizationPolicy** オブジェクトで設定され、すべてのコンポーネントは Service Mesh の一部です。挿入された Service Mesh サイドカーは、これらのルールを強制して、テナント間のネットワークトラフィックを分離します。

2.3. SERVICE MESH の保護

認可ポリシーと mTLS により、Service Mesh を保護できます。

手順

1. テナントのすべての Red Hat OpenShift Serverless プロジェクトがメンバーとして同じ **ServiceMeshMemberRoll** オブジェクトの一部であることを確認します。

```
apiVersion: maistra.io/v1
kind: ServiceMeshMemberRoll
metadata:
 name: default
```

```
 namespace: istio-system
spec:
 members:
   - knative-serving    # static value, needs to be here, see setup page
   - knative-eventing   # static value, needs to be here, see setup page
   - team-alpha-1       # example OpenShift project that belongs to the team-alpha tenant
   - team-alpha-2       # example OpenShift project that belongs th the team-alpha tenant
   - team-bravo-1       # example OpenShift project that belongs to the team-bravo tenant
   - team-bravo-2       # example OpenShift project that belongs th the team-bravo tenant
```

メッシュの一部であるすべてのプロジェクトは、mTLS を厳密モードで強制する必要があります。これにより、Istio はクライアント証明書が存在する接続のみを受け入れるようになり、Service Mesh サイドカーが **AuthorizationPolicy** オブジェクトを使用して接続元を検証できるようになります。

2. **knative-serving** および **knative-eventing** namespace に **AuthorizationPolicy** オブジェクトを使用して設定を作成します。

knative-default-authz-policies.yaml 設定ファイルの例

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-all-by-default
  namespace: knative-eventing
spec: { }
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: deny-all-by-default
  namespace: knative-serving
spec: { }
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-mt-channel-based-broker-ingress-to-imc-dispatcher
  namespace: knative-eventing
spec:
  action: ALLOW
  selector:
    matchLabels:
      app.kubernetes.io/component: "imc-dispatcher"
  rules:
    - from:
        - source:
            namespaces: [ "knative-eventing" ]
            principals: [ "cluster.local/ns/knative-eventing/sa/mt-broker-ingress" ]
      to:
        - operation:
            methods: [ "POST" ]
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
```

```
  name: allow-mt-channel-based-broker-ingress-to-kafka-channel
  namespace: knative-eventing
spec:
  action: ALLOW
  selector:
    matchLabels:
      app.kubernetes.io/component: "kafka-channel-receiver"
  rules:
    - from:
        - source:
            namespaces: [ "knative-eventing" ]
            principals: [ "cluster.local/ns/knative-eventing/sa/mt-broker-ingress" ]
      to:
        - operation:
            methods: [ "POST" ]
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-kafka-channel-to-mt-channel-based-broker-filter
  namespace: knative-eventing
spec:
  action: ALLOW
  selector:
    matchLabels:
      app.kubernetes.io/component: "broker-filter"
  rules:
    - from:
        - source:
            namespaces: [ "knative-eventing" ]
            principals: [ "cluster.local/ns/knative-eventing/sa/knative-kafka-channel-data-plane" ]
      to:
        - operation:
            methods: [ "POST" ]
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-imc-to-mt-channel-based-broker-filter
  namespace: knative-eventing
spec:
  action: ALLOW
  selector:
    matchLabels:
      app.kubernetes.io/component: "broker-filter"
  rules:
    - from:
        - source:
            namespaces: [ "knative-eventing" ]
            principals: [ "cluster.local/ns/knative-eventing/sa/imc-dispatcher" ]
      to:
        - operation:
            methods: [ "POST" ]
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
```

```
metadata:
  name: allow-probe-kafka-broker-receiver
  namespace: knative-eventing
spec:
  action: ALLOW
  selector:
    matchLabels:
      app.kubernetes.io/component: "kafka-broker-receiver"
  rules:
    - from:
        - source:
            namespaces: [ "knative-eventing" ]
            principals: [ "cluster.local/ns/knative-eventing/sa/kafka-controller" ]
      to:
        - operation:
            methods: [ "GET" ]
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-probe-kafka-sink-receiver
  namespace: knative-eventing
spec:
  action: ALLOW
  selector:
    matchLabels:
      app.kubernetes.io/component: "kafka-sink-receiver"
  rules:
    - from:
        - source:
            namespaces: [ "knative-eventing" ]
            principals: [ "cluster.local/ns/knative-eventing/sa/kafka-controller" ]
      to:
        - operation:
            methods: [ "GET" ]
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-probe-kafka-channel-receiver
  namespace: knative-eventing
spec:
  action: ALLOW
  selector:
    matchLabels:
      app.kubernetes.io/component: "kafka-channel-receiver"
  rules:
    - from:
        - source:
            namespaces: [ "knative-eventing" ]
            principals: [ "cluster.local/ns/knative-eventing/sa/kafka-controller" ]
      to:
        - operation:
            methods: [ "GET" ]
---
apiVersion: security.istio.io/v1beta1
```

```
kind: AuthorizationPolicy
metadata:
  name: allow-traffic-to-activator
  namespace: knative-serving
spec:
  selector:
    matchLabels:
      app: activator
  action: ALLOW
  rules:
    - from:
        - source:
            namespaces: [ "knative-serving", "istio-system" ]
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-traffic-to-autoscaler
  namespace: knative-serving
spec:
  selector:
    matchLabels:
      app: autoscaler
  action: ALLOW
  rules:
    - from:
        - source:
            namespaces: [ "knative-serving" ]
```

これらのポリシーは、Serverless システムコンポーネント間のネットワーク通信のアクセスルールを制限します。具体的には、次のルールが適用されます。

- **knative-serving** および **knative-eventing** namespace で明示的に許可されていないすべてのトラフィックを拒否します。
- **istio-system** および **knative-serving** namespace からアクティベーターへのトラフィックを許可する
- **knative-serving** namespace からオートスケーラーへのトラフィックを許可する
- **knative-eventing** namespace で Apache Kafka コンポーネントの正常性プローブを許可する
- **knative-eventing** namespace でチャネルベースのブローカーの内部トラフィックを許可する

3. 認可ポリシー設定を適用します。

```
$ oc apply -f knative-default-authz-policies.yaml
```

4. どの OpenShift プロジェクトが相互に通信できるかを定義します。この通信のために、テナントのすべての OpenShift プロジェクトには以下が必要です。
   - テナントのプロジェクトへの直接受信トラフィックを制限する 1 つの **AuthorizationPolicy** オブジェクト

- **knative-serving** プロジェクトで実行する Serverless のアクティベータコンポーネントを使用して受信トラフィックを制限する 1 つの **AuthorizationPolicy** オブジェクト
- Kubernetes が Knative Services で **PreStopHooks** を呼び出すことを許可する 1 つの **AuthorizationPolicy** オブジェクト

これらのポリシーを手動で作成する代わりに、**helm** ユーティリティーをインストールし、各テナントに必要なリソースを作成します。

Helm ユーティリティーのインストール

```
$ helm repo add openshift-helm-charts https://charts.openshift.io/
```

team alpha 用のサンプル設定の作成

```
$ helm template openshift-helm-charts/redhat-knative-istio-authz --version 1.31.0 --set
"name=team-alpha" --set "namespaces={team-alpha-1,team-alpha-2}" > team-alpha.yaml
```

team bravo の設定例の作成

```
$ helm template openshift-helm-charts/redhat-knative-istio-authz --version 1.31.0 --set
"name=team-bravo" --set "namespaces={team-bravo-1,team-bravo-2}" > team-bravo.yaml
```

5. 認可ポリシー設定を適用します。

```
$ oc apply -f team-alpha.yaml team-bravo.yaml
```

2.4. 設定の確認

curl コマンドを使用して、ネットワークトラフィック分離の設定を確認できます。

注記

次の例では、2 つのテナントがあり、それぞれが 1 つの namespace を持ち、**ServiceMeshMemberRoll** オブジェクトのすべての部分が、**team-alpha.yaml** ファイルおよび **team-bravo.yaml** ファイル内のリソースで設定されていると想定しています。

手順

1. 両方のテナントの namespace に Knative Services をデプロイします。

team-alpha のコマンド例

```
$ kn service create test-webapp -n team-alpha-1 \
   --annotation-service serving.knative.openshift.io/enablePassthrough=true \
   --annotation-revision sidecar.istio.io/inject=true \
   --env RESPONSE="Hello Serverless" \
   --image docker.io/openshift/hello-openshift
```

team-bravo のコマンド例

```
$ kn service create test-webapp -n team-bravo-1 \
    --annotation-service serving.knative.openshift.io/enablePassthrough=true \
    --annotation-revision sidecar.istio.io/inject=true \
    --env RESPONSE="Hello Serverless" \
    --image docker.io/openshift/hello-openshift
```

あるいは、次の YAML 設定を使用します。

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: test-webapp
  namespace: team-alpha-1
  annotations:
    serving.knative.openshift.io/enablePassthrough: "true"
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: 'true'
    spec:
      containers:
        - image: docker.io/openshift/hello-openshift
          env:
            - name: RESPONSE
              value: "Hello Serverless!"
---
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: test-webapp
  namespace: team-bravo-1
  annotations:
    serving.knative.openshift.io/enablePassthrough: "true"
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: 'true'
    spec:
      containers:
        - image: docker.io/openshift/hello-openshift
          env:
            - name: RESPONSE
              value: "Hello Serverless!"
```

2. 接続をテストするために **curl** Pod をデプロイします。

```
$ cat <<EOF | oc apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: curl
  namespace: team-alpha-1
  labels:
```

```
    app: curl
spec:
  replicas: 1
  selector:
    matchLabels:
      app: curl
  template:
    metadata:
      labels:
        app: curl
      annotations:
        sidecar.istio.io/inject: 'true'
    spec:
      containers:
      - name: curl
        image: curlimages/curl
        command:
        - sleep
        - "3600"
EOF
```

3. **curl** コマンドを使用して設定を確認します。
許可されているクラスターのローカルドメインを介して **team-alpha-1** → **team-alpha-1** をテストします。

コマンドの例

```
$ oc exec deployment/curl -n team-alpha-1 -it -- curl -v http://test-webapp.team-alpha-1:80
```

出力例

```
HTTP/1.1 200 OK
content-length: 18
content-type: text/plain; charset=utf-8
date: Wed, 26 Jul 2023 12:49:59 GMT
server: envoy
x-envoy-upstream-service-time: 9

Hello Serverless!
```

許可されている外部ドメインを介して **team-alpha-1** から **team-alpha-1** への接続をテストします。

コマンドの例

```
$ EXTERNAL_URL=$(oc get ksvc -n team-alpha-1 test-webapp -o custom-
columns=:.status.url --no-headers) && \
oc exec deployment/curl -n team-alpha-1 -it -- curl -ik $EXTERNAL_URL
```

出力例

```
HTTP/2 200
content-length: 18
content-type: text/plain; charset=utf-8
```

```
date: Wed, 26 Jul 2023 12:55:30 GMT
server: istio-envoy
x-envoy-upstream-service-time: 3629

Hello Serverless!
```

クラスターのローカルドメインを介して **team-alpha-1** から **team-bravo-1** への接続をテストしますが、これは許可されていません。

コマンドの例

```
$ oc exec deployment/curl -n team-alpha-1 -it -- curl -v http://test-webapp.team-bravo-1:80
```

出力例

```
* processing: http://test-webapp.team-bravo-1:80
*   Trying 172.30.73.216:80...
* Connected to test-webapp.team-bravo-1 (172.30.73.216) port 80
> GET / HTTP/1.1
> Host: test-webapp.team-bravo-1
> User-Agent: curl/8.2.0
> Accept: */*
>
< HTTP/1.1 403 Forbidden
< content-length: 19
< content-type: text/plain
< date: Wed, 26 Jul 2023 12:55:49 GMT
< server: envoy
< x-envoy-upstream-service-time: 6
<
* Connection #0 to host test-webapp.team-bravo-1 left intact
RBAC: access denied
```

許可されている外部ドメインを介して、**team-alpha-1** から **team-bravo-1** への接続をテストします。

コマンドの例

```
$ EXTERNAL_URL=$(oc get ksvc -n team-bravo-1 test-webapp -o custom-
columns=:.status.url --no-headers) && \
oc exec deployment/curl -n team-alpha-1 -it -- curl -ik $EXTERNAL_URL
```

出力例

```
HTTP/2 200
content-length: 18
content-type: text/plain; charset=utf-8
date: Wed, 26 Jul 2023 12:56:22 GMT
server: istio-envoy
x-envoy-upstream-service-time: 2856

Hello Serverless!
```

4. 検証用に作成されたリソースを削除します。

```
$ oc delete deployment/curl -n team-alpha-1 && \
oc delete ksvc/test-webapp -n team-alpha-1 && \
oc delete ksvc/test-webapp -n team-bravo-1
```

OpenShift Container Platform の関連情報

- Helm ユーティリティー
- Helm ユーティリティーのオプションリファレンス

第3章 SERVERLESS と COST MANAGEMENT SERVICE の統合

Cost Management は OpenShift Container Platform のサービスで、クラウドおよびコンテナーのコストをより正確に把握し、追跡することができます。これは、オープンソースの Koku プロジェクトに基づいています。

3.1. 前提条件

- クラスター管理者パーミッションがある。
- コスト管理を設定し、OpenShift Container Platform source を追加している。

3.2. コスト管理クエリーにラベルを使用する

コスト管理では **タグ** とも呼ばれるラベルは、ノード、namespace、または Pod に適用できます。各ラベルはキーと値のペアです。複数のラベルを組み合わせてレポートを生成できます。Red Hat ハイブリッドコンソール を使用して、コストに関するレポートにアクセスできます。

ラベルは、ノードから namespace に、namespace から Pod に継承されます。ただし、ラベルがリソースにすでに存在する場合、ラベルはオーバーライドされません。たとえば、Knative サービスにはデフォルトの **app=<revision_name>** ラベルがあります。

例 Knative サービスのデフォルトラベル

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: showcase
spec:
...
      labels:
        app: <revision_name>
...
```

app=my-domain のように namespace のラベルを定義した場合は、**app=my-domain** タグを使用してアプリケーションに問い合わせたときに、**app=<revision_name>** タグの Knative サービスから生じるコストは Cost Management Service では考慮されません。このタグを持つ Knative サービスのコストは、**app=<revision_name>** タグの下で照会する必要があります。

3.3. 関連情報

- ソースへのタグ付けの設定
- Cost Explorer を使用したコストの可視化

第4章 SERVERLESS と OPENSHIFT PIPELINES の統合

Serverless と OpenShift Pipelines を統合すると、Serverless サービスの CI/CD パイプライン管理が可能になります。この統合を使用すると、Serverless サービスのデプロイメントを自動化できます。

4.1. 前提条件

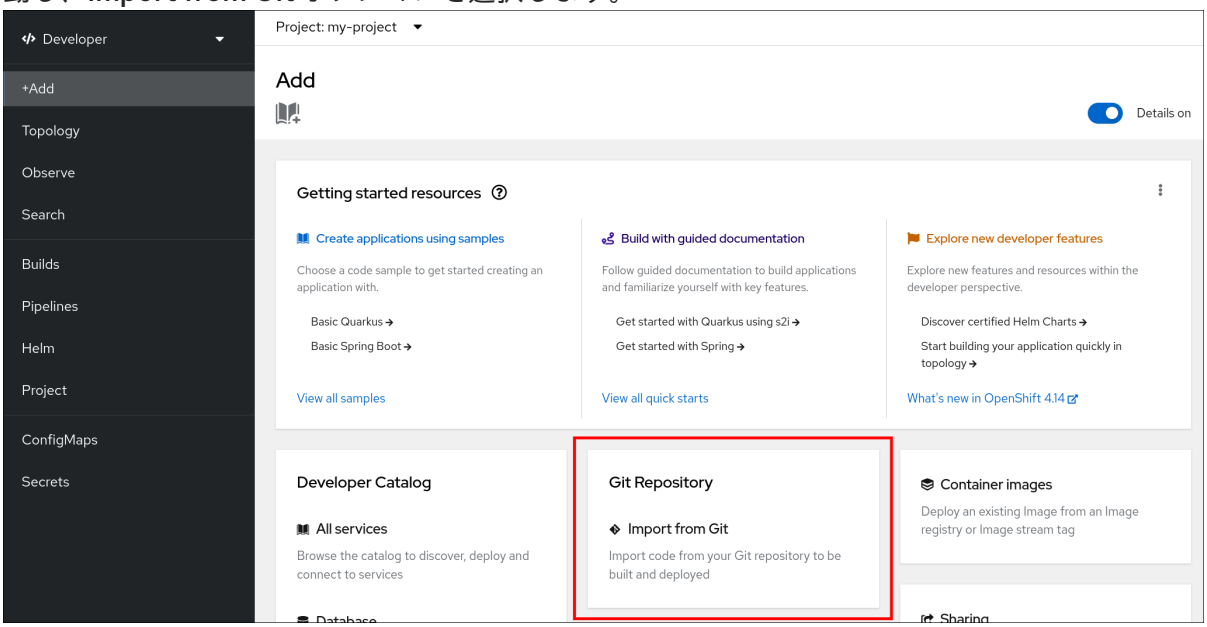
- **cluster-admin** 権限でクラスターにアクセスできる。
- OpenShift Serverless Operator および Knative Serving がクラスターにインストールされている。
- クラスターに OpenShift Pipelines Operator がインストールされている。

4.2. OPENSHIFT PIPELINES によってデプロイされるサービスの作成

OpenShift Container Platform Web コンソールを使用して、OpenShift Pipelines がデプロイするサービスを作成できます。

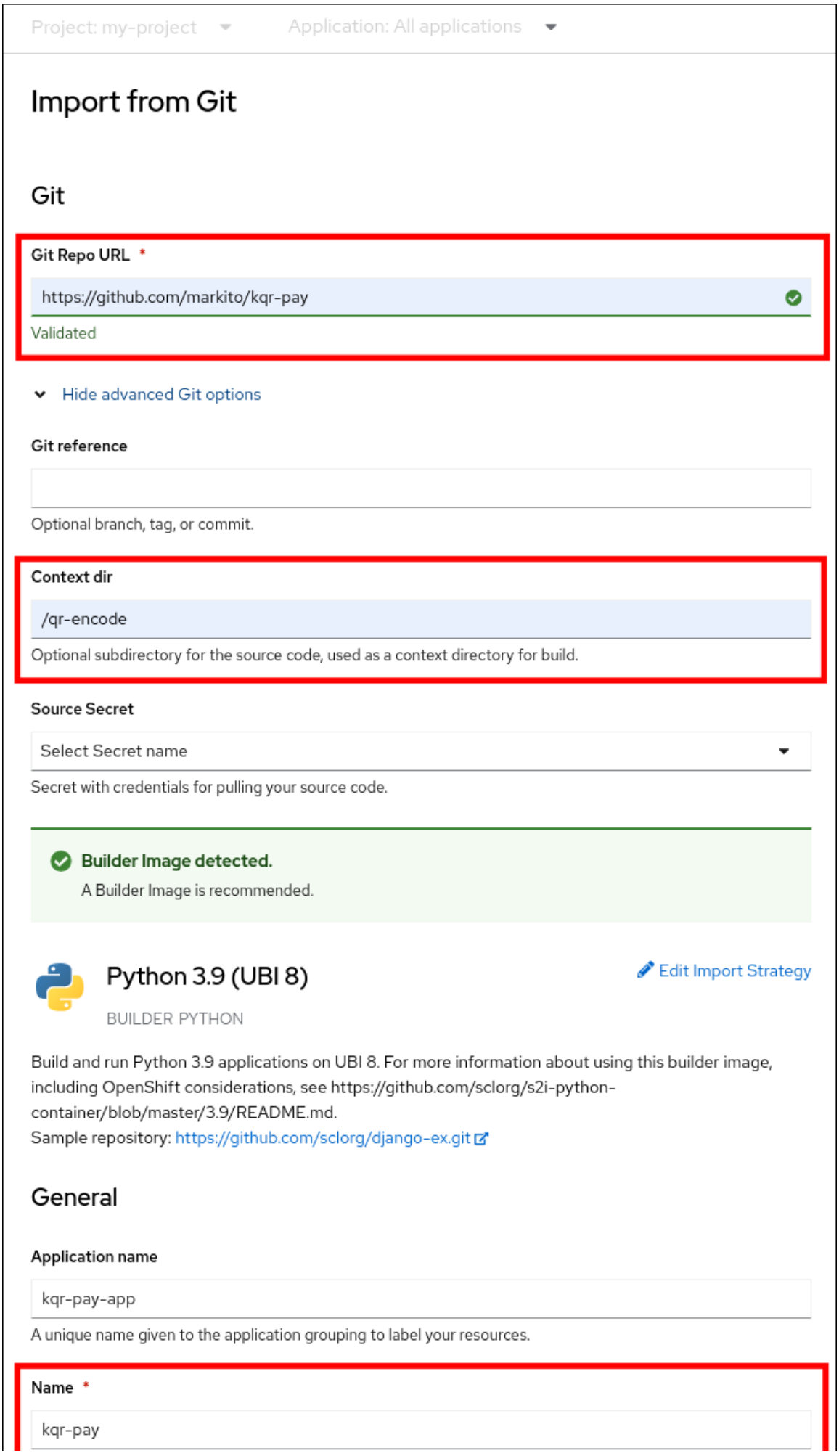
手順

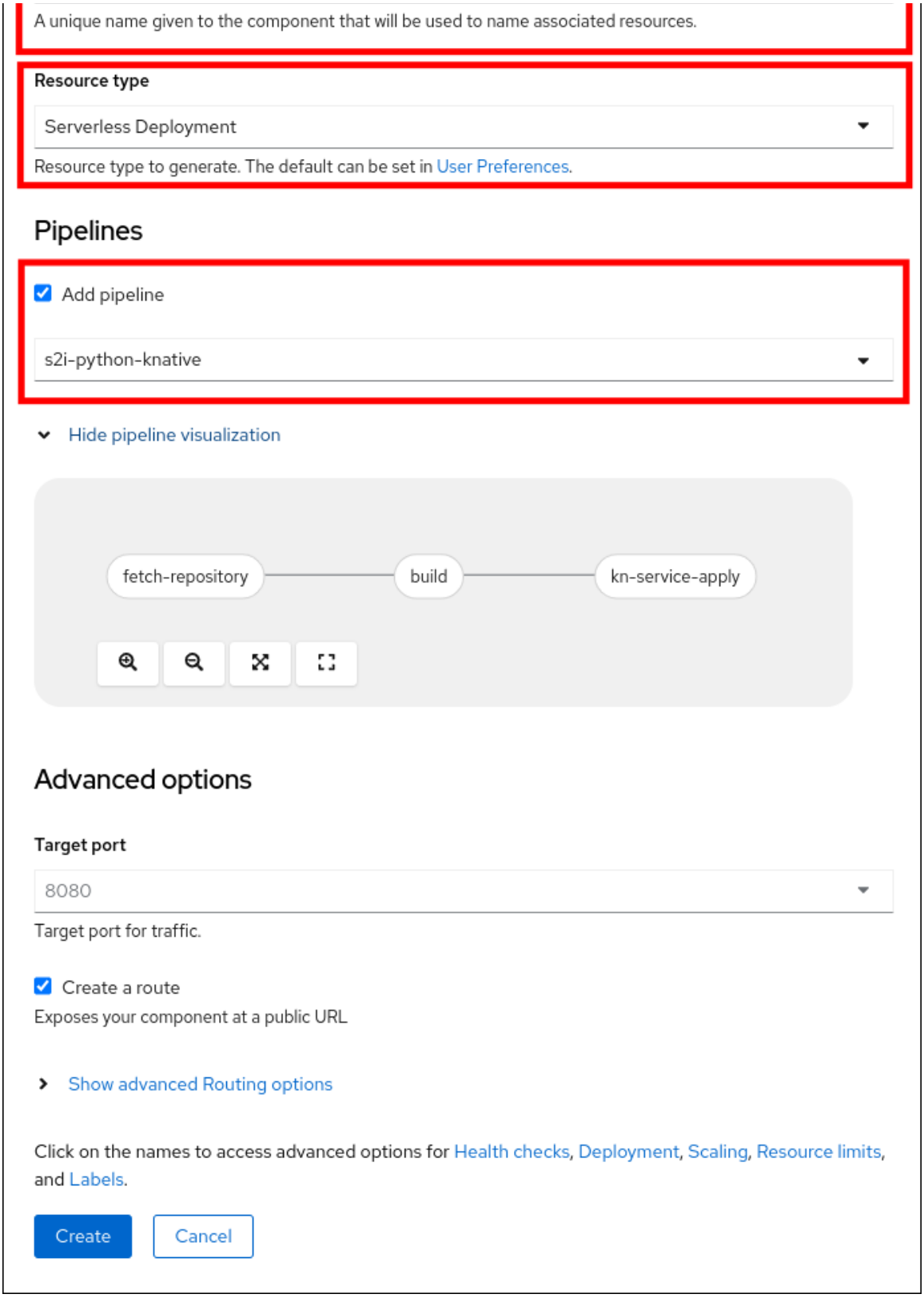
1. OpenShift Container Platform Web コンソールの **Developer** パースペクティブで、**+Add** に移動し、**Import from Git** オプションを選択します。



2. **Import from Git** ダイアログで、次の手順を実行してプロジェクトのメタデータを指定します。
   - Git リポジトリー URL を指定します。
   - 必要に応じて、コンテキストディレクトリーを指定します。これは、アプリケーションのソースコードのルートが含まれるリポジトリー内のサブディレクトリーです。
   - オプション: アプリケーション名を指定します。デフォルトでは、リポジトリー名が使用されます。
   - **Serverless デプロイメント** リソースタイプを選択します。
   - **Add pipeline** チェックボックスをオンにします。パイプラインはソースコードに基づいて自動的に選択され、その視覚化がスキーム上に表示されます。

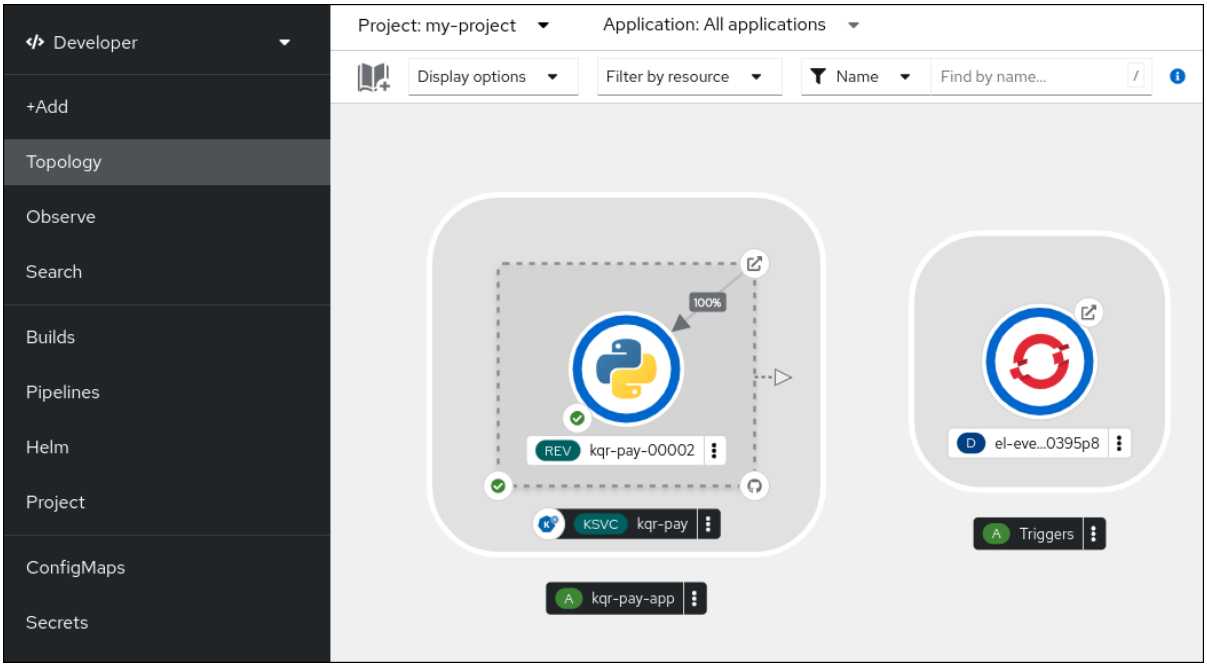
- その他の関連設定を指定します。



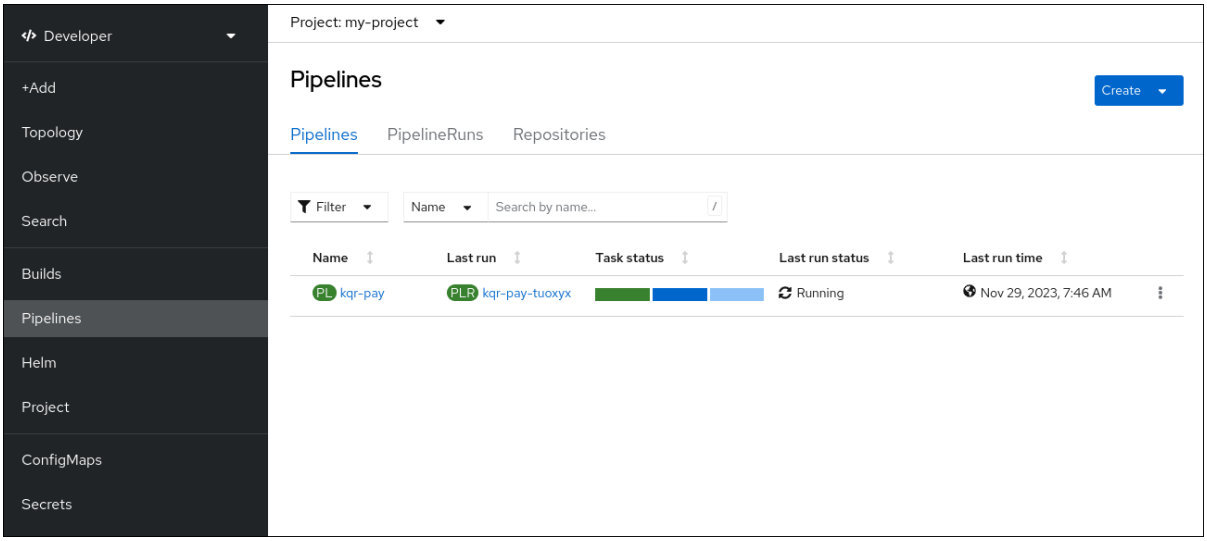


3. **Create** をクリックしてサービスを作成します。

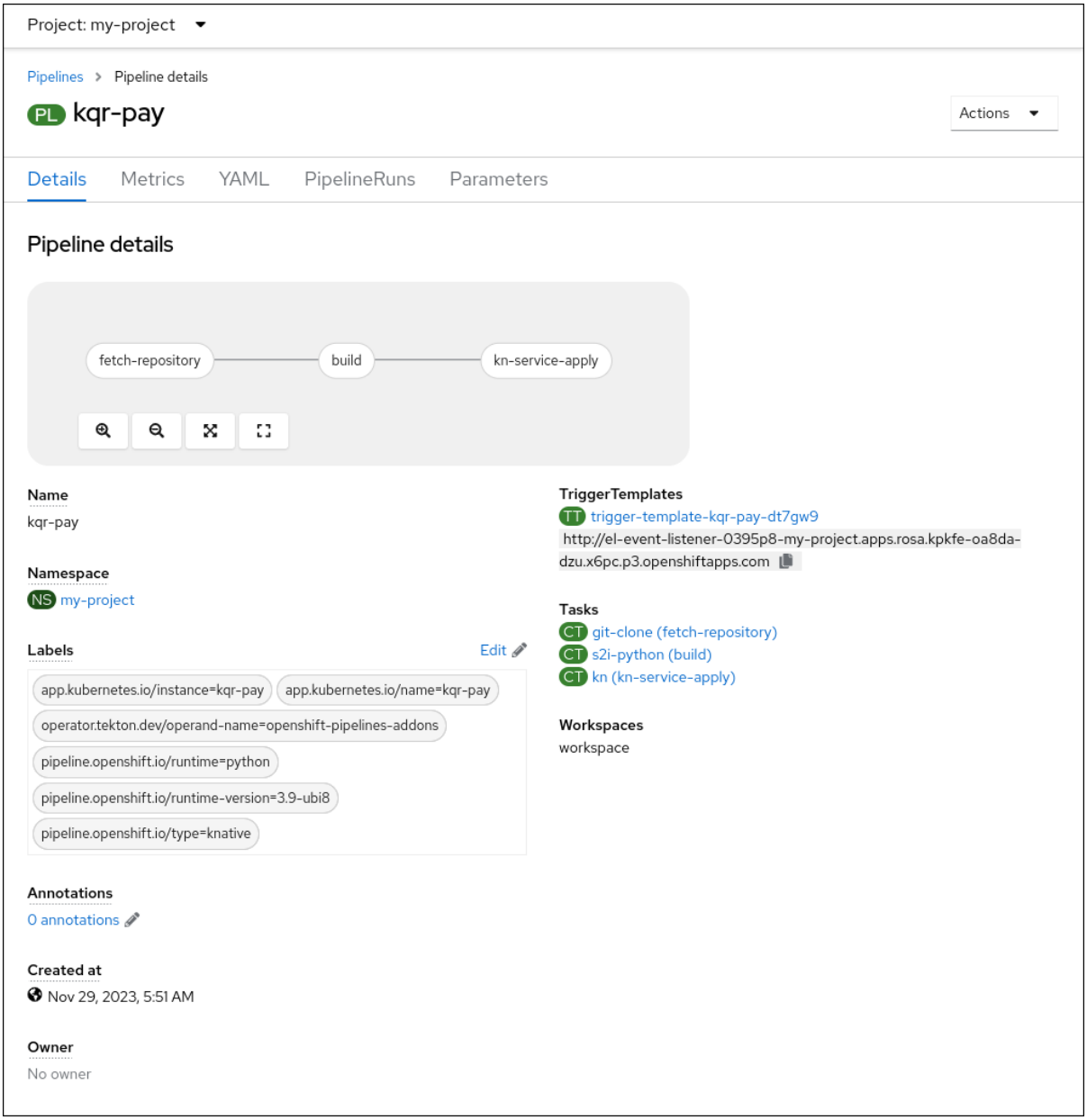
4. サービスの作成が開始すると、**Topology** 画面に移動します。この画面では、サービスと関連するトリガーが視覚化され、それらを操作できます。



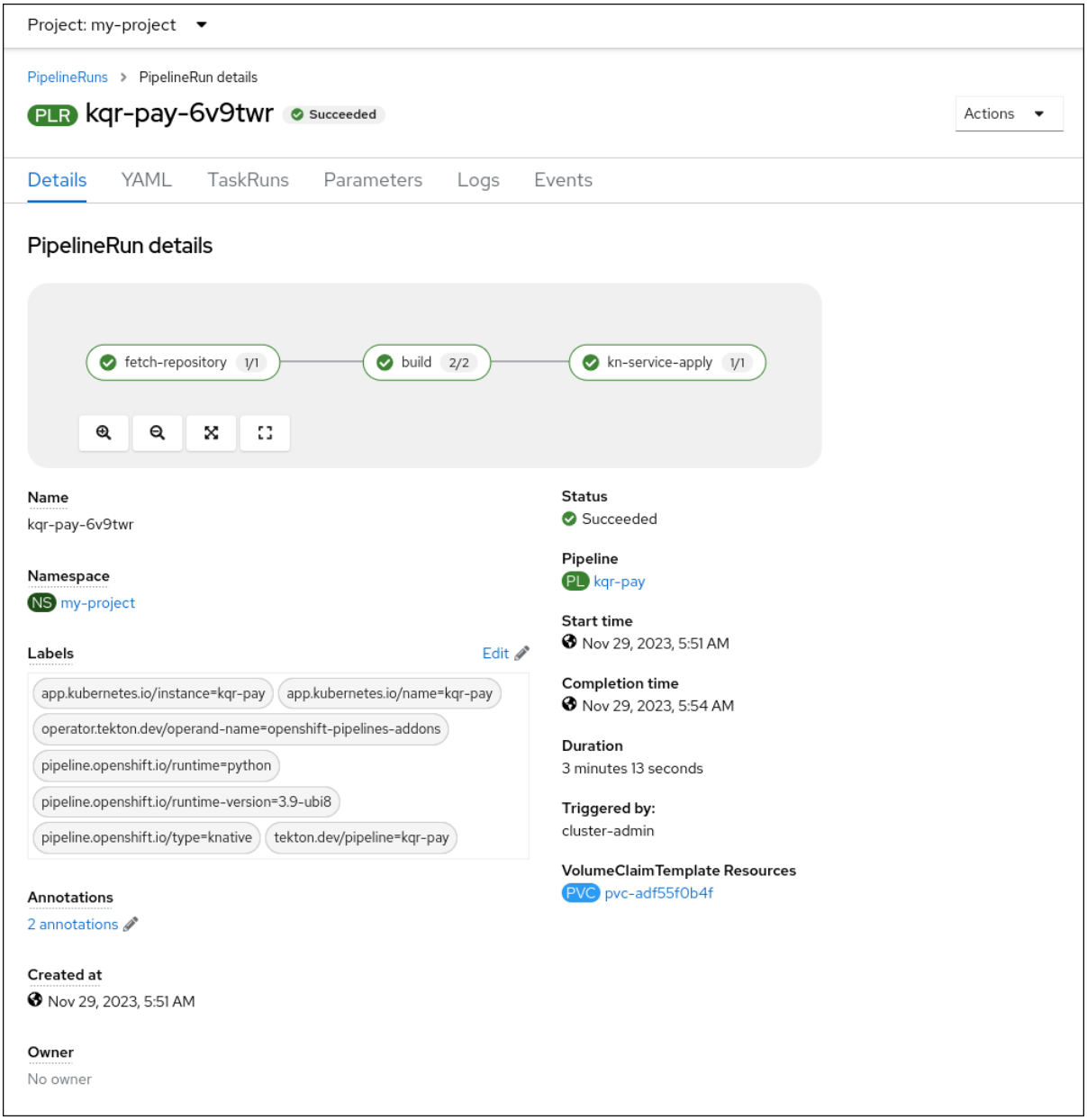
5. オプション: **Pipelines** ページに移動して、パイプラインが作成され、サービスが構築およびデプロイされていることを確認します。



6. パイプラインの詳細を表示するには、**Pipelines** ページでパイプラインをクリックします。



7. 現在のパイプライン実行の詳細を表示するには、**Pipelines** ページで実行の名前をクリックします。



4.3. 関連情報

- Red Hat OpenShift Pipelines のドキュメント

第5章 SERVERLESS アプリケーションでの NVIDIA GPU リソースの使用

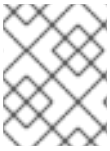
NVIDIA は、OpenShift Container Platform での GPU リソースの使用をサポートしています。OpenShift Container Platform での GPU リソースの設定に関する詳細は、OpenShift の GPU Operator を参照してください。

5.1. サービスの GPU 要件の指定

OpenShift Container Platform クラスターの GPU リソースが有効化された後に、Knative (**kn**) CLI を使用して Knative サービスの GPU 要件を指定できます。

前提条件

- OpenShift Serverless Operator、Knative Serving、および Knative Eventing がクラスターにインストールされている。
- Knative (**kn**) CLI がインストールされている。
- GPU リソースが OpenShift Container Platform クラスターで有効にされている。
- OpenShift Container Platform でアプリケーションおよび他のワークロードを作成するために、プロジェクトを作成しているか、適切なロールおよびパーミッションを持つプロジェクトにアクセスできる。



注記

NVIDIA GPU リソースの使用は、OpenShift Container Platform または OpenShift Dedicated の IBM zSystem および IBM Power ではサポートされていません。

手順

1. Knative サービスを作成し、**--limit nvidia.com/gpu=1** フラグを使用して、GPU リソース要件の制限を **1** に設定します。

```
$ kn service create hello --image <service-image> --limit nvidia.com/gpu=1
```

   GPU リソース要件の制限が **1** の場合、サービスには専用の GPU リソースが 1 つ必要です。サービスは、GPU リソースを共有しません。GPU リソースを必要とするその他のサービスは、GPU リソースが使用されなくなるまで待機する必要があります。

   1 GPU の制限は、1 GPU リソースの使用を超えるアプリケーションが制限されることも意味します。サービスが 2 つ以上の GPU リソースを要求する場合、これは GPU リソース要件を満たしているノードにデプロイされます。

2. オプション: 既存のサービスの場合は、**--limit nvidia.com/gpu=3** フラグを使用して、GPU リソース要件の制限を **3** に変更できます。

```
$ kn service update hello --limit nvidia.com/gpu=3
```

5.2. OPENSHIFT CONTAINER PLATFORM の関連情報

- 拡張リソースのリソースクォータの設定