

# Creating, Deploying, and Scaling out a StreamSets Microservice on Kubernetes

## Table of Contents

<b>Introduction</b>	<b>2</b>
<b>Deploy and configure the ingress controller</b>	<b>2</b>
<b>Obtain a TLS cert and key for the ingress controller's hostname</b>	<b>2</b>
<b>Create a Namespace for the StreamSets components</b>	<b>3</b>
<b>Store the tls cert and key in a tls secret</b>	<b>3</b>
<b>Create a StreamSets Kubernetes Environment</b>	<b>3</b>
<b>Launch a StreamSets Kubernetes Deployment</b>	<b>3</b>
<b>Import the Microservice pipeline</b>	<b>4</b>
<b>Create a Keystore for the Pipeline</b>	<b>4</b>
<b>Store the keystore and password in secrets</b>	<b>5</b>
<b>VolumeMount the keystore and password secrets into the StreamSets engine</b>	<b>5</b>
<b>Create a Service</b>	<b>6</b>
<b>Create an Ingress</b>	<b>8</b>
<b>Configure the REST Service's App ID</b>	<b>9</b>
<b>Configure the REST Service's TLS settings</b>	<b>10</b>
<b>Deploy a StreamSets IsPrime Microservice Client</b>	<b>11</b>
<b>Run the StreamSets IsPrime Microservice Client</b>	<b>13</b>
<b>Scale out the StreamSets IsPrime Microservice</b>	<b>14</b>

# Introduction

This document provides an example of creating, deploying and scaling out a [StreamSets Microservice Pipeline](#) on Kubernetes with TLS. This example uses [ingress-nginx](#) as an ingress controller, but you could easily use your preferred ingress mechanism instead.

The project [here](#) contains the example's artifacts.

You can also deploy and run the lsPrime microservice and client on a cloud-native, tarball or Docker-based engine, without worrying about the k8s aspects of this example (though also without the easy horizontal scaling of the microservice described in the last section of this doc).

## Deploy and configure the ingress controller

Deploy the ingress controller using the command [here](#).

Get the external IP of the ingress controller:

```
mark@Marks-MacBook-Pro-2 ~ % kubectl get svc -A
```

NAMESPACE	NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
default	kubernetes	ClusterIP	10.0.0.1	<none>
ingress-nginx	ingress-nginx-controller	LoadBalancer	10.0.124.74	4.154.229.120
ingress-nginx	ingress-nginx-controller-admission	ClusterIP	10.0.110.178	<none>
kube-system	ama-metrics-ksm	ClusterIP	10.0.135.18	<none>
kube-system	ama-metrics-operator-targets	ClusterIP	10.0.166.30	<none>
kube-system	azure-wi-webhook-webhook-service	ClusterIP	10.0.33.236	<none>
kube-system	kube-dns	ClusterIP	10.0.0.10	<none>
kube-system	metrics-server	ClusterIP	10.0.31.112	<none>
kube-system	network-observability	ClusterIP	10.0.182.34	<none>

Map a DNS name to the ingress controller's external IP. I'll use the DNS name `aks.onefoursix.com`

## Obtain a TLS cert and key for the ingress controller's hostname

Obtain a TLS cert and key for the ingress controller's hostname. I'll use a wildcard cert and key for `*.onefoursix.com` generated using Lets Encrypt in the files `tls.crt` and `tls.key`.

## Create a Namespace for the StreamSets components

I'll create a namespace named ns1:

```
$ kubectl create ns ns1
```

## Store the tls cert and key in a tls secret

```
$ kubectl create secret tls streamsets-tls \  
  --key ~/certs/tls.key --cert ~/certs/tls.crt
```

## Create a StreamSets Kubernetes Environment

Create a StreamSets Kubernetes Environment and deploy a StreamSets Kubernetes Agent  
See the docs [here](#).

## Launch a StreamSets Kubernetes Deployment

Launch a StreamSets Kubernetes deployment in the namespace `ns1`. See the docs [here](#).

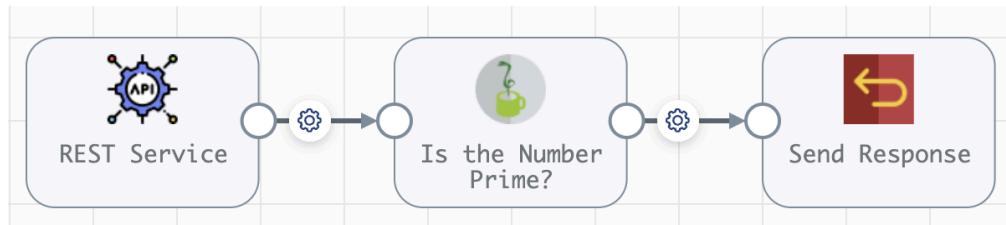
Include the [Jython Evaluator](#) stage library in the deployment configuration as the logic of the microservice is implemented in Jython.

This example requires StreamSets engine version 6.1.0 or higher.

# Import the Microservice pipeline

Import the `IsPrime` Microservice pipeline from the archive [here](#).

The pipeline looks like this:



The `IsPrime` Microservice takes a POST request payload like `{"number": 11}` like this:

```
$ curl -v https://aks.onefoursix.com/is-prime \
-H "Content-Type:application/json" \
-H "X-SDC-APPLICATION-ID:app1" \
-d '{"number": 11}'
```

The microservice returns `true` or `false` if the number is prime or not. For example:

```
{"statusCode":200,"data":[{"number":11,"is_prime":"true"}],"error":[]}
```

## Create a Keystore for the Pipeline

*Important note: You may encounter downstream errors trying to use keystores generated on macOS. If you are working on macOS I recommend you generate the keystore on Linux and then copy the keystore back to your Mac for the subsequent steps.*

Create a Java keystore in pkcs12 format for the pipeline's [REST Service](#) connector. I'll use the same wildcard TLS cert and key as described in the steps above using a command like this:

```
$ openssl pkcs12 -export -in tls.crt -inkey tls.key -name is-prime
-out is-prime-keystore.p12
```

This gives me the keystore file `is-prime-keystore.p12`.

You will be prompted to enter and confirm a password for the keystore; store that value in a text file on your local machine named `is-prime-keystore-password.txt`

## Store the keystore and password in secrets

Store the keystore and password in Kubernetes secrets:

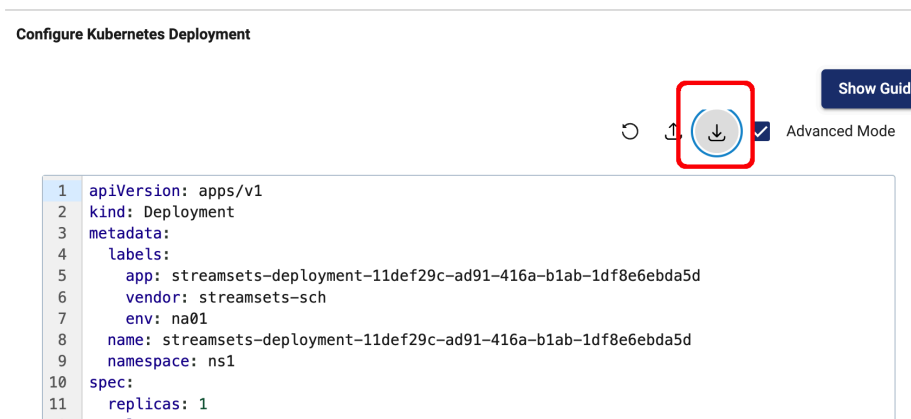
```
$ kubectl -n ns1 create secret generic is-prime-keystore
--from-file=is-prime-keystore.p12
```

```
$ kubectl -n ns1 create secret generic is-prime-keystore-password
--from-file=is-prime-keystore-password.txt
```

## VolumeMount the keystore and password secrets into the StreamSets engine

In the StreamSets Kubernetes deployment's [advanced mode](#) add Volumes for the two secrets and VolumeMount the secrets into the Deployment. See the yaml file [here](#) for an example.

*Tip: Don't try to edit the yaml within the StreamSets UI; instead, click the download button (see the screenshot below) to save the generated yaml to your local machine, edit the yaml using your favorite text editor, and then import the file back into the StreamSets UI.*



Once you have imported the edited yaml back into the StreamSets UI and saved your changes, the StreamSets engine pod(s) will get redeployed, and once they are back online, you can confirm the Volume and VolumeMounts are working by seeing if the two files are present in engine's `/resources` directory, using a command like this:

```
$ kubectl exec -it <engine-pod-id> -- bash -c 'ls -l /resources'
```

You should see output like this:

```
mark@Marks-MacBook-Pro-2 ~ % k exec -it streamsets-deployment-11def29c-ad91-416a-b1a
b-1df8e6ebda5d2n5xx -- bash -c 'ls -l /resources'
total 12
-rw-r--r-- 1 root root 12 Mar  5 20:25 is-prime-keystore-password.txt
-rw-r--r-- 1 root root 4420 Mar  5 20:25 is-prime-keystore.p12
```

## Create a Service

Create a Kubernetes ClusterIP Service to expose the microservice pipeline. I'll use port 8000 for the both the pipeline's and the services port

In order to get the value for the service's selector, describe one of the deployment's engine pod and grab its label using a command like this:

```
$ kubectl describe po <engine-pod-id> | grep Labels
```

You should see output like this:

```
mark@Marks-MacBook-Pro-2 ~ % kubectl describe po streamsets-deployment-11def29c-ad91-4
16a-b1ab-1df8e6ebda5d9mfhn | grep Labels
Labels:      app=streamsets-deployment-11def29c-ad91-416a-b1ab-1df8e6ebda5d
```

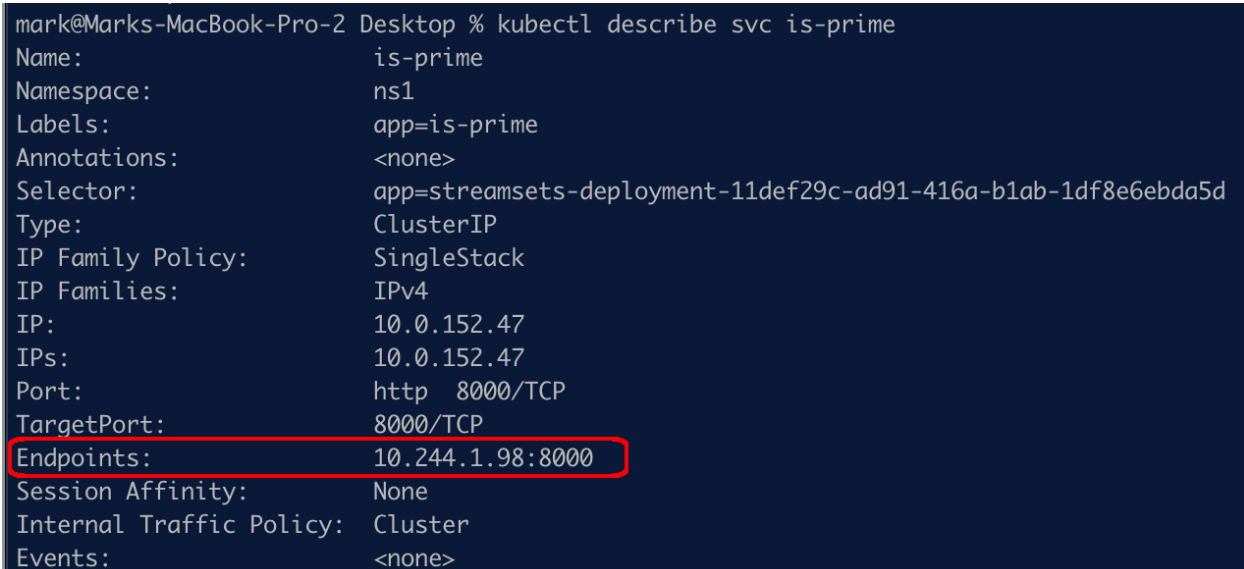
I created a file named `is-prime-service.yaml` on my local machine with this content, including the value of the app label in the selector:

```
apiVersion: v1
kind: Service
metadata:
  name: is-prime
  namespace: ns1
  labels:
    app: is-prime
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 8000
    targetPort: 8000
    protocol: TCP
  selector:
    app: streamsets-deployment-11def29c-ad91-416a-b1ab-1df8e6ebda5d
```

I'll apply that yaml to create the service:

```
$ kubectl -n ns1 apply -f is-prime-service.yaml
```

Describe the service and make sure it has a valid backend:



```
mark@Marks-MacBook-Pro-2 Desktop % kubectl describe svc is-prime
Name: is-prime
Namespace: ns1
Labels: app=is-prime
Annotations: <none>
Selector: app=streamsets-deployment-11def29c-ad91-416a-b1ab-1df8e6ebda5d
Type: ClusterIP
IP Family Policy: SingleStack
IP Families: IPv4
IP: 10.0.152.47
IPs: 10.0.152.47
Port: http 8000/TCP
TargetPort: 8000/TCP
Endpoints: 10.244.1.98:8000
Session Affinity: None
Internal Traffic Policy: Cluster
Events: <none>
```

→ That confirms the service's selector is correct.

# Create an Ingress

We'll need an ingress to expose the ClusterIP service to external callers, and the syntax for the ingress depends on your choice of ingress controller or gateway. On OpenShift you could create a Route rather than an ingress. This example is for [ingress-nginx](#).

I created a file named `is-prime-ingress.yaml` on my local machine with this content:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: is-prime
  namespace: ns1
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/rewrite-target: /$2
    nginx.ingress.kubernetes.io/backend-protocol: "https"
spec:
  ingressClassName: nginx
  tls:
  - hosts:
    - aks.onefoursix.com
    secretName: streamsets-tls
  rules:
  - host: aks.onefoursix.com
    http:
      paths:
      - path: /is-prime(/|$)(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: is-prime
            port:
              number: 8000
```



Describe the ingress and make sure it has a valid endpoint:

```
mark@Marks-MacBook-Pro-2 Desktop % kubectl describe ingress is-prime
Name: is-prime
Labels: <none>
Namespace: ns1
Address: 4.154.229.120
Ingress Class: nginx
Default backend: <default>
TLS:
  streamsets-tls terminates aks.onefoursix.com
Rules:
  Host      Path      Backends
  ----      -
  aks.onefoursix.com  /is-prime(/|$)(.*)  is-prime:8000 (10.244.1.98:8000)
Annotations:
  nginx.ingress.kubernetes.io/auth-tls-verify-client: off
  nginx.ingress.kubernetes.io/backend-protocol: https
  nginx.ingress.kubernetes.io/proxy-set-headers: X-Client-Cert $ssl_client_cert
  nginx.ingress.kubernetes.io/rewrite-target: /$2
  nginx.ingress.kubernetes.io/ssl-redirect: true
Events:
  Type      Reason      Age          From                      Message
  ----      -
  Normal    Sync        58s (x2 over 82s)  nginx-ingress-controller  Scheduled for sync
```

## Configure the REST Service's App ID

The screenshot shows the configuration interface for a REST Service, specifically the **HTTP** tab. The **HTTP** tab is highlighted with a red box. Below the tabs, there is a checkbox for **Use API Gateway** which is unchecked. The **HTTP Listening Port \*** is set to **8000**. The **List of Application IDs** section is highlighted with a red box, showing a list with one entry, **app1**, which is also highlighted with a red box. To the right of the **app1** entry are two circular icons: a blue one with a diagonal line and a red one with a minus sign.

# Configure the REST Service's TLS settings

The screenshot shows the 'TLS' tab of a configuration interface. It includes checkboxes for 'Use TLS' (checked), 'Require Client Authentication' (unchecked), and 'Use Remote Keystore' (unchecked). Below these are input fields for 'Keystore File' (is-prime-keystore.p12), 'Keystore Type' (PKCS-12 (p12 file)), and 'Keystore Password' (a runtime resource path).

General HTTP Data Format **TLS** HTTP Response

☒ Use TLS ⓘ <>

☐ Require Client Authentication ⓘ <>

☐ Use Remote Keystore ⓘ <>

Keystore File ⓘ <>

is-prime-keystore.p12

Keystore Type \* ⓘ <>

PKCS-12 (p12 file)

Keystore Password ⓘ ⓘ

`${runtime:loadResource('is-prime-keystore-password.txt', 'false')}`

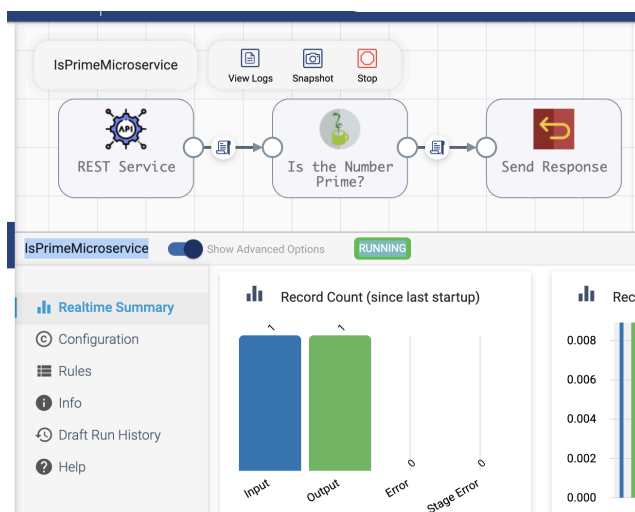
Start the

## Pipeline and Call the Service

Start the pipeline and call the service like this:

```
curl -v https://<ingress controller host name>/is-prime \
-H "Content-Type:application/json" \
-H "X-SDC-APPLICATION-ID:app1" \
-d '{"number": 11}'
```

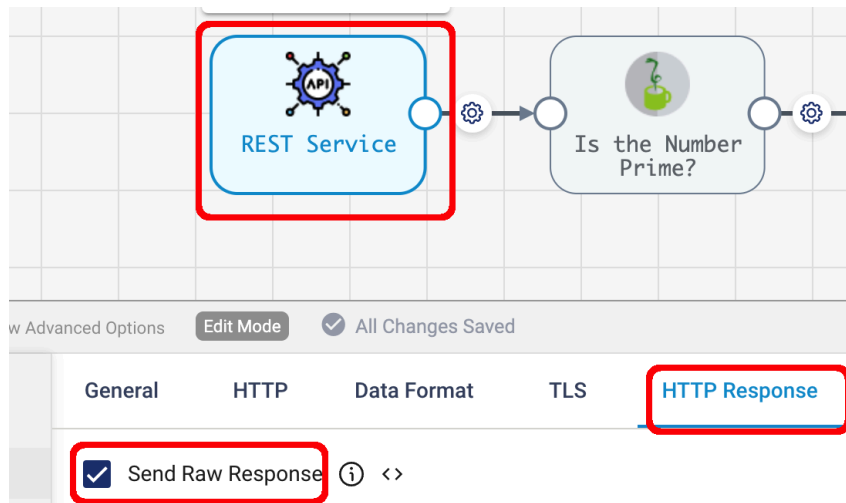
You should see the request is handled by the service:



And you should get a response like this:

```
{"statusCode":200,"data":[{"number":11,"is_prime":"true"}],"error":[]}
```

Note that you can also set this option in the Rest Service Origin's config:



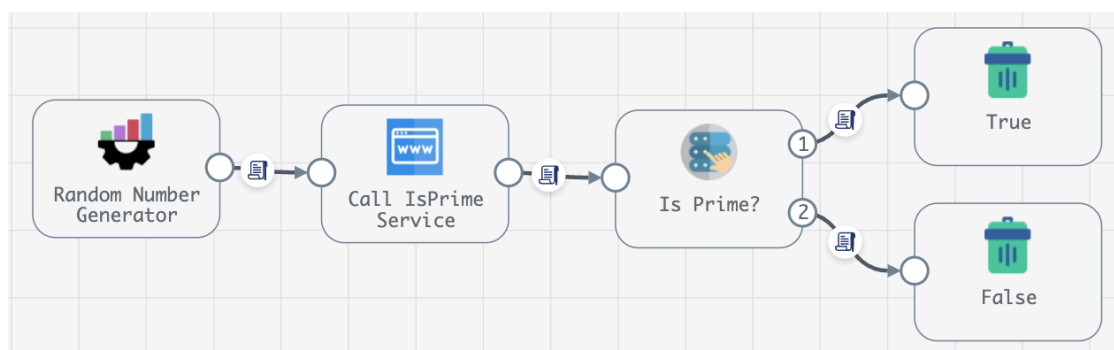
Which will return a response like this:

```
{"number":11,"is_prime":"true"}
```

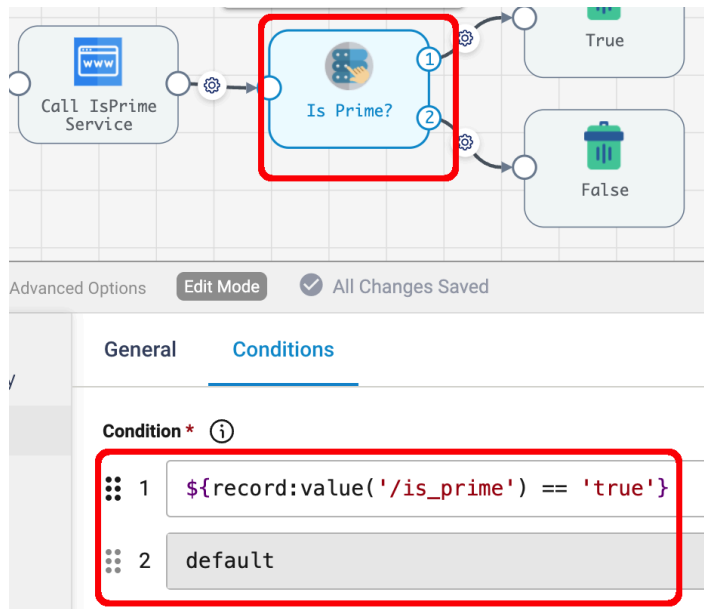
The IsPrime microservice examples below use the Send Raw Response setting to simplify the response parsing by the client

## Deploy a StreamSets IsPrime Microservice Client

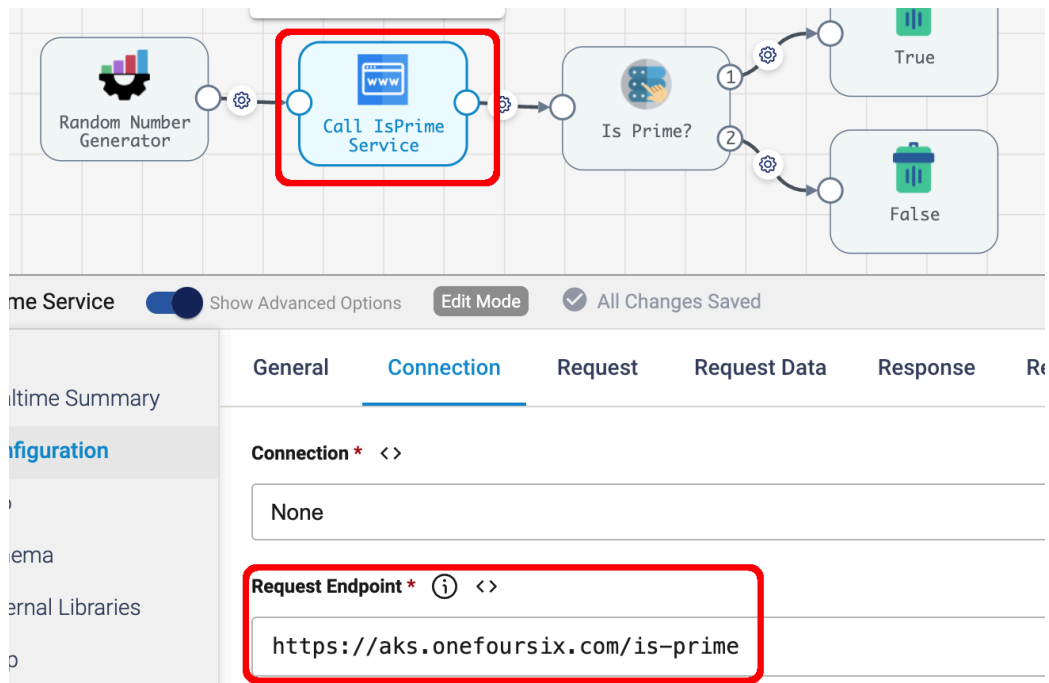
Import the IsPrime Microservice Client pipeline from [here](#). That pipeline looks like this:



The pipeline generates an endless stream of random integers, passes each one in a request to the IsPrime service and routes prime number responses to the top leg of the Is Prime? Stream Selector and non-prime number responses to the lower leg. The condition in the Is Prime? Stream Selector looks like this:

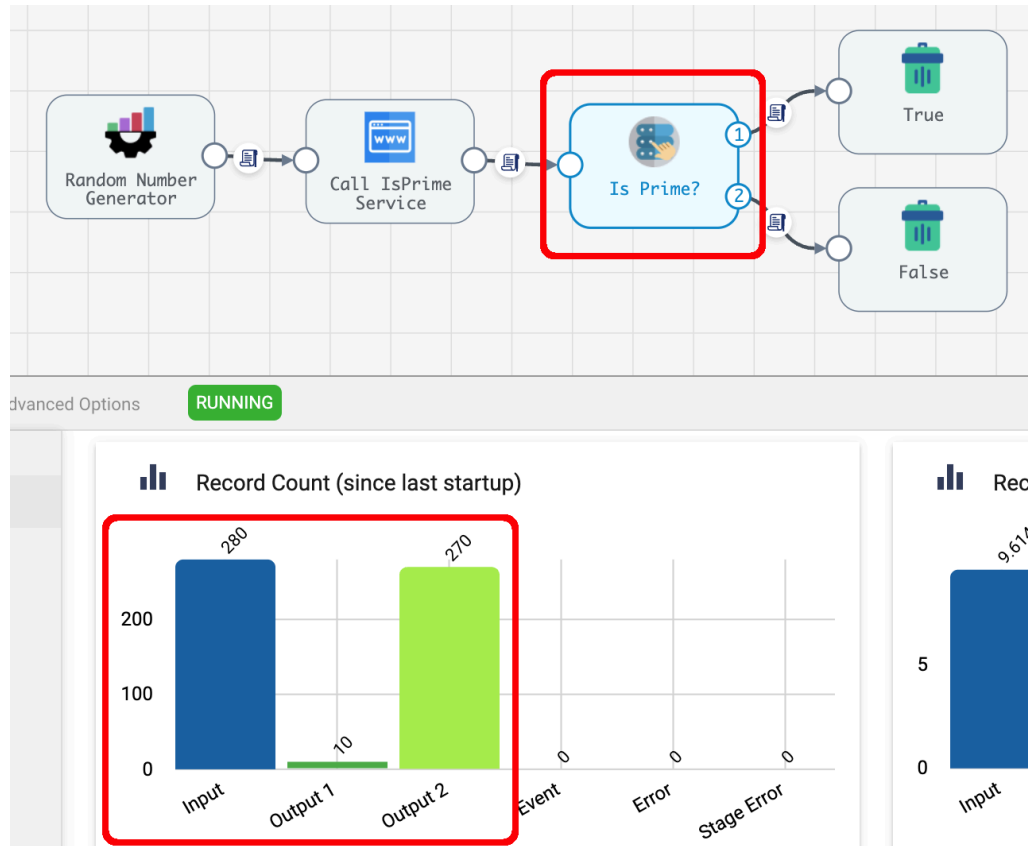


Set the URL of your IsPrime microservice here:



# Run the StreamSets IsPrime Microservice Client

Run the StreamSets IsPrime Microservice Client and click on the `Is Prime?` Stream Selector to observe the relative number of prime vs non-prime random integers, in my case 10 out of 280:



# Scale out the StreamSets IsPrime Microservice

To scale out the IsPrime Microservice, start by editing the deployment and bumping up the number of instances; I'll set that value to 3:

## Edit Deployment

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    labels:
5      app: streamsets-deployment-11def29c-ad91-416
6      vendor: streamsets-sch
7      env: na01
8    name: streamsets-deployment-11def29c-ad91-416a
9    namespace: ns1
10 spec:
11   replicas: 3
12   selector:
13     matchLabels:
14       app: streamsets-deployment-11def29c-ad91-4
```

If you are not using the Advanced Kubernetes Configuration, just set the UI control like this:

### Configure Kubernetes Deployment

Kubernetes Labels: [+ Add](#) [Bulk Edit Mode](#)

Enable Autoscaling: ☐

Desired Instances




Save your changes.

You should see three engines are now online:

Data Collector      Transformer

---

**Data Collector Engines** ( 3 with current filters ) ? ↺ ≡ ⋮

<input type="checkbox"/>	Engine	Deployment	R	CPU Load	Memory Us	Last Reported	
<input type="checkbox"/>	 streamsets-deployment-11def...	aks (Kuberne...	1	0.43 %	31.73 %	a minute ago	⋮
<input type="checkbox"/>	 streamsets-deployment-11def...	aks (Kuberne...	0	0.84 %	45.57 %	a minute ago	⋮
<input type="checkbox"/>	 streamsets-deployment-11def...	aks (Kuberne...	0	0.68 %	45.46 %	a minute ago	⋮

And you can see the three engine pods:

```
mark@Marks-MacBook-Pro-2 ~ % k get po
NAME                                READY   STATUS    RESTARTS   AGE
streamsets-agent-dep-f700bf93-223e-4bf0-984e-b90de05dd8af-nmhjd  1/1     Running   0          4d22h
streamsets-deployment-11def29c-ad91-416a-b1ab-1df8e6ebda5d2n5xx  1/1     Running   0          5h8m
streamsets-deployment-11def29c-ad91-416a-b1ab-1df8e6ebda5dcs4rj  1/1     Running   0          3m10s
streamsets-deployment-11def29c-ad91-416a-b1ab-1df8e6ebda5dsjtdc  1/1     Running   0          3m10s
```

Next, create a [Job Instance](#) for the IsPrime Microservice pipeline. Set the Job's engine label to match the scaled out deployment and set the Job's number of instances to -1 (which will force the Job to run one pipeline instance per engine that has the matching label):

### Edit Job

Configure job details to determine how engines run the pipeline. [Learn more](#)

Deployment:

Engine Labels:  ✕ Add New...

Enable Failover: ☒

[Hide Advanced Options](#) ^

Number of Instances:

Start the Job and it should be running on three engines:

[Job Instances](#) > Job for IsPrimeMicroservice

Job Instance Details

Monitor Job

Stop

Synchronize

Edit

Export

Status

ACTIVE

Start Time

Mar 5, 2025, 5:39:59 PM


Finish Time

Dec 31, 1969, 4:00:00 PM

Run Count

1


Data Collectors



streamsets-deployment-11def29c-ad91-4...

RUNNING


a few seconds ago



streamsets-deployment-11def29c-ad91-4...

RUNNING

a few seconds ago



streamsets-deployment-11def29c-ad91-4...

RUNNING

a few seconds ago

Describe the service to see the three endpoints:

```
mark@Marks-MacBook-Pro-2 ~ % k describe svc is-prime
Name: is-prime
Namespace: ns1
Labels: app=is-prime
Annotations: <none>
Selector: app=streamsets-deployment-11def29c-ad91-416a-b1ab-1df8e6ebda5d
Type: ClusterIP
IP Family Policy: SingleStack
IP Families: IPv4
IP: 10.0.152.47
IPs: 10.0.152.47
Port: http 8000/TCP
TargetPort: 8000/TCP
Endpoints: 10.244.1.249:8000,10.244.0.36:8000,10.244.0.89:8000
Session Affinity: None
Internal Traffic Policy: Cluster
Events: <none>
```

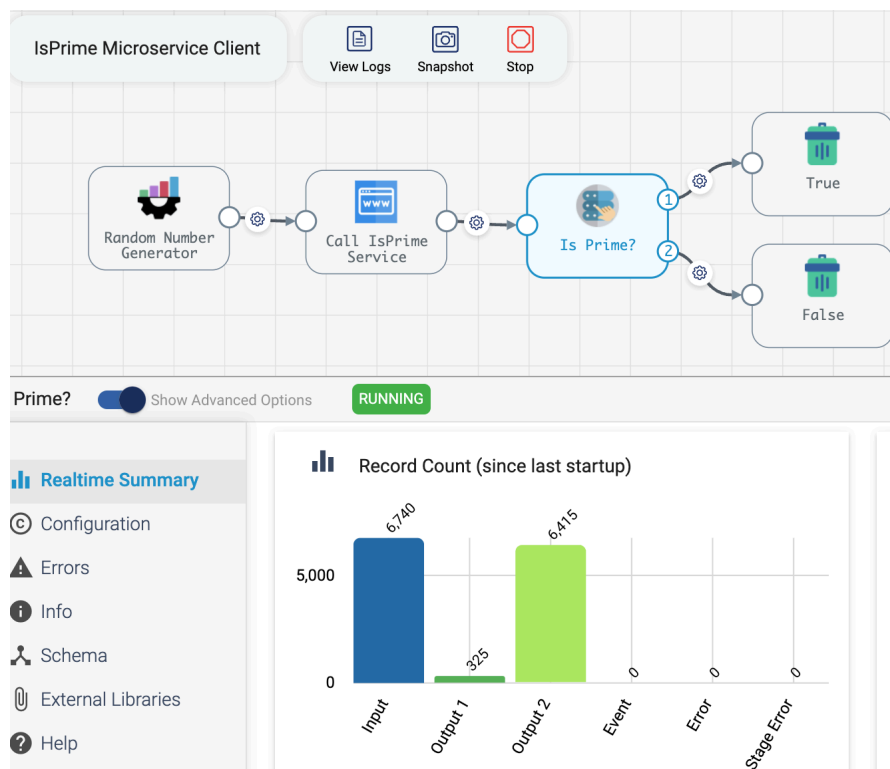
The Kubernetes service will automatically round-robin requests across the three instances (yay Kubernetes!).



To put more load on our scaled-out microservice, we can scale up the IsPrime Microservice Client by bumping its REST Service Origin to run on three concurrent threads (the docs on multi-threaded pipelines are [here](#)):

The screenshot shows the configuration panel for the 'Random Number Generator' component. The 'Number of Threads' is set to 3, which is highlighted with a red box. The 'Batch Size' is set to 10. The panel includes tabs for 'General' and 'Data Generator', and a sidebar with options like 'Realtime Summary', 'Configuration', 'Info', 'Schema', 'External Libraries', and 'Help'.

Start the IsPrime Microservice Client pipeline and note it is working just as before and its display is for all threads running concurrently:



Monitor the IsPrime Microservice Job and once again note it is running three instances:

The screenshot shows the configuration for the 'IsPrimeMicroservice' job. The workflow consists of three stages: 'REST Service', 'Is the Number Prime?', and 'Send Response'. A yellow box contains a curl command: 

```
$ % curl -v https://aks.onefoursix.com/prime \ -H "Content-Type:application/json" \ -H "X-SDC-APPLICATION-ID:" -d '{"number": 11}'
```

Below the workflow, the 'Data Collectors' table is displayed, showing three instances in a 'RUNNING' state.

Hostname	Status	Last Reported Time
streamsets-deployment-11def29c-ad91-416a-b1ab-1df8e6ebd...	RUNNING	a few seconds ago
streamsets-deployment-11def29c-ad91-416a-b1ab-1df8e6ebd...	RUNNING	a minute ago
streamsets-deployment-11def29c-ad91-416a-b1ab-1df8e6ebd...	RUNNING	a few seconds ago

In the Realtime summary, pick any of the three instances to see the metrics for that instance:

The screenshot shows the 'Realtime Summary' for the selected instance 'streamsets-deployment-11def29c-ad91-416a-b1ab-1df8e6ebda5dsjtdc:18630'. The left sidebar highlights 'Realtime Summary'. The main area displays four charts: 'Record Count (since last startup)', 'Record Throughput (records / sec)', and 'Runtime Statistics'.

**Record Count (since last startup):**

Category	Count
Input	3,878
Output	3,878
Error	0
Stage Error	0

**Record Throughput (records / sec):**

Time Period	Throughput
1m	~8.5
5m	~6.5
15m	~3.5
30m	~2.0
mean	~5.0

**Runtime Statistics:**

Metric	Value
Last Batch Input Count:	1
Last Batch Output Count:	1
Last Batch Error Count:	0
Records Count:	0
Last Batch Messages Count:	0

You should see the number of requests per instance will be roughly the same.