Cloud Native Runtimes for VMware Tanzu 1.0

Cloud Native Runtimes for VMware Tanzu 1.0

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

Cloud Native Runtimes for Tanzu Overview 7
- Scaling 7
- Development 8
- Deployment 8
- Event-Driven Apps 8

Cloud Native Runtimes for Tanzu Release Notes 9
- v1.0.3 9
  - Resolved Issues 9
  - Components 9
- v1.0.2 9
  - Resolved Issues 9
- v1.0.1 10
  - Resolved Issues 10
- v1.0.0 10
  - Breaking Changes 10
  - New Features 10
  - Known Issues 11
  - Components 11
- v0.2.0 12
  - Features 12
  - Known Issues 12
  - Components 12

Installing Cloud Native Runtimes for Tanzu 13
- Prerequisites 13
- Create a Kubernetes Cluster 15
- Download Cloud Native Runtimes 15
- Use Image Relocation with Cloud Native Runtimes 16
  - Prerequisites 16
  - Relocate Image to Private Registry 16
- Install Cloud Native Runtimes 17
  - Install on Tanzu Kubernetes Grid 17
  - Install on TKGI 18
  - Install on Tanzu Mission Control 18

Install on vSphere 18
Install on Kubernetes Cloud Platforms 18
Install on a Local Kubernetes Cluster Provider 19
Set Up External DNS 19

Installing Cloud Native Runtimes for Tanzu with an Existing Contour Installation 21
About Using Contour with Cloud Native Runtimes 21
Prerequisites 21
Identify Your Contour Version 22
Install Cloud Native Runtimes on a Cluster with Your Existing Contour Instances 22

Integrating RabbitMQ with Cloud Native Runtimes for Tanzu 24
About the RabbitMQ Operators 24
Install RabbitMQ Cluster Kubernetes Operator v1.8.2 24
Install RabbitMQ Messaging Topology Kubernetes Operator v1.2.1 25
Next Steps 26

Verifying Your Installation 27
Verify with a Private Registry 27
Verify Installation of Knative Serving, Knative Eventing, and TriggerMesh SAWS 27

Preparing to Create a Service 27

Verifying Knative Serving 28
About Verifying Knative Serving 28
Prerequisites 28
Test Knative Serving 28
Delete the Example Knative Service 30

Verify Knative Eventing 30
About Verifying Knative Eventing 30
Prerequisites 30
Prepare the RabbitMQ Environment 31
Verify Knative Eventing 32
Delete the Eventing Resources 33

Verifying TriggerMesh SAWS 34
Prerequisites 34
Verify TriggerMesh SAWS 34

Developing Locally on Kind 37
Prerequisites 37
Configure Your Local Kind Cluster 37
Install Cloud Native Runtimes Locally 38
Test Your CoreDNS 38
Set the Host Machine DNS 38
Set on MacOS 38
Set on Linux 39
Set on Windows 39
Reset the Host Machine DNS 39
Reset on MacOS 39
Reset on Linux 39
Reset on Windows 40

Enabling Automatic TLS Certificate Provisioning for Cloud Native Runtimes for Tanzu 41
Prerequisites 41
Enable Auto TLS Using an HTTP01 Challenge 41
Enable Auto TLS Using a DNS01 Challenge 43

Configuring Observability for Cloud Native Runtimes for Tanzu 46
Logging 46
Configure Logging with Fluent Bit 46
Forward Logs to vRealize 47
Metrics 47
Tracing 48
Configuring Tracing 48
Forwarding Trace Data to a Data Visualization Tool 48
Sending Trace Data to an Observability Platform 49
Use Wavefront Dashboards 49
Import Wavefront Dashboards 50
Import Dashboards with the Wavefront API 50
Import with the Ruby Wavefront CLI 50

Configuring Cloud Native Runtimes for Tanzu with Avi Vantage 52
Integrate Avi Vantage with Cloud Native Runtimes 52
About Routing with Avi Vantage and Cloud Native Runtimes 53

Configuring Cloud Native Runtimes for Tanzu with Tanzu Service Mesh 55
Run Cloud Native Runtimes on a Cluster Attached to Tanzu Service Mesh 55
Next Steps 56

Troubleshooting Cloud Native Runtimes for Tanzu 57
Cannot connect to app on AWS 57
Symptom 57
Solution 57
minikube Pods Fail to Start 57
Symptom 57
Solution 57
Knative Services never become ready when using AutoTLS 57
Symptom 58
Solution 58
Installation fails with kapp-controller v0.16 58
Symptom 58
Solution 58
Installation fails to reconcile app/cloud-native-runtimes 59
Symptom 59
Explanation 59
Solution 59
Example 1: The Cloud Provider does not support the creation of Service type LoadBalancer 59
Example 2: The webhook deployment failed 60
Cloud Native Runtimes Installation Fails with Existing Contour Installation 61
Symptom 61
Solution 61
Upgrading Cloud Native Runtimes for Tanzu 62
Upgrade from Beta to GA 62
Uninstalling Cloud Native Runtimes for Tanzu 63

Cloud Native Runtimes for Tanzu Overview

Cloud Native Runtimes for Tanzu is a serverless application runtime for Kubernetes that is based on Knative and runs on a single Kubernetes cluster. For information about Knative, see the Knative documentation. Cloud Native Runtimes capabilities are included in VMware Tanzu Advanced Edition and VMware Tanzu Application Platform. For information about Tanzu Advanced Edition, see Overview of Tanzu Advanced Edition. For information about Tanzu Application Platform, see About Tanzu Application Platform.

Cloud Native Runtimes is compatible with clusters from the following Kubernetes platform providers:

- VMware Tanzu Kubernetes Grid. See VMware Tanzu Kubernetes Grid documentation.
- VMware Tanzu Kubernetes Grid Integrated Edition (TKGI). See VMware Tanzu Kubernetes Grid Integrated Edition documentation.
- vSphere 7.0 with Tanzu. See VMware vSphere 7.0 Release Notes.
- Amazon Elastic Kubernetes Service (EKS). See Amazon Elastic Kubernetes Service in the AWS documentation.
- Azure Kubernetes Service (AKS). See Azure Kubernetes Service in the Azure documentation.
- Google Kubernetes Engine (GKE). See Google Kubernetes Engine in the Google Cloud documentation.

Cloud Native Runtimes supports:

- Scale-to-zero
- Scale-from-zero
- Event-triggered workloads

You can integrate Cloud Native Runtimes with the following products:

- Tanzu Observability by Wavefront. See Configuring Observability for Cloud Native Runtimes for Tanzu.
- Tanzu Build Service. See Tanzu Build Service documentation.
- Avi Vantage. See Configuring Cloud Native Runtimes for Tanzu with Avi Vantage.
- RabbitMQ. See Integrating RabbitMQ with Cloud Native Runtimes for Tanzu.
- Tanzu Service Mesh. See Configuring Cloud Native Runtimes for Tanzu with Tanzu Service Mesh.

Scaling

With Cloud Native Runtimes, you can scale the number of pods up and down based on the incoming request rate. Pods can be configured to scale to zero when the containers are not processing requests. Pods can be configured to automatically start when a new request arrives. Cloud Native Runtimes supports bounded concurrency, letting you limit how many requests a pod can process at the same time. Adaptive scaling lets you compute according to the amount of traffic in order to conserve server resources.

Development

With Cloud Native Runtimes, you can use a developer abstraction on top of Kubernetes, letting you develop without managing servers and updating workloads. Knative includes abstractions like the Service developer abstraction that manages several Kubernetes serving and deploying concepts. Administrators can create a unified developer experience across cloud platforms with a shared Knative foundation.

Deployment

Cloud Native Runtimes lets developers use blue/green or canary deployment methods, and built in traffic splitting. Traffic splitting lets you control how changes to your app are rolled out to a set percentage of users. These features provide the ability to control how features are rolled out incrementally to users.

Event-Driven Apps

Cloud Native Runtimes supports event-driven apps by using Knative. You can use event triggers to scale your app-based on demand. For example, you can configure an event source to send an event notification to a broker when a file is uploaded to a storage bucket, store that event, forward that event to matching triggers, and then run image recognition on that video file. You can use many different kinds of events and event sources. Using the Knative framework makes many external integrations and microservices available.

Cloud Native Runtimes for Tanzu Release Notes

This topic contains release notes for Cloud Native Runtimes for Tanzu v1.0.

v1.0.3

Release Date: October 25, 2021

Resolved Issues

This release has the following fixes:

- **Kubernetes 1.22** compatibility issues

Components

Cloud Native Runtimes v1.0.3 updates the following component versions:

Release	Details
Version	v1.0.3
Release date	October 25, 2021
Component	**Version**
Knative Eventing	0.23.5
Knative Discovery	0.23.0
Knative Eventing RabbitMQ Integration	0.23.2
Knative cert-manager Integration	0.23.0
Knative Serving Contour Integration	0.23.1
VMware Tanzu Sources for Knative	0.23.1
TriggerMesh Sources from Amazon Web Services (SAWS)	1.6.0
vSphere Event Sources	0.23.0

v1.0.2

Release Date: August 30, 2021

Resolved Issues

This release has the following fixes:

- **ConfigMap changes now persist in the following namespaces:**
  - knative-sources
  - triggermesh
  - contour-external
  - contour-internal
- **Use Cloud Native Runtimes together with a private registry and an existing Contour:** Cloud Native Runtimes is v1.0.1 and earlier was not compatible with using private registry and an existing Contour.
- **rabbitmq-controller-manager-token secrets:** `rabbitmq-controller-manager-token` secrets are now created only once.

v1.0.1

Release Date: July 30, 2021

Resolved Issues

This release has the following fixes:

- **Fixes Installation fails with image relocation on a private registry.**

v1.0.0

Release Date: July 21, 2021

Breaking Changes

Breaking changes in this release:

- **Cloud Native Runtimes v1.0.0 installation requires kapp-controller v0.17.0 or later:** Before installing v1.0.0, install kapp-controller v0.17.0 or later on your Kubernetes cluster.
- **If you previously installed Cloud Native Runtimes v0.2.0 or later, uninstall that earlier version before you install v1.0.0:** See Upgrading Cloud Native Runtimes.
- **Cloud Native Runtimes v1.0.0 depends on Kubernetes v1.18 or later.**

New Features

New features in this release:

- **(Beta) Eventing integration for RabbitMQ:** Knative Eventing Brokers can use RabbitMQ for better performance and reliability. The eventing integration for RabbitMQ is in beta. VMware does not recommend using the eventing integration for RabbitMQ in a production environment.
- **TriggerMesh Sources for Amazon Web Services:** Knative Events can be sourced from AWS S3, SQS, and other parts of the AWS platform.

- **Avi Vantage for Cloud Native Runtimes:** You can configure Cloud Native Runtimes to integrate with Avi Vantage.
- **Tanzu Service Mesh:** You can use this workaround use Tanzu Service Mesh with Cloud Native Runtimes. See Configuring Cloud Native Runtimes with Tanzu Service Mesh.

Known Issues

This release has the following issues:

- **Cloud Native Runtimes installation fails on Calico CNI in vSphere 7.0 with Tanzu environments:** If your installation fails, use the Antrea CNI. For information about the Antrea CNI, see About Upgrading from the Flannel CNI to the Antrea CNI in the Tanzu Kubernetes Grid Integrated Edition documentation.
- **Some patches of Tanzu Kubernetes Grid install an incompatible version of kapp-controller:** On some Kubernetes versions and cloud providers, Tanzu Kubernetes Grid v1.3.1 installs kapp-controller v0.16.0, which is incompatible with Cloud Native Runtimes. For more information, see Installation fails with kapp-controller v0.16 in *Troubleshooting*.
- **Installation fails with image relocation on a private registry:** Your installation fails with image relocation and you see a `Failed to pull image` error message when you use a private registry.
- The recommended versions of RabbitMQ Cluster Operator, cert-manager, and RabbitMQ Messaging Topology Operator have been updated to ensure compatibility with Kubernetes v1.22.0+. The recommended versions are now:
    - RabbitMQ Cluster Kubernetes Operator v1.8.2
    - cert-manager v1.5.3
    - RabbitMQ Messaging Topology Kubernetes Operator v1.2.1

Components

Cloud Native Runtimes v1.0.0 uses the following component versions:

Release	Details
Version	v1.0
Release date	July 21, 2021
Component	**Version**
Knative Eventing	0.23.1
Knative Discovery	0.23.1
Knative Eventing RabbitMQ Integration	0.23.0
Knative cert-manager Integration	0.23.0
Knative Serving Contour Integration	0.23.0
VMware Tanzu Sources for Knative	0.23.0
TriggerMesh Sources from Amazon Web Services (SAWS)	1.6.0

vSphere Event Sources	0.23.0

v0.2.0

Release Date: March 31, 2021

Features

Features in this release:

- **Knative Serving:** You can deploy serverless applications with autoscaling and URL access.
- **(Beta) Knative Eventing:** You can connect Kubernetes workloads to event sources including external integrations and other workloads. Knative Eventing features are in beta. VMware does not recommend using the Knative eventing functionality in a production environment.

Known Issues

This release has the following issue:

- **Cloud Native Runtimes installation fails on Calico CNI in vSphere 7.0 with Tanzu environments:** If your installation fails, use the Antrea CNI. For information about the Antrea CNI, see About Upgrading from the Flannel CNI to the Antrea CNI in the Tanzu Kubernetes Grid Integrated Edition documentation.

Components

Cloud Native Runtimes v0.2.0 uses the following component versions:

Release	Details
Version	v0.2.0
Release date	March 31, 2021
Component	**Version**
Knative Serving	0.20.0
Knative Serving Contour Integration	0.20.0
Knative RabbitMQ Eventing	0.20.0
Knative Eventing	0.20.1
VMware Tanzu Sources for Knative	0.20.0

Installing Cloud Native Runtimes for Tanzu

This topic describes how to install Cloud Native Runtimes for Tanzu. This includes installing the serving and eventing services. You must install a Kubernetes cluster on a cloud platform provider, install command line tools, configure your cluster, and download Cloud Native Runtimes before installing. You install Cloud Native Runtimes on a Kubernetes cluster.

Prerequisites

The following prerequisites are required to install Cloud Native Runtimes:

- Kubernetes v1.18 or later
    - For information about creating a compatible Kubernetes cluster, see Create a Kubernetes Cluster. Cloud Native Runtimes is compatible with a Kubernetes cluster on the following Kubernetes providers:
        - Tanzu Kubernetes Grid v1.3.1 and later
        - Tanzu Kubernetes Grid Integrated Edition (TKGI)
        - Tanzu Mission Control
        - vSphere 7.0 with Tanzu
        - Google Kubernetes Engine (GKE)

          Note: GKE Autopilot is not supported.
        - Azure Kubernetes Service
        - Amazon Elastic Kubernetes Service
        - Docker Desktop
        - kind
        - minikube

      Note: For a cluster with one node, set CPUs to at least 6, memory to at least 6.0 GB, and disk storage to at least 30 GB. For a cluster with multiple nodes, set CPUs to at least 2, memory to at least 4.0 GB, and disk storage to at least 20 GB for each node.
    - Your Cloud Provider must support the creation of Services of type `LoadBalancer`. The exception is local installation, which does not require support for Service type `LoadBalancer`.

      For information about Service type `LoadBalancer`, see the Kubernetes documentation and your cloud provider documentation. For more information about Tanzu Kubernetes Grid support for Service type `LoadBalancer`, see Install VMware

NSX Advanced Load Balancer on a vSphere Distributed Switch.

- Kapp-controller v0.17.0 or later. To download kapp-controller, see Install in the Carvel documentation.

  Note: Kapp-controller is pre-installed on Tanzu Kubernetes Grid v1.3.1 and later.

- Command line tools. The following command line tools are required:
  - kubectl (v1.18 or later)
  - kapp (v0.34.0 or later)
  - ytt (v0.30.0 or later)
  - kbld (v0.28.0 or later)
  - kn
- (Highly recommended for production environments) A domain name for your installation. You use this domain name to set up the external DNS as described in Set Up External DNS below.
- (Optional) Use the Octant Plugin for Knative to view, manage, create, and delete Knative resources within Octant. For information about installing Octant, see Octant Plugin for Knative in GitHub.
- If you are installing Cloud Native Runtimes on a cluster that is attached to Tanzu Service Mesh, see Configuring Cloud Native Runtimes with Tanzu Service Mesh.
- Pod Security Policy role bindings. If you have pod security policies (PSP) enabled on your Kubernetes cluster, create one of the following role bindings on the Kubernetes cluster where you install kapp-controller and Cloud Native Runtimes:
  - vSphere 7.0 with Tanzu:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: kapp-controller-psp-role-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: psp:vmware-system-restricted
subjects:
- kind: ServiceAccount
  name: kapp-controller-sa
  namespace: kapp-controller
```

  - Tanzu Kubernetes Grid Integrated Edition (TKGI)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: kapp-controller-psp-role-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: psp:restricted
subjects:
```

```
- kind: ServiceAccount
  name: kapp-controller-sa
  namespace: kapp-controller
```

Create a Kubernetes Cluster

To use Cloud Native Runtimes, you must have a Kubernetes cluster. See the following resources to create and configure your Kubernetes cluster, depending on your platform provider. Cloud Native Runtimes is compatible with a Kubernetes cluster on any of the following Kubernetes cloud platform providers:

- **Tanzu Kubernetes Grid.** For information about using clusters with Tanzu Kubernetes Grid, see Connect to and Examine Tanzu Kubernetes Clusters.
- **Tanzu Kubernetes Grid Integrated Edition.** For information about using clusters with TKGI, see Managing Kubernetes Clusters and Workloads in the TKGI documentation.
- **Tanzu Mission Control.** For information about using clusters with Tanzu Mission Control, see Provision a Cluster in the Tanzu Mission Control documentation.
- **vSphere 7.0 with Tanzu.** For information about using clusters for vSphere 7.0 with Tanzu, see vSphere with Tanzu Configuration and Management in the VMware vSphere documentation.
- **Google Kubernetes Engine.** For information about using clusters with Google Kubernetes Engine (GKE), see the Google Kubernetes Engine documentation.

  Note: GKE Autopilot is not supported.
- **Azure Kubernetes Service.** For information about using clusters with Azure Kubernetes Service (AKS), see the Azure Kubernetes Service documentation.
- **Amazon Elastic Kubernetes Service.** For information about using clusters with Amazon Elastic Kubernetes Service (EKS), see Amazon EKS Clusters in the EKS documentation.
- **Docker Desktop.** For information about using clusters with Docker Desktop, see the Docker Desktop documentation.
- **kind.** For information about using clusters with kind, see the kind documentation.
- **minikube.** For information about using clusters with minikube, see the minikube documentation.

Download Cloud Native Runtimes

To install Cloud Native Runtimes, you must first download the installation package from VMware Tanzu Network.

To download Cloud Native Runtimes:

1. Log into VMware Tanzu Network.
2. Navigate to the Cloud Native Runtimes release page.
3. Download the `cloud-native-runtimes-1.0.x.tgz` archive.
4. Extract the contents of `cloud-native-runtimes-1.0.x.tgz`, for example:

```
tar -xvf cloud-native-runtimes-1.0.0.tgz
```

Use Image Relocation with Cloud Native Runtimes

Follow this image relocation procedure if either of the following are true:

- You do not have access to the VMware Harbor registry.
- Your security policies require that you access images from a designated private registry.

If you are installing Cloud Native Runtimes using image relocation with a registry that does not have a publicly-rooted certificate, you need to provision your cluster with a self-signed certificate. For information about provisioning a cluster with a self-signed certificate, see How to Set Up a Harbor Registry with Self-Signed Certificates for Tanzu Kubernetes Clusters.

Prerequisites

In addition to the prerequsites listed above, you need the following prerequisites:

- imgpkg v0.13.0 or later. To download imgpkg, see the imgpkg website.
- To use image relocation with a private registry, set the following environment variables:
  - `cnr_registry__server`. Where `cnr_registry__server` is the URI of the registry.
  - `cnr_registry__username`. Where `cnr_registry__username` is the username for the registry.
  - `cnr_registry__password`. Where `cnr_registry__password` is the password to access the registry.

  Note: The environment variables include two underscore symbols (_).

Relocate Image to Private Registry

To relocate the Cloud Native Runtimes image to a private registry:

1. Download `cloud-native-runtimes-1.0.x.lock` file from the Cloud Native Runtimes release page.
2. Log in to your registry through Docker or, for other authentication options, such as environment variables, see the imgpkg documentation.
3. Push the bundle to a registry. Run:

   ```
   imgpkg copy --lock cloud-native-runtimes-1.0.x.lock --to-repo LINK-TO-PRIVATE-R
   EPO --lock-output LOCK-OUTPUT
   ```

   Where:

   - `LINK-TO-PRIVATE-REPO` is the path to the private registry.
   - `LOCK-OUTPUT` is the name of your lock output file.

   Note: If you do not have the certificates for your private registry, then add `--registry-verify-certs=false` to the command and to the command in step 4.

For example:

```
 $ imgpkg copy -lock cloud-native-runtimes-1.0.0.lock -to-repo my.corp
.registry/cnr -lock-output ./relocated.lock -registry-verify-certs=fal
se
```

4. Pull your image. Run:

   ```
   imgpkg pull --lock LOCK-OUTPUT -o ./cloud-native-runtimes
   ```

   Where `LOCK-OUTPUT` is the name of your lock output file.

   For example:

   ```
   $ imgpkg pull -lock ./relocated.lock -o ./cloud-native-runtimes
   ```

5. Navigate to the `cloud-native-runtimes` directory. Run:

   ```
   cd cloud-native-runtimes
   ```

6. Mark the `install.sh` file as executable by updating the install script permission. Run:

   ```
   chmod +x ./bin/install.sh
   ```

7. Follow the steps in Preparing to Create a Service to create a secret for your private registry.

Install Cloud Native Runtimes

Use one of the following procedures, depending on your platform, to install Cloud Native Runtimes. To install, you target the cluster and run the installation script.

Note: If you see the following error message after you run the Cloud Native Runtimes installation script, see Installing Cloud Native Runtimes with an Existing Contour Installation:

`Could not proceed with installation. Refer to Cloud Native Runtimes documentation for details on how to utilize an existing Contour installation. Another app owns the custom resource definitions listed below.`

Install on Tanzu Kubernetes Grid

To install Cloud Native Runtimes on Tanzu Kubernetes Grid:

1. Target the cluster you want to use. See Connect to Your New Cluster in the Tanzu Kubernetes Grid documentation.
2. Verify that you are targeting the correct Kubernetes cluster. Run:

   ```
   kubectl cluster-info
   ```

3. Run the installation script from the `cloud-native-runtimes` directory:

   ```
   ./bin/install.sh
   ```

   Note: If the installation fails with a `kapp: Error:` message, see Installation fails with kapp-

controller v0.16 in *Troubleshooting*.

Install on TKGI

To install Cloud Native Runtimes on Tanzu Kubernetes Grid Integrated Edition:

1. Target the cluster you want to use. See Create a Kubernetes Cluster in the TKGI documentation.
2. Verify that you are targeting the correct Kubernetes cluster. Run:

   ```
   kubectl cluster-info
   ```

3. Run the installation script from the `cloud-native-runtimes` directory:

   ```
   ./bin/install.sh
   ```

Install on Tanzu Mission Control

To install Cloud Native Runtimes on Tanzu Mission Control:

1. Target the cluster you want to use. See Register Your Management Cluster in the Tanzu Mission Control documentation.
2. Verify that you are targeting the correct Kubernetes cluster. Run:

   ```
   kubectl cluster-info
   ```

3. Run the installation script from the `cloud-native-runtimes` directory:

   ```
   ./bin/install.sh
   ```

Install on vSphere

To install Cloud Native Runtimes on vSphere 7.0 with Tanzu:

1. Target the cluster you want to use. See Register Your Management Cluster with Tanzu Mission Control in the VMware Tanzu Kubernetes Grid documentation.
2. Verify that you are targeting the correct Kubernetes cluster. Run:

   ```
   kubectl cluster-info
   ```

3. Run the installation script from the `cloud-native-runtimes` directory:

   ```
   cnr_provider=tkgs ./bin/install.sh
   ```

Install on Kubernetes Cloud Platforms

To install Cloud Native Runtimes on Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), or Google Kubernetes Engine (GKE):

1. Target the cluster you want to use:
   - For Elastic Kubernetes Service (EKS), see the Amazon documentation.

- For Azure Kubernetes Service (AKS), see the Azure documentation.
- For Google Kubernetes Engine (GKE), see the Google documentation.

2. Verify that you are targeting the correct Kubernetes cluster. Run:

```
kubectl cluster-info
```

3. Run the installation script from the `cloud-native-runtimes` directory:

```
./bin/install.sh
```

Install on a Local Kubernetes Cluster Provider

To install Cloud Native Runtimes on Docker Desktop, kind, or minikube:

Note: To install on minikube, you need at least 4GB of available system RAM for all pods to start.

1. Target the cluster you want to use. See Docker Desktop for Mac user manual, kind User Guide, or minikube start.
2. Verify that you are targeting the correct Kubernetes cluster. Run:

```
kubectl cluster-info
```

3. Run the installation script from the `cloud-native-runtimes` directory:

```
cnr_provider=local ./bin/install.sh
```

Set Up External DNS

Knative uses `example.com` as the default domain. After Cloud Native Runtimes is installed on your cluster, you change the default domain to your custom domain.

Note: If you are setting up Cloud Native Runtimes for development or testing, you do not have to set up an external DNS. However, if you want to access your workloads (apps) over the internet, then you do need to set an external DNS.

To set up the custom domain and its external DNS record:

1. Set your custom domain by following the instructions Edit using kubectl or Apply from a file in the Knative documentation.

   When your workloads are created, Knative automatically creates URLs for each workload based on this custom domain.

2. Get the address of the cluster load balancer:

```
kubectl get service envoy -n contour-external --output 'jsonpath={.status.loadB
alancer.ingress}'
```

   If this command returns a URL instead of an IP address, then `ping` the URL to get the load balancer IP address.

3. Create a wildcard DNS `A` record that assigns the custom domain to the load balancer IP. Follow the instructions provided by your domain name registrar for creating records.

   The record created looks like:

   ```
   *.DOMAIN IN A TTL LOADBALANCER-IP
   ```

   Where:

   - `DOMAIN` is the custom domain.
   - `TTL` is the time-to-live.
   - `LOADBALANCER-IP` is the load balancer IP.

   For example:

   ```
   *.mydomain.com IN A 3600 198.51.100.6
   ```

Installing Cloud Native Runtimes for Tanzu with an Existing Contour Installation

This topic describes how to configure Cloud Native Runtimes for Tanzu with your existing Contour instance. Cloud Native Runtimes uses Contour to manage internal and external access to the services in a cluster.

If you see an error about `an existing Contour installation` when you run the `install.sh` script, then follow the procedures on this page to install Cloud Native Runtimes.

About Using Contour with Cloud Native Runtimes

Cloud Native Runtimes needs two instances of Contour. By default, Cloud Native Runtimes deploys an instance of Contour to the `contour-external` namespace and a second instance to the `contour-internal` namespace. One instance exposes services outside the cluster, and the other is for services that are private in your network.

The instructions on this page assume that, in your cluster, you have an existing Contour installation that you manage. To install Cloud Native Runtimes, you must allow it to use your existing Contour `CustomResourceDefinition`s.

If you already use a Contour instance to route requests from clients outside the cluster and you want to share resources with Knative, then you can choose to use your Contour for external services. Similarly, if you use a Contour instance to route requests from clients inside the cluster and want to share resources with Knative, choose to use your Contour for internal services.

If you do not have or do not want to reuse Contour instances for services, then Cloud Native Runtimes will create them.

Prerequisites

The following prerequisites are required to configure Cloud Native Runtimes with an existing Contour installation:

- Contour v1.14. To identify your cluster's Contour version, see Identify Your Contour Version below.
- Contour `CustomResourceDefinition`s versions:

Resource Name	Version
`extensionservices.projectcontour.io`	v1alpha1
`httpproxies.projectcontour.io`	v1
`tlscertificatedelegations.projectcontour.io`	v1

Identify Your Contour Version

To identify your cluster's Contour version:

1. Run:

```
export CONTOUR_NAMESPACE=contour-namespace
export CONTOUR_DEPLOYMENT=$(kubectl get deployment --namespace $CONTOUR_NAMESPA
CE --output name)
kubectl get $CONTOUR_DEPLOYMENT --namespace $CONTOUR_NAMESPACE --output jsonpat
h="{.spec.template.spec.containers[].image}"
kubectl get crds extensionservices.projectcontour.io --output jsonpath="{.statu
s.storedVersions}"
kubectl get crds httpproxies.projectcontour.io --output jsonpath="{.status.stor
edVersions}"
kubectl get crds tlscertificatedelegations.projectcontour.io --output jsonpath=
"{.status.storedVersions}"
```

   Where `CONTOUR-NAMESPACE` is the namespace where Contour is installed on your Kubernetes cluster.

Install Cloud Native Runtimes on a Cluster with Your Existing Contour Instances

To install Cloud Native Runtimes on a cluster with an existing Contour instance, you customize the installation script to reuse the `CustomResourceDefinition`s from your cluster. The variables that you set depend on whether or not you want Cloud Native Runtimes to reuse your existing Contour instance.

If you chose not to use your existing Contour installations, the script installs new instances of Contour to the namespaces `contour-external` and/or `contour-internal` using your cluster's `CustomResourceDefinition`s.

Note: If your Contour instance is removed or configured incorrectly, apps running on Cloud Native Runtimes will lose connectivity.

To install Cloud Native Runtimes on a cluster with existing Contour instances:

1. Decide if you want to reuse your Contour instance(s). See About Using Contour with Cloud Native Runtimes above.
2. Run the appropriate install command given below:

Use your Contour for external services?	Use your Contour for internal services?	Run:
No	No	`cnr_ingress__reuse_crds=true ./bin/install.sh`
Yes	No	`cnr_ingress__reuse_crds=true cnr_ingress__external__namespace=EXTERNAL-CONTOUR ./bin/install.sh`

Use your Contour for external services?	Use your Contour for internal services?	Run:

No	Yes	`cnr_ingress__reuse_crds=true`
		`cnr_ingress__internal__namespace=INTERNAL-CONTOUR`
		`./bin/install.sh`

Yes	Yes	`cnr_ingress__reuse_crds=true`
		`cnr_ingress__external__namespace=EXTERNAL-CONTOUR`
		`cnr_ingress__internal__namespace=INTERNAL-CONTOUR`
		`./bin/install.sh`

Where `EXTERNAL-CONTOUR` and `INTERNAL-CONTOUR` are the namespaces where Contour is installed on your Kubernetes cluster.

Make sure you type two underscore symbols (_) after `ingress`, `external`, and `internal`.

Integrating RabbitMQ with Cloud Native Runtimes for Tanzu

Cloud Native Runtimes for Tanzu supports using RabbitMQ as an event source to react to messages sent to a RabbitMQ exchange or as an event broker to distribute events within your app. The integration allows you to create:

- **A RabbitMQ broker**: A Knative Eventing broker backed by RabbitMQ. This broker uses RabbitMQ exchanges to store CloudEvents that are then routed from one component to another.
- **A RabbitMQ source:** An event source that translates external messages on a RabbitMQ exchange to CloudEvents, which can then be used with Knative Serving or Knative Eventing over HTTP.

About the RabbitMQ Operators

Before you can use or test RabbitMQ eventing on Cloud Native Runtimes, you need to install the following products on your Kubernetes cluster:

- RabbitMQ Cluster Kubernetes Operator v1.8.2. See Install RabbitMQ Cluster Kubernetes Operator v1.8.2 below.
- RabbitMQ Messaging Topology Kubernetes Operator v1.2.1. See Install RabbitMQ Messaging Topology Kubernetes Operator v1.2.1 below.
- cert-manager v1.5.3 and later. See Installation in the cert-manager documentation.

Install RabbitMQ Cluster Kubernetes Operator v1.8.2

The RabbitMQ Cluster Kubernetes Operator (cluster Operator) automates the lifecycle, creation, upgrade, and shutdown, of RabbitMQ clusters on Kubernetes:

To install the cluster Operator:

1. Create the `rabbitmq-system` namespace on your Kubernetes cluster where Cloud Native Runtimes is installed:

   ```
   kubectl create namespace rabbitmq-system
   ```

2. Define the following role binding:

   ```
   kubectl apply -f - << EOF
   kind: RoleBinding
   apiVersion: rbac.authorization.k8s.io/v1
   metadata:
   ```

```
  name: rabbitmq-cluster-operator-psp
  namespace: rabbitmq-system
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cnr-restricted
subjects:
- kind: ServiceAccount
  name: rabbitmq-cluster-operator
  namespace: rabbitmq-system
EOF
```

3. Install the RabbitMQ Cluster Kubernetes Operator v1.8.2 on your Kubernetes cluster where Cloud Native Runtimes is installed:

   ```
   kubectl apply -f https://github.com/rabbitmq/cluster-operator/releases/download
   /v1.8.2/cluster-operator.yml
   ```

   For general information about the RabbitMQ Cluster Kubernetes Operator, see rabbitmq/cluster-operator in GitHub.

Install RabbitMQ Messaging Topology Kubernetes Operator v1.2.1

The RabbitMQ Messaging Topology Kubernetes Operator (topology Operator) manages the topologies, or exchange types, of RabbitMQ clusters provisioned by the cluster Operator.

There are two YAMLs for the RabbitMQ Messaging Topology Kubernetes Operator:

- `messaging-topology-operator-with-certmanager.yaml`: Requires that you have cert-manager v1.5.3 installed
- `messaging-topology-operator.yaml`: Use if you want to generate and include your own certificates

To install the topology Operator:

1. Read the README.md for the topology Operator in GitHub and decide which YAML to install.
2. If you are installing the `messaging-topology-operator-with-certmanager.yaml`, then:
   1. Create the cert-manager namespace on your Kubernetes cluster where Cloud Native Runtimes is installed:

      ```
      kubectl create namespace cert-manager
      ```

   2. Define the following role binding:

      ```
      kubectl apply -f - << EOF
      kind: RoleBinding
      apiVersion: rbac.authorization.k8s.io/v1
      metadata:
        name: cert-manager-psp
        namespace: cert-manager
      roleRef:
      ```

```
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cnr-restricted
subjects:
- kind: ServiceAccount
  name: cert-manager
  namespace: cert-manager
- kind: ServiceAccount
  name: cert-manager-cainjector
  namespace: cert-manager
- kind: ServiceAccount
  name: cert-manager-webhook
  namespace: cert-manager
EOF
```

3. Define the following role binding:

```
kubectl apply -f - << EOF
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: rabbitmq-topology-psp
  namespace: rabbitmq-system
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cnr-restricted
subjects:
- kind: ServiceAccount
  name: messaging-topology-operator
  namespace: rabbitmq-system
EOF
```

4. Install the RabbitMQ Messaging Topology Operator v1.2.1 on your Kubernetes cluster where Cloud Native Runtimes is installed.

   For general information about the topology Operator v1.2.1, see the RabbitMQ Messaging Topology Operator v1.2.1 Release Notes in GitHub.

Next Steps

After completing these installations, you can:

- Verify your Knative Eventing installation using an example RabbitMQ broker. For instructions, see Verify Knative Eventing.
- Create a broker, producer, and a consumer to use RabbitMQ and Knative Eventing with your own app.

Verifying Your Installation

You can verify that your Cloud Native Runtimes for Tanzu installation was successful by testing Knative Serving, Knative Eventing, and TriggerMesh Sources for Amazon Web Services (SAWS).

Verify with a Private Registry

To verify your installation with a private registry, follow the procedure in Preparing to Create a Service to create a secret for your private registry before following the steps in the next section.

Verify Installation of Knative Serving, Knative Eventing, and TriggerMesh SAWS

To verify the installation of Knative Serving, Knative Eventing, and Triggermesh SAWS:

1. Create a namespace and environment variable for the test. Run:

```
export WORKLOAD_NAMESPACE='cnr-demo'
kubectl create namespace ${WORKLOAD_NAMESPACE}
```

2. Verify installation of the components that you intend to use:

To test...	Create...	For instructions, see...
Knative Serving	a test service.	Verifying Knative Serving
Knative Eventing	a broker, a producer, and a consumer.	Verifying Knative Eventing
TriggerMesh SAWS	an AWS source and trigger it.	Verifying TriggerMesh SAWS

3. Delete the namespace that you created for the tests. Run:

```
kubectl delete namespaces ${WORKLOAD_NAMESPACE}
unset WORKLOAD_NAMESPACE
```

Preparing to Create a Service

This topic explains how to create private registry credentials to prepare for creating a service. If you are using a private registry, you must add the same credentials you used for your private registry to the service account you used to create Knative services.

Note: You must complete following steps for each namespace where you create services.

1. Create a secret for your private registry credentials. Run:

```
kubectl -n "${WORKLOAD NS}" create secret docker-registry registry-credentials
```

```
\
--docker-server "$cnr_registry__server" \
--docker-username "$cnr_registry__username" \
--docker-password "$cnr_registry__password"
```

Where:

- `WORKLOAD_NS` is the namespace where you want to create services.
- $cnr_registry__server, $cnr_registry__username, and $cnr_registry__password environment variables have the same values you used to install Cloud Native Runtimes with a private registry. See Prerequisites in the Use Image Relocation with Cloud Native Runtimes section.

2. Add the secret to your namespace's default service account.

```
kubectl patch serviceaccount -n ${WORKLOAD_NS} default -p '{"imagePullSecrets":
 [{"name": "registry-credentials"}]}'
```

Verifying Knative Serving

This topic describes how to verify that Knative Serving was successfully installed.

About Verifying Knative Serving

To verify that Knative Serving was successfully installed, create an example Knative service and test it.

The procedure below shows you how to create an example Knative service using the Cloud Native Runtimes sample app, `hello-yeti`. This sample is custom built for Cloud Native Runtimes and is stored in the VMware Harbor registry.

Note: If you do not have access to the Harbor registry, you can use the Hello World - Go sample app in the Knative documentation.

Prerequisites

Before you verify Knative Serving, you must have created the `cnr-demo` namespace and variable. See step 1 of Verifying Your Installation.

Test Knative Serving

To create an example Knative service and use it to test Knative Serving:

1. If you are verifying on Tanzu Mission Control or vSphere 7.0 with Tanzu, then create a role binding in the `cnr-demo` namespace. Run:

```
kubectl apply -n "${WORKLOAD_NAMESPACE}" -f - << EOF
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: ${WORKLOAD_NAMESPACE}-psp
```

```
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cnr-restricted
subjects:
- kind: Group
  name: system:serviceaccounts:${WORKLOAD_NAMESPACE}
EOF
```

2. Deploy the sample app using the `kn` CLI. Run:

```
kn service create hello-yeti -n ${WORKLOAD_NAMESPACE} \
  --image projects.registry.vmware.com/tanzu_serverless/hello-yeti@sha256:17d64
0edc48776cfc604a14fbabf1b4f88443acc580052eef3a753751ee31652 --env TARGET='hello
-yeti'
```

If you are verifying on Tanzu Mission Control or vSphere 7.0 with Tanzu, then add `--user 1001` to the command above to run it as a non-root user.

3. Run one of the following commands to retrieve the external address for your ingress, depending on your IaaS:

IaaS:

- Tanzu Kubernetes Grid on AWS
- Tanzu Mission Control on AWS
- Amazon Elastic Kubernetes Service

Run:

```
export EXTERNAL_ADDRESS=$(kubectl get service envoy -n contour-external
\
-output 'jsonpath={.status.loadBalancer.ingress[0].hostname}')
```

IaaS:

- vSphere 7.0 on Tanzu
- Tanzu Kubernetes Grid on vSphere/Azure/GCP
- Tanzu Kubernetes Grid Integrated Edition
- Tanzu Mission Control on vSphere
- Azure Kubernetes Service
- Google Kubernetes Engine

Run:

```
export EXTERNAL_ADDRESS=$(kubectl get service envoy -n contour-external
\
-output 'jsonpath={.status.loadBalancer.ingress[0].ip}')
```

IaaS: Local Kubernetes Cluster:

- Docker desktop
- Kind
- Minikube

Run:

```
export EXTERNAL_ADDRESS='localhost:8080'
```

And, on another terminal, set up port forwarding. Run:

```
kubectl -n contour-external port-forward svc/envoy 8080:80
```

4. Connect to the app. Run:

```
curl -H "Host: hello-yeti.${WORKLOAD_NAMESPACE}.example.com" ${EXTERNAL_ADDRESS
}
```

If external DNS is correctly configured, you can also visit the URL in a web browser.

On success, you see a reply from our mascot, Carl the Yeti.

Delete the Example Knative Service

After verifying your serving installation, delete the example Knative service and unset the environment variable:

1. Run:

```
kn service delete hello-yeti -n ${WORKLOAD_NAMESPACE}
unset EXTERNAL_ADDRESS
```

2. If you created port forwarding in step 4 above, then terminate that process.

Verify Knative Eventing

This topic describes how to verify that Knative Eventing was successfully installed.

Note: The Knative eventing functionality is in beta. VMware does not recommend using Knative eventing functionality in a production environment.

About Verifying Knative Eventing

You can verify Knative Eventing by setting up a broker, creating a producer, and creating a consumer. If your installation was successful, you can create a test eventing workflow and see that the events appear in the logs.

You can use either an in-memory broker or a RabbitMQ broker to verify Knative Eventing:

- **RabbitMQ broker**: Using a RabbitMQ broker to verify Knative Eventing is a scalable and reliable way to verify your installation. Verifying with RabbitMQ uses methods similar to production environments.
- **In-memory broker**: Using an in-memory broker is a fast and lightweight way to verify that the basic elements of Knative Eventing are installed. An in-memory broker is not meant for production environments or for use with apps that you intend to take to production.

Prerequisites

Before you verify Knative Eventing, you must:

- Have created the `cnr-demo` namespace and variable. See step 1 of Verifying Your Installation.
- Create the following role binding in the `cnr-demo` namespace. Run:

```
kubectl apply -f - << EOF
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: ${WORKLOAD_NAMESPACE}-psp
  namespace: ${WORKLOAD_NAMESPACE}
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cnr-restricted
subjects:
- kind: Group
  name: system:serviceaccounts:${WORKLOAD_NAMESPACE}
EOF
```

Prepare the RabbitMQ Environment

If you are using a RabbitMQ broker to verify Knative Eventing, follow the procedure in this section. If you are verifying with the in-memory broker, skip to Verify Knative Eventing.

To prepare the RabbitMQ environment before verifying Knative Eventing:

1. Set up the RabbitMQ integration as described in Integrating RabbitMQ with Cloud Native Runtimes for Tanzu.
2. On the Kubernetes cluster where Cloud Native Runtimes is installed, deploy a RabbitMQ cluster using the RabbitMQ Cluster Operator by running:

```
kubectl apply -f - << EOF
apiVersion: rabbitmq.com/v1beta1
kind: RabbitmqCluster
metadata:
  name: my-rabbit
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  replicas: 1
  override:
    statefulSet:
      spec:
        template:
          spec:
            securityContext: {}
            containers: []
            initContainers:
            - name: setup-container
              securityContext:
                runAsUser: 999
                runAsGroup: 999
EOF
```

Note: The `override` section can be omitted if your cluster allows containers to run as `root`.

Verify Knative Eventing

To verify installation of Knative Eventing create and test a broker, procedure, and consumer in the `cnr-demo` namespace:

1. Create a broker.

   For the RabbitMQ broker. Run:

   ```
   kubectl apply -f - << EOF
   apiVersion: eventing.knative.dev/v1
   kind: Broker
   metadata:
     name: default
     namespace: ${WORKLOAD_NAMESPACE}
     annotations:
       eventing.knative.dev/broker.class: RabbitMQBroker
   spec:
     config:
       apiVersion: rabbitmq.com/v1beta1
       kind: RabbitmqCluster
       name: my-rabbit
       namespace: ${WORKLOAD_NAMESPACE}
   EOF
   ```

   For the in-memory broker. Run:

   ```
   kubectl create -f - <<EOF
   apiVersion: eventing.knative.dev/v1
   kind: Broker
   metadata:
     name: default
     namespace: ${WORKLOAD_NAMESPACE}
   EOF
   ```

2. Create a consumer for the events. Run:

   ```
   cat <<EOF | kubectl create -f -
   apiVersion: serving.knative.dev/v1
   kind: Service
   metadata:
     name: event-display
     namespace: ${WORKLOAD_NAMESPACE}
   spec:
     template:
       spec:
         containers:
           - image: gcr.io/knative-releases/knative.dev/eventing-contrib/cmd/event
   _display
   EOF
   ```

3. Create a trigger. Run:

   ```
   kubectl apply -f - << EOF
   ```

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: event-display
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  broker: default
  subscriber:
    ref:
     apiVersion: serving.knative.dev/v1
     kind: Service
     name: event-display
     namespace: ${WORKLOAD_NAMESPACE}
EOF
```

4. Create a producer. Run:

```
cat <<EOF | kubectl create -f -
apiVersion: sources.knative.dev/v1
kind: PingSource
metadata:
  name: test-ping-source
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  schedule: "*/1 * * * *"
  data: '{"message": "Hello Eventing!"}'
  sink:
    ref:
      apiVersion: eventing.knative.dev/v1
      kind: Broker
      name: default
      namespace: ${WORKLOAD_NAMESPACE}
EOF
```

5. Verify that the event appears in your consumer logs. Run:

```
kubectl logs -l serving.knative.dev/service=event-display -c user-container -n
${WORKLOAD_NAMESPACE} --since=10m --tail=50
```

Delete the Eventing Resources

After verifying your serving installation, clean up by deleting the resources used for the test:

1. Delete the eventing resources:

```
kubectl delete pingsource/test-ping-source -n ${WORKLOAD_NAMESPACE}
kubectl delete trigger/event-display -n ${WORKLOAD_NAMESPACE}
kubectl delete kservice/event-display -n ${WORKLOAD_NAMESPACE}
kubectl delete broker/default -n ${WORKLOAD_NAMESPACE}
```

2. If you created a RabbitMQ cluster:

```
kubectl delete rabbitmqcluster/my-rabbit -n ${WORKLOAD_NAMESPACE}
```

3. Delete the role binding:

```
kubectl delete rolebinding/${WORKLOAD_NAMESPACE}-psp -n ${WORKLOAD_NAMESPACE}
```

Verifying TriggerMesh SAWS

This topic describes how to verify that TriggerMesh Sources for Amazon Web Services (SAWS) was installed successfully.

TriggerMesh SAWS allows you to consume events from your AWS services and send them to workloads running in your cluster.

Cloud Native Runtimes for Tanzu includes an installation of the Triggermesh SAWS controller and CRDs. You can find the controller in the `triggermesh` namespace.

For general information about TriggerMesh SAWS, see aws-event-sources in GitHub.

The procedure below shows you how to test TriggerMesh SAWS using the example of an event source for Amazon CodeCommit. If you want to test using a different AWS service, see samples in GitHub. The basic steps are the same, regardless of the AWS service you choose: create a broker, trigger, and consumer and then test.

Prerequisites

Before you verify TriggerMesh SAWS with AWS CodeCommit, you must have:

- An AWS service account
- An AWS CodeCommit repository with push and pull access

Verify TriggerMesh SAWS

To verify TriggerMesh SAWS with AWS CodeCommit:

1. Create a broker:

```
kubectl apply -f - << EOF
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: broker
  namespace: ${WORKLOAD_NAMESPACE}
EOF
```

2. Create a trigger:

```
kubectl apply -f - << EOF
---
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: trigger
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  broker: broker
  subscriber:
    ref:
```

```
    apiVersion: serving.knative.dev/v1
    kind: Service
    name: consumer
    namespace: ${WORKLOAD_NAMESPACE}
EOF
```

3. Create a consumer:

```
kubectl apply -f - << EOF
---
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: consumer
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  template:
    spec:
      containers:
        - image: gcr.io/knative-releases/knative.dev/eventing-contrib/cmd/event
_display
EOF
```

4. Add an AWS service account secret:

   Note: Kubernetes uses base64 encoding to store secrets. Convert your AWS keys to base64 before proceeding. For more information about base64 encoding, see Distribute Credentials Securely Using Secrets in the Kubernetes documentation.

```
kubectl apply -f - << EOF
apiVersion: v1
data:
  aws_access_key_id: ID-BASE64
  aws_secret_access_key: KEY-BASE64
kind: Secret
metadata:
  name: awscreds
  namespace: ${WORKLOAD_NAMESPACE}
type: Opaque
EOF
```

   Where:

   - `ID-BASE64` is the AWS access key for your AWS service account encoded in base64.
   - `KEY-BASE64` is your AWS access key for your AWS service account encoded in base64.

5. Create the AWSCodeCommitSource:

```
kubectl apply -f - << EOF
apiVersion: sources.triggermesh.io/v1alpha1
kind: AWSCodeCommitSource
metadata:
  name: source
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  arn: ARN
```

```
  branch: BRANCH

  eventTypes:
    - push
    - pull_request

  credentials:
    accessKeyID:
      valueFromSecret:
        name: awscreds
        key: aws_access_key_id
    secretAccessKey:
      valueFromSecret:
        name: awscreds
        key: aws_secret_access_key

  sink:
    ref:
      apiVersion: eventing.knative.dev/v1
      kind: Broker
      name: broker
      namespace: ${WORKLOAD_NAMESPACE}
EOF
```

Where:

- `ARN` is Amazon Resource Name (ARN) of your CodeCommit repository. For example, `arn:aws:codecommit:eu-central-1:123456789012:triggermeshtest`.
- `BRANCH` is the branch of your CodeCommit repository that you want the trigger to watch. For example, `main`.

6. Create an event by pushing a commit to your CodeCommit repository.
7. Watch the consumer logs to see that the event appears after a minute:

```
kubectl logs -l serving.knative.dev/service=consumer -c user-container -n ${WOR
KLOAD_NAMESPACE} --since=10m --tail=50
```

Developing Locally on Kind

This topic explains how to configure Cloud Native Runtimes for Tanzu to develop locally on kind. Developing locally on kind allows you to visit URLs generated by Knative for your services in a browser, or use curl to view URLs.

Prerequisites

Before you follow the procedures below, you must have access to:

- Docker Hub
- The internet

Because this feature uses the CoreDNS container image hosted on Docker Hub, you need internet access to pull the image.

Configure Your Local Kind Cluster

To develop locally with Cloud Native Runtimes on kind, you need to create a cluster with ports mapped from your host machine to your kind cluster.

To create a cluster and map ports from your host machine to your kind cluster:

1. Create and save a file named `kind-cluster.yaml` that contains following code:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
 - role: control-plane
 - role: worker
   extraPortMappings:
   - containerPort: 31443 # expose port 31443 of the node to port 80 on the hos
t for use later by Contour ingress (envoy)
     hostPort: 443
   - containerPort: 31080 # expose port 31080 of the node to port 80 on the hos
t for use later by Contour ingress (envoy)
     hostPort: 80
   - containerPort: 30053 # expose CoreDNS port to port 53 on the host machine
     hostPort: 53
     protocol: udp
   - containerPort: 30053 # expose CoreDNS port to port 53 on the host machine
     hostPort: 53
     protocol: tcp
```

2. Create a kind cluster. Run:

```
kind create cluster --config kind-cluster.yaml
```

Install Cloud Native Runtimes Locally

After you create a local cluster with kind, you can install Cloud Native Runtimes on that cluster with a flag to enable local DNS. The install script and flag adds a CoreDNS deployment in the `cloud-native-runtimes` namespace.

1. Ensure that your Kubernetes client targets your local kind cluster. Run:

   ```
   kubectl cluster-info
   ```

   If your Kubernetes client targets your local cluster, the return includes your local cluster information.

2. From the `cloud-native-runtimes` directory, run the installation script:

   ```
   cnr_provider=local cnr_local_dns__enable=true ./bin/install.sh
   ```

Note: Knative Serving routes use `example.com` as the default domain. See Setting up a custom domain in the Knative documentation. If you want to use the Local DNS feature with your custom domain, specify your custom domain in the install command: for example, `cnr_provider=local cnr_local_dns__enable=true cnr_local_dns__domain=mydomain.com`.

Test Your CoreDNS

You can verify that your CoreDNS is configured correctly using Dig. For information about using Dig DNS lookup, see Dig in the Google Admin Toolbox.

To use Dig to verify your CoreDNS:

1. Run:

   ```
   dig test.YOUR-DOMAIN @127.0.0.1
   ```

   Where `YOUR-DOMAIN` is the name of your domain.

   If your CoreDNS is configured correctly, the return includes the IPv4 loopback address `127.0.0.1`.

Set the Host Machine DNS

To use your CoreDNS to resolve queries, set your host machine to use the IPv4 loopback address `127.0.0.1` as the DNS server.

If you are using DHCP, disconnecting and reconnecting your network connection resets your nameserver.

If your local cluster is deleted, your normal DNS queries continue working because you have other DNS servers configured.

Set on MacOS

To set the host machine DNS on MacOS:

1. Create a directory called `/etc/resolver`, if the directory does not already exist.
2. Inside this directory, create a file with the same name as your domain. If you are using the default domain, the filename is `/etc/resolver/example.com`.
3. Populate the file with the following nameserver to let your machine know to send any requests for your domain to localhost: `nameserver 127.0.0.1`.

Set on Linux

To set the host machine DNS on Linux:

1. Add the following line to the top of your `/etc/resolv.conf` file:

   ```
   nameserver 127.0.0.1
   ```

   The nameserver entries are an ordered list, so anything not served by localhost continues down the file to try other entries.

Note: Setting `127.0.0.1` as your primary DNS server can break `nslookup` on Linux, because it uses only the primary DNS server.

Set on Windows

To set the host machine DNS on Windows:

1. Complete the steps under Change your DNS servers settings, and enter the IPv4 loopback address `127.0.0.1` as the preferred DNS server. See Change your DNS servers settings in the Google Public DNS documentation.
2. (Optional) Input another preferred DNS server as the alternate DNS server.
3. (Optional) You may need to disable IPv6 for the resolution to be successful.

Note: Setting `127.0.0.1` as your primary DNS server can break `nslookup` on Windows, because it uses only the primary DNS server.

Reset the Host Machine DNS

You can reset the host machine DNS to another IPv4 loopback address using the procedure for your operating system below.

Reset on MacOS

To reset the host machine DNS on MacOS:

1. Remove the `/etc/resolver/YOUR-DOMAIN` file.

Reset on Linux

To reset the host machine DNS on Linux:

1. Remove the following line from the top of your `/etc/resolv.conf` file:

   ```
   nameserver 127.0.0.1
   ```

Reset on Windows

To reset the host machine DNS on Windows:

1. Complete the steps under Change your DNS servers settings, and enter the IPv4 loopback address as your original preferred DNS server address. See Change your DNS servers settings in the Google Public DNS documentation.
2. (Optional) Enable IPv6 if you disabled it when you set the host machine DNS.

Enabling Automatic TLS Certificate Provisioning for Cloud Native Runtimes for Tanzu

You can configure Cloud Native Runtimes for Tanzu to automatically obtain and renew TLS certificates for your workloads. Automatic TLS certificate provisioning allows you to secure your clusters and domains without manually generating or renewing certificates. Automatic TLS certificate provisioning reduces the manual certificate workload for admins and developers.

Cloud Native Runtimes supports both `HTTP01` and `DNS01` cert-manager challenge types. For more information about `cert-manager` challenge types, see ACME in the cert-manager documentation.

VMware recommends using Let's Encrypt as your certificate authority. However, you can integrate Cloud Native Runtimes with any ACME compatible certificate authority.

Prerequisites

You can enable HTTPS with Automatic TLS certificate provisioning for Cloud Native Runtimes.

You need the following prerequisites to use secure HTTPS connections with automatic TLS certificate provisioning:

- A cluster configured to use a custom domain. See Setting Up a Custom Domain in the Knative documentation.
- A DNS provider configured with your domain name.
- cert-manager version 1.0.0 or later. See Installing cert-manager for TLS certificates in the Knative documentation.
- `HTTP01` **challenges**: An internet-reachable cluster.
- `DNS01` **challenges**: API access to set DNS records.

Enable Auto TLS Using an HTTP01 Challenge

You can use the `HTTP01` challenge type to validate a domain with Cloud Native Runtimes. The `HTTP01` challenge requires that your load balancer be reachable from the internet via HTTP.

Note: With the `HTTP01` challenge type, you provision a certificate for each service.

To enable automatic TLS certificate provisioning using a `HTTP01` challenge, do the following:

1. Create a cert-manager `Issuer` or `ClusterIssuer` for the `HTTP01` challenge. See Issuer in the cert-manager documentation. The following example creates a `ClusterIssuer` using the Let's Encrypt Certificate Authority. See Let's Encrypt. To use a `ClusterIssuer` for the `HTTP01`

challenge, run:

```
kubectl apply -f - <<EOF
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-http01-issuer
spec:
  acme:
    privateKeySecretRef:
      name: letsencrypt
    server: https://acme-v02.api.letsencrypt.org/directory
    solvers:
    - http01:
        ingress:
          class: contour
EOF
```

2. To validate that your `ClusterIssuer` was created successfully, run:

```
kubectl get clusterissuer letsencrypt-http01-issuer --output yaml
```

3. Edit your `config-certmanager` ConfigMap in the `knative-serving` namespace to reference the `ClusterIssuer` you created. Run:

```
kubectl edit configmap config-certmanager --namespace knative-serving
```

4. To define which ClusterIssuer will be used by Knative to issue certificates, add the following `issuerRef` block under the `data` section of the `config-certmanager` ConfigMap:

```
...
data:
...
 issuerRef: |
   kind: ClusterIssuer
   name: letsencrypt-http01-issuer
```

5. To validate that your ConfigMap was updated successfully, run:

```
kubectl get configmap config-certmanager --namespace knative-serving --output j
sonpath="{.data.issuerRef}"
```

6. Edit the `config-network` ConfigMap in the `knative-serving` namespace to enable automatic TLS certificate provisioning and specify how HTTP requests are handled. Run:

```
kubectl edit configmap config-network --namespace knative-serving
```

Note: For `HTTP01` challenges, the `httpProtocol` field must be set to `Enabled` for the cluster to accept `HTTP01` challenge requests.

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: config-network
```

```
 namespace: knative-serving
data:
 ...
 autoTLS: Enabled
 ...
 httpProtocol: Enabled
 ...
```

7. To validate that your ConfigMap was updated successfully, run:

```
kubectl get configmap config-network --namespace knative-serving --output jsonp
ath="{.data.autoTLS}"
kubectl get configmap config-network --namespace knative-serving --output jsonp
ath="{.data.httpProtocol}"
```

8. Verify that your automatic TLS certificate instance is configured correctly by deploying a sample app. See Verify Auto TLS in the Knative documentation.

Enable Auto TLS Using a DNS01 Challenge

The `DNS01` challenge validates that you control your domain's DNS by accessing and updating your domain's TXT record. You need to provide a `cert-manager` with your DNS API credentials. For a list of DNS01 providers supported for the ACME `Issuer`, see the cert-manager documentation.

Note

: You can provision certificates **per service** only.

To enable automatic TLS certificate provisioning using a `DNS01` challenge, do the following:

1. Set up credentials for `cert-manager` to access your DNS records. For information about setting up credentials for your ACME `Issuer` supported DNS provider, see Supported DNS01 providers in the cert-manager documentation. In the next step, you create an Issuer on `cert-manager` with the configuration you set up.
2. Create a cert-manager Issuer or ClusterIssuer for DNS01 challenge on the `cert-manager` Issuer you set up in the previous step. The following example uses Let's Encrypt and Google Cloud DNS. For information about other DNS providers supported by cert-manager, see the cert-manager documentation. The `Issuer` assumes that your Kubernetes secret holds credentials for the service account created. Run the following command to apply the ClusterIssuer:

```
kubectl apply --filename - <<EOF
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-dns-issuer
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    # This will register an issuer with LetsEncrypt.
    email: MY-EMAIL
    privateKeySecretRef:
```

```
      # Set privateKeySecretRef to any unused secret name.
      name: letsencrypt-dns-issuer
    solvers:
    - dns01:
        cloudDNS:
          project: $PROJECT_ID
          # Set this to the secret that we publish our service account key
          # in the previous step.
          serviceAccountSecretRef:
            name: cloud-dns-key
            key: key.json
EOF
```

Where `MY-EMAIL` is your email address.

3. To verify that your ClusterIssuer is created successfully, run:

```
kubectl get clusterissuer letsencrypt-dns-issuer --output yaml
```

4. Edit your `config-certmanager` ConfigMap in the `knative-serving` namespace to reference the ClusterIssuer created in the previous step. Run:

```
kubectl edit configmap config-certmanager --namespace knative-serving
```

5. Add an `issuerRef` block under the `data` section of your ConfigMap. This defines the ClusterIssuer Knative uses to issue certificates. Run:

```
...
data:
...
 issuerRef: |
   kind: ClusterIssuer
   name: letsencrypt-dns-issuer
```

6. To validate that your file was updated successfully, run:

```
kubectl get configmap config-certmanager --namespace knative-serving --output j
sonpath="{.data.issuerRef}"
```

7. To enable automatic TLS certificate provisioning and specify how HTTP requests are handled, edit your `config-network` ConfigMap in the `knative-serving` namespace:

```
kubectl edit configmap config-network --namespace knative-serving
```

Note: When using the DNS01 challenge type, the `httpProtocol` field must be set to `Enabled`.

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: config-network
 namespace: knative-serving
data:
 ...
 autoTLS: Enabled
 ...
```

```
  httpProtocol: Enabled
  ...
```

8. Validate that your file was updated successfully. Run:

```
kubectl get configmap config-network --namespace knative-serving --output jsonp
ath="{.data.autoTLS}"
kubectl get configmap config-network --namespace knative-serving --output jsonp
ath="{.data.httpProtocol}"
```

9. Verify that your automatic TLS certificate instance is functioning correctly by deploying a sample app. See Verify Auto TLS in the Knative documentation.

Configuring Observability for Cloud Native Runtimes for Tanzu

You can set up integrations with third-party observability tools to use logging, metrics, and tracing with Cloud Native Runtimes for Tanzu. These observability integrations allow you to monitor and collect detailed metrics from your clusters on Cloud Native Runtimes. You can collect logs and metrics for all workloads running on a cluster. This includes Cloud Native Runtimes components or any apps running on Cloud Native Runtimes. The integrations in this topic are recommended by VMware, however you can use any Kubernetes compatible logging, metrics, and tracing platforms to monitor your cluster workload.

Logging

You can collect and forward logs for all workloads on a cluster, including Cloud Native Runtimes components or any apps running on Cloud Native Runtimes. You can use any logging platform that is compatible with Kubernetes to collect and forward logs for Cloud Native Runtimes workloads. VMware recommends using Fluent Bit to collect logs and then forward logs to vRealize. The following sections describe configuring logging for Cloud Native Runtimes with Fluent Bit and vRealize as an example.

Configure Logging with Fluent Bit

You can use Fluent Bit to collect logs for all workloads on a cluster, including Cloud Native Runtimes components or any apps running on Cloud Native Runtimes. For more information about using Fluent Bit logs, see Fluent Bit Kubernetes Logging in the Fluent Bit documentation.

Fluent Bit lets you collect logs from Kubernetes containers, add Kubernetes metadata to these logs, and forward logs to third-party log storage services. For more information about collecting logs, see Logging in the Knative documentation.

If you are using Tanzu Mission Control (TMC), vSphere 7.0 with Tanzu, or Tanzu Kubernetes Cluster to manage your cloud native environment, you must set up a role binding that grants required permissions to Fluent Bit containers in order to configure logging with any integration. Then, you can follow the instructions in the Fluent Bit documentation to complete the logging configuration. For more information about configuring Fluent Bit logging, see Installation in the Fluent Bit documentation.

To configure logging with Fluent Bit for your Cloud Native Runtimes environment:

1. VMware recommends that you add any integrations to the `ConfigMap` in both your Knative Serving and Knative Eventing namespaces. Follow the logging configuration steps in the Fluent Bit documentation to create the `Namespace`, `ServiceAccount`, `Role`, `RoleBinding`, and `ConfigMap`. To view these steps, see Installation in the Fluent Bit documentation.

2. If you are using TMC, vSphere with Tanzu, or Tanzu Kubernetes Cluster to manage your cloud native environment, create a role binding in the Kubernetes namespace where your integration will be deployed to grant permission for privileged Fluent Bit containers. For information about creating a role binding on a Tanzu platform, see Add a Role Binding in the TMC documentation. For information about viewing your Kubernetes namespaces, see Viewing Namespaces in the Kubernetes documentation. Create the following role binding:

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: RoleBinding
   metadata:
     name: fluentbit-psp-rolebinding
     namespace: FLUENTBIT-NAMESPACE
   roleRef:
     apiGroup: rbac.authorization.k8s.io
     kind: ClusterRole
     name:  PRIVILEGED-CLUSTERROLE
   subjects:
   - kind: ServiceAccount
     name: FLUENTBIT-SERVICEACCOUNT
     namespace: FLUENTBIT-NAMESPACE
   ```

   Where:

   - `FLUENTBIT-NAMESPACE` is your Fluent Bit namespace.
   - `PRIVILEGED-CLUSTERROLE` is the name of your privileged cluster role.
   - `FLUENTBIT-SERVICEACCOUNT` is your Fluent Bit service account.

3. To verify that you have configured logging successfully, run the following to access logs through your web browser:

   ```
   kubectl port-forward --namespace logging service/log-collector 8080:80
   ```

   For more information about accessing Fluent Bit logs, see Logging in the Knative documentation.

Forward Logs to vRealize

After you configure log collection, you can forward logs to log management services. vRealize Log Insight is one service you can use with Cloud Native Runtimes. vRealize Log Insight is a scalable log management solution that provides log management, dashboards, analytics, and third-party extensibility for infrastructure and apps. For more information about vRealize Log Insight, see the VMware vRealize Log Insight Documentation.

To forward logs from your Cloud Native Runtimes environment to vRealize, you can use a new or existing instance of Tanzu Kubernetes Cluster. For information about how to configure log forwarding to vRealize from Tanzu Kubernetes Cluster, see the Configure Log forwarding from VMware Tanzu Kubernetes Cluster to vRealize Log Insight Cloud blog.

Metrics

Cloud Native Runtimes integrates with Prometheus and Tanzu Observability by Wavefront to collect metrics on components or apps. For more information about integrating with Prometheus, see

Overview in the Prometheus documentation and Kubernetes Integration in the Wavefront documentation.

You can configure Prometheus endpoints on Cloud Native Runtimes components in order to be able to collect metrics on your components or apps. For information on annotations required to collect metrics on apps, see Per-Pod Prometheus Annotations in the WeaveWorks documentation.

You can use annotation based discovery with Prometheus to define which Kubernetes objects in your Cloud Native Runtimes environment to add metadata and collect metrics in a more automated way. For more information about using annotation based discovery, see Annotation based discovery in GitHub.

You can then use the Wavefront Collector for Kubernetes collector to dynamically discover and scrape pods with the `prometheus.io/scrape` annotation prefix. For information about the Kubernetes collector, see Wavefront Collector for Kubernetes in GitHub.

Note

: All Cloud Native Runtimes related metrics are emitted with the prefix `tanzu.vmware.com/cloud-native-runtimes.*`.

Tracing

Tracing is a method for understanding the performance of specific code paths in apps as they handle requests. You can configure tracing to collect performance metrics for your apps or Cloud Native Runtimes components. You can trace which aspects of Cloud Native Runtimes and workloads running on Cloud Native Runtimes are performing poorly.

Configuring Tracing

You can configure tracing for your apps on Cloud Native Runtimes. To do this, you configure tracing for both Knative Serving and Eventing by editing the ConfigMap for your Knative namespace.

To configure tracing, do the following:

1. Configure the `config-tracing` ConfigMap in your Knative component namespace. VMware recommends that you add any integrations to the ConfigMap in both your Serving and Eventing namespaces. For information on how to enable request traces in each component, see the following Knative documentation:
    - Serving. See Accessing request traces.
    - Eventing. See Accessing CloudEvent traces.

Forwarding Trace Data to a Data Visualization Tool

You can use the OpenTelemetry integration with Tanzu Observability by Wavefront to forward trace data to Tanzu Observability by Wavefront. For information about forwarding trace data, see Sending Metrics Data to Wavefront in the Wavefront documentation.

To configure to send trace data to Cloud Native Runtimes tracing with Tanzu Observability by Wavefront and the OpenTelemetry integration, do the following:

1. Use the following documentation to configure the OpenTelemetry Integration to send trace data to with Cloud Native Runtimes. For more information about sending trace data with OpenTelemetry, see OpenTelemetry Integration in the Wavefront documentation.
2. Deploy the Wavefront Proxy. For more information about wavefront proxies, see Deploy a Wavefront Proxy in Kubernetes in the Wavefront documentation.
    - Use the following .yaml file to install the Wavefront proxy in your Kubernetes cluster: wavefront.yaml.
    - Provide the URL of your Wavefront instance and a Wavefront token.
    - Uncomment the lines indicated in the yaml file to enable consumption of Zipkin traces.

Sending Trace Data to an Observability Platform

You can send trace data to an observability and analytics platform to view and monitor your trace data in dashboards.

One way to do this is to integrate Tanzu Observability by Wavefront with your Cloud Native Runtimes environment. To view your trace data in Wavefront, you configure Cloud Native Runtimes to send traces to the Wavefront proxy and then configure the Wavefront proxy to consume Zipkin spans.

For more information about using Zipkin for tracing, see the Zipkin documentation.

You can send trace data from Cloud Native Runtimes to Wavefront by using Zipkin as the backend and defining the Zipkin endpoint as the Wavefront proxy URL listening over port `9411`. You configure Cloud Native Runtimes to send traces directly to the Wavefront proxy by editing the `zipkin-endpoint` property in the ConfigMap to point to the Wavefront proxy URL. You can configure the Wavefront proxy to consume Zipkin spans by listening to port 9411.

To send trace data to Tanzu Observability by Wavefront:

1. Edit the ConfigMap to enable the Zipkin tracing integration. VMware recommends that you add any integrations to the ConfigMap in both your Serving and Eventing namespaces. Edit the Knative config-tracing ConfigMap to set `backend` to `zipkin` and pass the Wavefront proxy URL in the zipkin-endpoint field:

```
Kubectl edit configmap config-tracing —namespace knative-serving apiVersion: v1
kind: ConfigMap
metadata:
name: config-tracing
...
data:
backend: "zipkin"
zipkin-endpoint: "http://wavefront-proxy.default.svc.cluster.local:9411/api/v2/
spans"  ...
```

Use Wavefront Dashboards

Cloud Native Runtimes provides two Wavefront dashboards in JSON format. You can use these dashboard to monitor your apps and investigate performance issues. For information about configuring dashboards, see Create and Customize Dashboards in the Wavefront documentation.

The following Wavefront dashboards are compatible with Cloud Native Runtimes: - Application Operator Service View. See `app-operator-service-view.json` in the Cloud Native Runtimes installation .tar file. - Application Operator Revision View. See `app-operator-revision-view.json` in the Cloud Native Runtimes installation .tar file.

To import a dashboard JSON file, use one of the following methods: - Wavefront REST API - Wavefront CLIs.

You must provide the URL of your Wavefront instance and a Wavefront token. For more information about Wavefront tokens, see Generating an API Token in the Wavefront documentation.

Import Wavefront Dashboards

You can import the Wavefront dashboards using either the Wavefront API or the Ruby Wavefront CLI. For more information about Wavefront dashboard, see Import Dashboards with the Wavefront API or Import with a Ruby Wavefront CLI below.

Import Dashboards with the Wavefront API

To import a Wavefront dashboard with the Wavefront API, run:

```
curl -H "Content-Type: application/json" -H 'Authorization: Bearer <wavefront-token>'
\
    https://<wavefront-instance>.wavefront.com/api/v2/dashboard -d @observability/wave
front/app-operator-service-view.json

curl -H "Content-Type: application/json" -H 'Authorization: Bearer <wavefront-token>'
\
    https://<wavefront-instance>.wavefront.com/api/v2/dashboard -d @dashboards/wavefro
nt/app-operator-revision-view.json
```

After you run the import code, the Wavefront API creates two dashboards with the following names and URLs:

- **Title**: `Cloud Native Runtimes App Operator - Service View`

  URL: `https://<wavefront-instance>.wavefront.com/dashboards/App-Operator-Service-Level`

- **Title**: `Cloud Native Runtimes App Operator - Revision View`

  URL: `https://<wavefront-instance>.wavefront.com/dashboards/App-Operator-Revision-Level`

Import with the Ruby Wavefront CLI

To import a Wavefront dashboard with the Ruby Wavefront CLI, run:

```
export WAVEFRONT_TOKEN=<wavefront-token>
export WAVEFRONT_ENDPOINT=<wavefront-instance>.wavefront.com

wf config envvars
wf dashboard import observability/wavefront/app-operator-service-view.json
wf dashboard import dashboards/wavefront/app-operator-revision-view.json
```

After you run the import code, the Ruby Wavefront CLI creates two dashboards with a name and URL.

The Service View of the Cloud Native Runtimes App Operator dashboard will have the following title and URL:

- **Title**: `Cloud Native Runtimes App Operator - Service View`

  URL: `https://<wavefront-instance>.wavefront.com/dashboards/App-Operator-Service-Level`

The Revision View of the Cloud Native Runtimes App Operator dashboard will have the following title and URL:

- **Title**: `Cloud Native Runtimes App Operator - Revision View`

  URL: `https://<wavefront-instance>.wavefront.com/dashboards/App-Operator-Revision-Level`

Configuring Cloud Native Runtimes for Tanzu with Avi Vantage

You can configure Cloud Native Runtimes for Tanzu to integrate with Avi Vantage. Avi Vantage is a multi-cloud platform that delivers features such as load balancing, security, and container ingress services. The Avi Controller provides a control plane. Avi Service Engines provides a data plane. The Avi Service Engines forward incoming traffic to your Kubernetes cluster's Envoy pods, which are created and managed by Contour.

For information about Avi Vantage, see Avi Documentation.

Integrate Avi Vantage with Cloud Native Runtimes

This procedure assumes that you have already installed Cloud Native Runtimes. See Installing Cloud Native Runtimes. If you have not already installed Cloud Native Runtimes, you need to create a cluster, run the install script, and set up Contour in addition to the steps below. For more information about installing with Contour, see Installing Cloud Native Runtimes with an Existing Contour Installation.

To configure Cloud Native Runtimes with Avi Vantage, do the following:

1. Deploy the Avi Controller on any Avi supported infrastructure providers. For a list of Avi supported providers, see Avi Installation Guides. For more information about deploying an Avi Controller, see Install Avi Kubernetes Operator in the Avi Vantage documentation.
2. Deploy the Avi Kubernetes Operator to your Kubernetes cluster where Cloud Native Runtimes is hosted. See Install AKO for Kubernetes in the Avi Vantage documentation.
3. Connect to a test app and verify that it is reachable. Run:

   ```
   "curl -H KNATIVE-SERVICE-DOMAIN" ENVOY-IP
   ```

   Where:

   - `KNATIVE-SERVICE-DOMAIN` is the name of your domain.
   - `ENVOY-IP` is the IP address of your Envoy instance.

   For more information about deploy a sample application and connect to the application, see Test Knative Serving.
4. (Optional) Create a DNS record that will configure your KService URL to point to the Avi Service Engines, and resolve to the external IP of the Envoy. You can create a DNS record on any platform that supports DNS services. Refer to the documentation for your DNS service platform for more information.

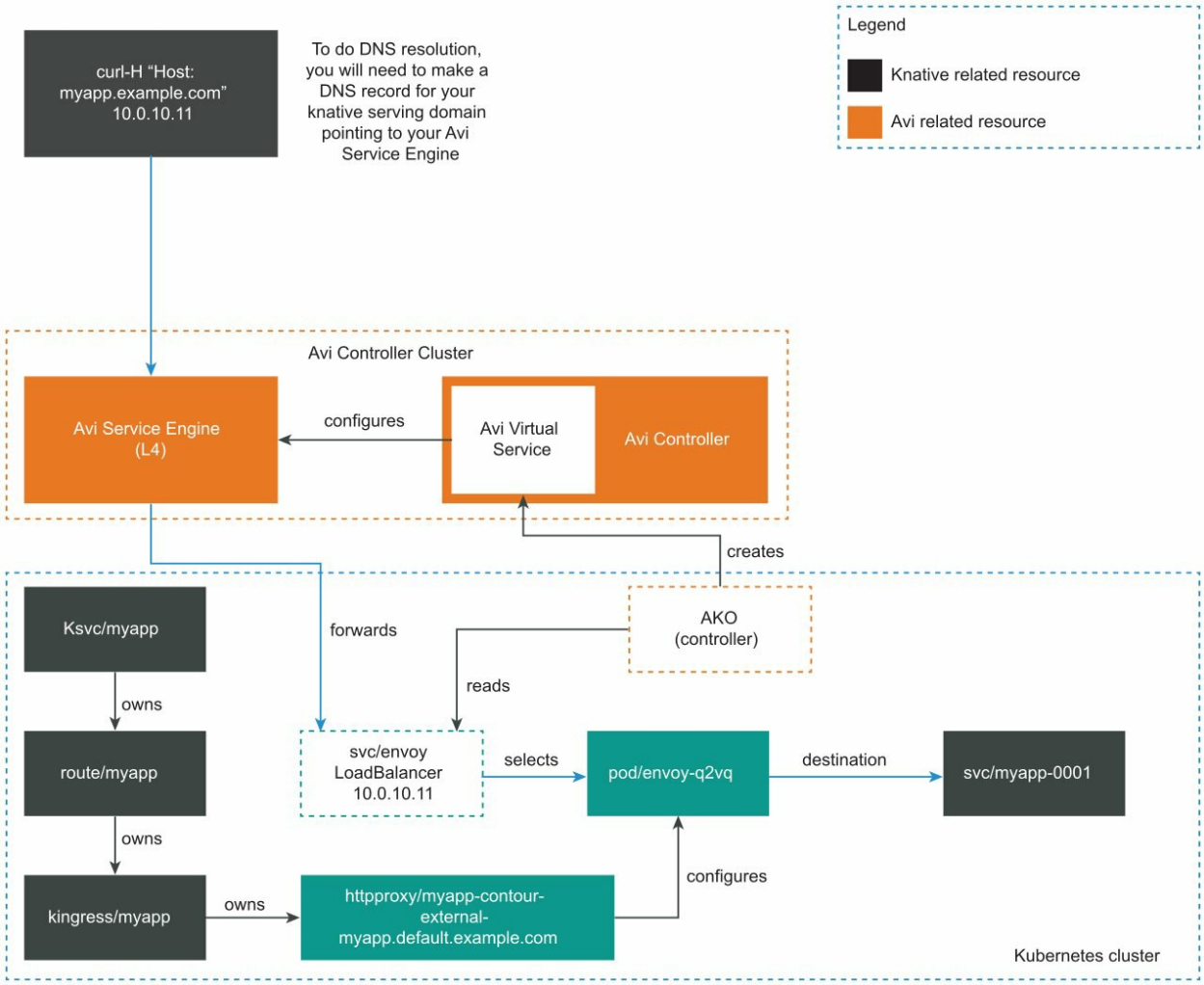
   To get the KService URL, run:

```
kn route describe APP-NAME | grep "URL"
```

To get Envoy's external IP, follow step 3 in Test Knative Serving in *Verifying your Serving Installation*.

About Routing with Avi Vantage and Cloud Native Runtimes

The following diagram shows how Avi Vantage integrates with Cloud Native Runtimes:



When Contour creates a Kubernetes LoadBalancer service for Envoy, the Avi Kubernetes Operator (AKO) sees the new LoadBalancer service. Then Avi Controller creates a Virtual Service. For information about LoadBalancer services, see Type LoadBalancer in the Kubernetes documentation.

For each Envoy service, Avi Controller creates a corresponding Virtual Service. See Virtual Services in the Avi Vantage documentation.

After Avi Controller creates a Virtual Service, the Controller configures the Avi Service Engines to forward traffic to the Envoy pods. The Envoy pods route traffic based on incoming requests, including traffic splitting and path based routing.

The Avi Controller provides Envoy with an external IP address so that apps are reachable by the app developer.

Note: Avi does not interact directly with any Cloud Native Runtimes resources. Avi Vantage forwards

all incoming traffic to Envoy.

Configuring Cloud Native Runtimes for Tanzu with Tanzu Service Mesh

This topic describes the workaround for using Tanzu Service Mesh with Cloud Native Runtimes for Tanzu. You cannot install Cloud Native Runtimes on a cluster that has Tanzu Service Mesh attached. If you want to install Cloud Native Runtimes on a cluster where Tanzu Service Mesh is attached, follow the procedure below.

This workaround describes how Tanzu Service Mesh can be configured to ignore the Cloud Native Runtimes namespaces. This allows Contour to provide ingress routing for the Knative workloads, while Tanzu Service Mesh continues to satisfy other connectivity concerns.

Note: Cloud Native Runtimes workloads are unable to use Tanzu Service Mesh features like Global Namespace, Mutual Transport Layer Security authentication (mTLS), retries, and timeouts.

For information about Tanzu Service Mesh, see Tanzu Service Mesh Documentation.

Run Cloud Native Runtimes on a Cluster Attached to Tanzu Service Mesh

This procedure assumes you have a cluster attached to Tanzu Service Mesh, and that you have not yet installed Cloud Native Runtimes.

Note: If you installed Cloud Native Runtimes on a cluster that has Tanzu Service Mesh attached before doing the procedure below, pods fail to start. To fix this problem, follow the procedure below and then delete all pods in the excluded namespaces.

Configure Tanzu Service Mesh to ignore namespaces related to Cloud Native Runtimes:

1. Navigate to the **Cluster Overview** tab in the Tanzu Service Mesh UI.
2. On the cluster where you want to install Cloud Native Runtimes, click **...**, then select **Edit Cluster...**.
3. Create an Is Exactly rule for each of the following namespaces:
    - contour-external
    - contour-internal
    - knative-serving
    - knative-eventing
    - knative-sources
    - knative-discovery
    - triggermesh

- vmware-sources
- cloud-native-runtimes
- rabbitmq-system
- kapp-controller
- The namespace or namespaces where you plan to run Knative workloads.

Next Steps

After configuring Tanzu Service Mesh, install Cloud Native Runtimes and verify your installation:

1. Install Cloud Native Runtimes. See Installing Cloud Native Runtimes.
2. Verify your installation. See Verifying Your Installation.

Note: You must create all Knative workloads in the namespace or namespaces where you plan to run these Knative workloads. If you do not, your pods fail to start.

Troubleshooting Cloud Native Runtimes for Tanzu

This topic describes troubleshooting information for problems during Cloud Native Runtimes for Tanzu installation or configuration.

Cannot connect to app on AWS

Symptom

On AWS, you see the following error when connecting to your app:

```
curl: (6) Could not resolve host: a**********************7.us-west-2.elb.amazonaws.co
m
```

Solution

Try connecting to your app again after 5 minutes. The AWS LoadBalancer name resolution takes several minutes to propagate.

minikube Pods Fail to Start

Symptom

On minikube, you see the following error when installing Cloud Native Runtimes:

```
3:03:59PM: error: reconcile job/contour-certgen-v1.10.0 (batch/v1) namespace: contour-
internal
Pod watching error: Creating Pod watcher: Get "https://192.168.64.17:8443/api/v1/pods?
labelSelector=kapp.k14s.io%2Fapp%3D1618232545704878000&watch=true": dial tcp 192.168.6
4.17:8443: connect: connection refused
kapp: Error: waiting on reconcile job/contour-certgen-v1.10.0 (batch/v1) namespace: co
ntour-internal:
  Errored:
   Listing schema.GroupVersionResource{Group:"", Version:"v1", Resource:"pods"}, names
paced: true:
    Get "https://192.168.64.17:8443/api/v1/pods?labelSelector=kapp.k14s.io%2Fassociati
on%3Dv1.572a543d96e0723f858367fcf8c6af4e": unexpected EOF
```

Solution

Increase your available system RAM to at least 4 GB.

Knative Services never become ready when using AutoTLS

Symptom

In the `config-network` ConfigMap, `httpProtocol` is set to either `Redirected` or `Disabled`, like the example below:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-network
  namespace: knative-serving
data:
  ...
  autoTLS: Enabled
  httpProtocol: Redirected # OR Disabled
  ...
```

When you create a Knative service, the service does not become ready.

```
$ kubectl get ksvc helloworld
NAME            URL                                        LATESTCREATED           LATESTRE
ADY             READY     REASON
helloworld   https://helloworld.default.example.com   helloworld-jfnzt-1   helloworld-
jfnzt-1   Unknown   EndpointsNotReady
```

Solution

This is a known issue in `1.0.x` versions of Cloud Native Runtimes. Set `httpProtocol` to `Enabled` so that AutoTLS to works properly, like the example below:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-network
  namespace: knative-serving
data:
  ...
  autoTLS: Enabled
  httpProtocol: Enabled
  ...
```

Installation fails with kapp-controller v0.16

Symptom

When installing Cloud Native Runtimes, you see the following error:

```
kapp: Error: waiting on reconcile app/cloud-native-runtimes (kappctrl.k14s.io/v1alpha1
) namespace: cloud-native-runtimes:
  Finished unsuccessfully (Reconcile failed:  (message: Fetching (0): Unsupported way
to fetch templates))
```

Solution

Install kapp-controller v0.17.0 or later on your cluster. Cloud Native Runtimes requires kapp-controller support for `imgpkgBundle fetcher`, which was introduced in kapp-controller v0.17.0.

On some Kubernetes versions and cloud providers, Tanzu Kubernetes Grid v1.3.1 installs kapp-controller v0.16.0, which is incompatible with Cloud Native Runtimes. For more information about Tanzu Kubernetes Grid kapp-controller versions, see the TKG Release Notes.

Installation fails to reconcile app/cloud-native-runtimes

Symptom

When installing Cloud Native Runtimes, you see one of the following errors:

```
11:41:16AM: ongoing: reconcile app/cloud-native-runtimes (kappctrl.k14s.io/v1alpha1) n
amespace: cloud-native-runtime
11:41:16AM:  ^ Waiting for generation 1 to be observed
kapp: Error: Timed out waiting after 15m0s
```

Or,

```
3:15:34PM:  ^ Reconciling
3:16:09PM: fail: reconcile app/cloud-native-runtimes (kappctrl.k14s.io/v1alpha1) names
pace: cloud-native-runtimes
3:16:09PM:  ^ Reconcile failed:  (message: Deploying: Error (see .status.usefulErrorMe
ssage for details))

kapp: Error: waiting on reconcile app/cloud-native-runtimes (kappctrl.k14s.io/v1alpha1
) namespace: cloud-native-runtimes:
  Finished unsuccessfully (Reconcile failed:  (message: Deploying: Error (see .status.
usefulErrorMessage for details)))
```

Explanation

The `cloud-native-runtimes` deployment app installs the subcomponents of Cloud Native Runtimes. Error messages about reconciling indicate that one or more subcomponents have failed to install.

Solution

Use the following procedure to examine logs:

1. Get the logs from the `cloud-native-runtimes` app by running:

   ```
   kubectl get app/cloud-native-runtimes -n cloud-native-runtimes -o jsonpath="{.s
   tatus.deploy.stdout}"
   ```

   Note: If the command does not return log messages, then kapp-controller is not installed or is not running correctly.

2. Review the output for subcomponent deployments that have failed or are still ongoing. See the examples below for suggestions on resolving common problems.

Example 1: The Cloud Provider does not support the creation of Service type

LoadBalancer

Follow these steps to identify and resolve the problem of the cloud provider not supporting services of type `LoadBalancer`:

1. Search the log output for `Load balancer`, for example by running:

   ```
   kubectl -n cloud-native-runtimes get app cloud-native-runtimes -ojsonpath="{.st
   atus.deploy.stdout}" | grep "Load balancer" -C 1
   ```

2. If the output looks similar to the following, ensure that your cloud provider supports services of type `LoadBalancer`. For more information, see Prerequisites.

   ```
   6:30:22PM: ongoing: reconcile service/envoy (v1) namespace: contour-external
   6:30:22PM:  ^ Load balancer ingress is empty
   6:30:29PM: ---- waiting on 1 changes [322/323 done] ----
   ```

Example 2: The webhook deployment failed

Follow these steps to identify and resolve the problem of the `webhook` deployment failing in the `vmware-sources` namespace:

1. Review the logs for output similar to the following:

   ```
   10:51:58PM: ok: reconcile customresourcedefinition/httpproxies.projectcontour.i
   o (apiextensions.k8s.io/v1) cluster
   10:51:58PM: fail: reconcile deployment/webhook (apps/v1) namespace: vmware-sour
   ces
   10:51:58PM:  ^ Deployment is not progressing: ProgressDeadlineExceeded (message
   : ReplicaSet "webhook-6f5d979b7d" has timed out progressing.)
   ```

2. Run `kubectl get pods` to find the name of the pod:

   ```
   kubectl get pods --show-labels -n NAMESPACE
   ```

   Where `NAMESPACE` is the namespace associated with the reconcile error, for example, `vmware-sources`.

   For example,

   ```
   $ kubectl get pods --show-labels -n vmware-sources
   NAME                       READY   STATUS    RESTARTS   AGE   LABELS
   webhook-6f5d979b7d-cxr9k   0/1     Pending   0          44h   app=webhook,kapp.
   k14s.io/app=1626302357703846007,kapp.k14s.io/association=v1.9621e0a793b4e925077
   dd557acedbcfe,pod-template-hash=6f5d979b7d,role=webhook,sources.tanzu.vmware.co
   m/release=v0.23.0
   ```

3. Run `kubectl logs` and `kubectl describe`:

   ```
   kubectl logs PODNAME -n NAMESPACE
   kubectl describe pod PODNAME -n NAMESPACE
   ```

   Where:

- `PODNAME` is found in the output of step 3, for example `webhook-6f5d979b7d-cxr9k`.
- `NAMESPACE` is the namespace associated with the reconcile error, for example, `vmware-sources`.

For example:

```
$ kubectl logs webhook-6f5d979b7d-cxr9k -n vmware-sources

$ kubectl describe pod webhook-6f5d979b7d-cxr9k  -n vmware-sources
Events:
Type     Reason            Age                 From               Message
----     ------            ----                ----               -------
Warning  FailedScheduling  80s (x14 over 14m)  default-scheduler  0/1 nodes are
 available: 1 Insufficient cpu.
```

4. Review the output from the `kubectl logs` and `kubectl describe` commands and take further action.

   For this example of the webhook deployment, the output indicates that the scheduler does not have enough CPU to run the pod. In this case, the solution is to add nodes or CPU cores to the cluster. If you are using Tanzu Mission Control (TMC), increase the number of workers in the node pool to three or more through the TMC UI. See Edit a Node Pool, in the TMC documentation.

Cloud Native Runtimes Installation Fails with Existing Contour Installation

Symptom

You see the following error message when you run the install script:

```
Could not proceed with installation. Refer to Cloud Native Runtimes documentation for
details on how to utilize an existing Contour installation. Another app owns the custo
m resource definitions listed below.
```

Solution

Follow the procedure in Install Cloud Native Runtimes on a Cluster with Your Existing Contour Instances to resolve the issue.

Upgrading Cloud Native Runtimes for Tanzu

This topic describes how to upgrade Cloud Native Runtimes for Tanzu to the latest version.

There is no formal upgrade path from the beta version of Cloud Native Runtimes to a general availability (GA) release. To upgrade from beta to Cloud Native Runtimes v1.x, you need to uninstall the old release and install the latest, as outlined below.

Upgrade from Beta to GA

To upgrade Cloud Native Runtimes from v0.2 to v1.0 or later:

1. Uninstall. See Uninstalling Cloud Native Runtimes.
2. Install. See Installing Cloud Native Runtimes.

Uninstalling Cloud Native Runtimes for Tanzu

This topic describes how to uninstall Cloud Native Runtimes for Tanzu.

To uninstall Cloud Native Runtimes

1. Run:

```
kapp delete -a cloud-native-runtimes -n cloud-native-runtimes
kubectl delete ns cloud-native-runtimes
```