

Cloud Native Runtimes for VMware Tanzu 2.3

Cloud Native Runtimes for VMware Tanzu 2.3

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

<https://docs.vmware.com/>

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

Copyright © 2024 Broadcom. All Rights Reserved. The term “Broadcom” refers to Broadcom Inc. and/or its subsidiaries. For more information, go to <https://www.broadcom.com>. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies. [Copyright and trademark information](#).

Contents

Cloud Native Runtimes Overview	8
Overview	8
Warnings	9
Cloud Native Runtimes Release Notes	10
v2.3.6	10
Change Log	10
Components	10
v2.3.4	10
Components	10
v2.3.2	11
Resolved Issues	11
Components	11
v2.3.1	11
Breaking Changes	11
New Features	11
Resolved Issues	12
Known Issues	12
Components	12
Integrations you can use with Cloud Native Runtimes	13
Cloud Native Runtimes integrations	13
Install Cloud Native Runtimes	14
Prerequisites	14
Install	14
Administrator Guide for Cloud Native Runtimes	18
Configure your External DNS with Cloud Native Runtimes	18
Overview	18
Configure custom domain	18
Configure Knative Service Domain Template	19
Securing Your Web Workloads in Cloud Native Runtimes	20
Prerequisites	20

Overview of Cloud Native Runtimes TLS Configurations	20
Default TLS configuration in Cloud Native Runtimes	20
Custom TLS configuration in Cloud Native Runtimes	21
Replace the shared ingress issuer at the Tanzu Application Platform's level	21
Designate another ingress issuer for your workloads in Cloud Native Runtimes only	21
Provide an existing TLS certificate for your workloads in Cloud Native Runtimes	22
Resources on custom TLS configuration for Cloud Native Runtimes:	22
Configure Cloud Native Runtimes to use a custom Issuer or ClusterIssuer	22
Configure a custom issuer	22
Configure Cloud Native Runtimes to use the custom issuer	23
Verify the issuance of certificates	23
Use wildcard certificates with Cloud Native Runtimes	24
Configure an issuer for wildcard certificates	24
Configure Cloud Native Runtimes to use wildcard certificates	25
Verify the issuance of wildcard certificates	26
Use your existing TLS Certificate for Cloud Native Runtimes	26
Overview	26
Prerequisites	27
Deactivate HTTP-to-HTTPS redirection	28
Opt out from an ingress issuer and deactivate automatic TLS feature	28
Deactivate TLS	29
Installing Cloud Native Runtimes with your Existing Contour Installation	29
About Using Contour with Cloud Native Runtimes	29
Prerequisites	30
Identify Your Contour Version	30
Install Cloud Native Runtimes on a Cluster with Your Existing Contour Instances	30
Configuring Eventing with RabbitMQ for Cloud Native Runtimes	31
Overview	31
Install VMware Tanzu RabbitMQ for Kubernetes	31
Next Steps	31
Configuring Observability for Cloud Native Runtimes	32
Overview	32


Logging	32
Configure Logging with Fluent Bit	32
Forward Logs to vRealize	33
Metrics	33
Tracing	34
Configuring Tracing	34
Forwarding Trace Data to an Observability Platform or Data Visualization Tool	34
Sending Trace Data to VMware Aria Operations for Applications	35
Configuring Cloud Native Runtimes with Avi Vantage	37
Overview	37
Integrate Avi Vantage with Cloud Native Runtimes	37
About Routing with Avi Vantage and Cloud Native Runtimes	38
Configuring Cloud Native Runtimes with Tanzu Service Mesh	39
Overview	39
Run Cloud Native Runtimes on a Cluster Attached to Tanzu Service Mesh	39
Next Steps	40
Customizing Cloud Native Runtimes	40
Customizing Cloud Native Runtimes	40
Troubleshooting Cloud Native Runtimes	41
Updates fail with error annotation value is immutable	41
Symptom	41
Explanation	42
Solution	42
Cannot connect to app on AWS	43
Symptom	43
Solution	44
minikube Pods Fail to Start	44
Symptom	44
Solution	44
Pulling an image with imgpkg overwrites the cloud-native-runtimes directory	44
Symptom	44
Solution	44
Installation fails to reconcile app/cloud-native-runtimes	44
Symptom	44
Explanation	45
Solution	45

Example 1: The Cloud Provider does not support the creation of Service type LoadBalancer	45
Example 2: The webhook deployment failed	46
Cloud Native Runtimes Installation Fails with Existing Contour Installation	47
Symptom	47
Solution	47
Knative Service Fails to Come up Due to Invalid HTTPProxy	47
Symptom	47
Solution	48
When using auto-tls, Knative Service Fails with CertificateNotReady.	48
Symptom	48
Explanation	48
Solution	49
Option 1: Change the domain_template	49
Option 2: Shorten the names of Knative Services or Namespaces	49
Verifying Your Installation	50
Prerequisites	50
Verify Installation of Knative Serving, Knative Eventing, and TriggerMesh SAWS	51
Verifying Knative Serving for Cloud Native Runtimes	51
About Verifying Knative Serving	51
Prerequisites	52
Test Knative Serving	52
Delete the Example Knative Service	54
Verify Knative Eventing with Cloud Native Runtimes	54
About Verifying Knative Eventing	54
Prerequisites	54
Prepare the RabbitMQ Environment	55
Verify Knative Eventing	56
Setup RabbitMQ Broker as the default in the cluster (optional)	58
Setup RabbitMQ Broker as the default in a namespace (optional)	59
Delete the Eventing Resources	61
Verifying TriggerMesh SAWS for Cloud Native Runtimes	62
Overview	62
Prerequisites	62
Verify TriggerMesh SAWS	62
Upgrading Cloud Native Runtimes	65

Prerequisites	65
Upgrade Cloud Native Runtimes	65
Uninstalling Cloud Native Runtimes	66
Overview	66
Uninstall	66

Cloud Native Runtimes Overview

This topic gives you an overview of Cloud Native Runtimes, commonly known as CNRs.



Note

Starting with the v2.4 release, Cloud Native Runtimes has moved to the [Tanzu Application Platform v1.7 and later documentation](#).

Overview

Cloud Native Runtimes is enterprise supported Knative, with the Carvel tools suite for deployment and Contour for networking. Cloud Native Runtimes offers everything Knative does and some extras that make it ideal for cloud native application development. Cloud Native Runtimes gives developers environmental simplicity and administrators deployment control and it works on any single Kubernetes cluster running Kubernetes v1.25 and later.

Cloud Native Runtimes utilizes Knative’s main features of Serving and Eventing to provide:

- Automatic pod scaling.
- Traffic splitting by code release version.
- Event-triggered workloads.

Cloud Native Runtimes simplifies the Developer experience.

Kubernetes Developers need to know:	Cloud Native Runtimes Developers need to know:
Pods	Pods
Deployment & Rollout Progress	Knative Service
Service (networking model)	
Ingress	
Labels and selectors	

Cloud Native Runtimes increases Administrator control and support.

Administrators can:
Manage infrastructure costs with request driven autoscaling
Test deployments with traffic splitting by code version
Use Carvel command tools to simplify deployment

Administrators can:

Receive Enterprise Support when they need it

Cloud Native Runtimes works well with these use cases:

- Batch Jobs Processing
- AI/ML
- Application or Network Monitoring
- IOT
- Event driven and serverless application architectures

For more information on the software that makes Cloud Native Runtimes see:


- Knative Documentation [Home - Knative](#)
- Carvel Tools Suite Documentation [Carvel - Home](#)
- Contour Networking Documentation [Contour](#)

Warnings

Eventing in Tanzu Application Platform is deprecated and marked for removal in Tanzu Application Platform v1.7.0.

Cloud Native Runtimes Release Notes

This topic contains release notes for Cloud Native Runtimes for Tanzu v2.3, commonly known as CNRs.



Note

Starting with the v2.4 release, Cloud Native Runtimes has moved to the [Tanzu Application Platform v1.7 and later documentation](#).

v2.3.6

Release Date: December 12, 2023

Change Log

- Rebuilt binaries using GoLang 1.20.11

Components

Cloud Native Runtimes v2.3.6 uses the following component versions:

	Release	Details
Version		2.3.6
Release date		December 12, 2023
	Component	Version
	Knative Serving	1.10.6
	Knative cert-manager Integration	1.10.4
	Knative Serving Contour Integration	1.10.3

v2.3.4

Release Date: November 14, 2023

Components

Cloud Native Runtimes v2.3.4 uses the following component versions:

	Release	Details
--	---------	---------

Version	2.3.4
Release date	November 14, 2023
Component	Version
Knative Serving	1.10.6
Knative cert-manager Integration	1.10.4
Knative Serving Contour Integration	1.10.3

v2.3.2

Release Date: October 10, 2023

Resolved Issues

This release has the following fixes:

- When running on a cluster v1.25 or higher that enforces Pod Security Standards, Knative components now include the appropriate SecurityContext settings to comply with the `restricted` level.

Components

Cloud Native Runtimes v2.3.2 uses the following component versions:

Release	Details
Version	2.3.2
Release date	October 10, 2023
Component	Version
Knative Serving	1.10.2
Knative cert-manager Integration	1.10.1
Knative Serving Contour Integration	1.10.0

v2.3.1

Release Date: 27 Jul 2023

Breaking Changes

This release has the following breaking changes:

- provider config option:** The deprecation of the `provider` configuration option is announced in the [release notes of Cloud Native Runtimes 2.0](#). As part of this release, the option is removed completely.

New Features

This release has the following new features:

- **New `default_external_scheme` config option:**
 - Configures `default-external-scheme` on Knative's `config-network` ConfigMap with default scheme to use for Knative Service URLs. Supported values are either `http` or `https`. You cannot set with the `default_tls_secret` option.

Resolved Issues

This release has the following fixes:

- New toggle feature for how to make ConfigMap updates

For some ConfigMaps in Cloud Native Runtimes, such as `config-features`, the option to update using an overlay was not taking effect. This issue is fixed. With this version, the legacy behavior remains the same, but VMware introduces a configuration to opt-in into updating ConfigMaps using overlays in Cloud Native Runtimes, as it is for all Tanzu Application Platform components. To configure this option, edit your `cnr-values.yaml` file to change the following configuration:

```
allow_manual_configmap_update: false.
```

In a future planned release of Cloud Native Runtimes, `false` will be the default configuration.

VMware plans to release Cloud Native Runtimes without the option to switch and `false` will be the permanent behavior.

Known Issues

This release has the following known issues:

- Knative Serving: Certain app name, namespace, and domain combinations produce Knative Services with status `CertificateNotReady`. See [Troubleshooting](#).

Components

Cloud Native Runtimes v2.3.1 uses the following component versions:

	Release	Details
Version		2.3.1
Release date		27 Jul 2023
	Component	Version
	Knative Serving	1.10.2
	Knative cert-manager Integration	1.10.1
	Knative Serving Contour Integration	1.10.0

Integrations you can use with Cloud Native Runtimes

This topic tells you the supported integrations for Cloud Native Runtimes. For more information regarding these integrations, see the [Cloud Native Runtimes Administrator Guide](#).

Cloud Native Runtimes integrations

Cloud Native Runtimes integration	Version	Documentation
VMware Tanzu Observability	Supported	Configuring Observability for Cloud Native Runtimes
Avi Vantage	Supported	Configuring Cloud Native Runtimes with Avi Vantage
Rabbit MQ	Supported	Configuring Cloud Native Runtimes with RabbitMQ
Tanzu Service Mesh	Supported	Configuring Cloud Native Runtimes with Tanzu Service Mesh

For tools and software compatibility, please refer to [Tanzu Application Platform's requirements](#).

Install Cloud Native Runtimes

This document describes how you can install Cloud Native Runtimes, commonly known as CNRs, from the Tanzu Application Platform package repository.

Note: Use the instructions on this page if you do not want to use a profile to install packages. Both the full and light profiles include Cloud Native Runtimes. For more information about profiles, see [Installing the Tanzu Application Platform Package and Profiles](#).

Prerequisites

Before installing Cloud Native Runtimes:

- Complete all prerequisites to install Tanzu Application Platform. For more information, see [Prerequisites](#).
- Contour is installed in the cluster. Contour can be installed from the [Tanzu Application package repository](#). If you have an existing Contour installation, see [Installing Cloud Native Runtimes with an Existing Contour Installation](#).
- By default, Tanzu Application Platform installs and uses a self-signed certificate authority for issuing TLS certificates to components by using ingress issuer. For more information, see [Ingress Certificates](#). To successfully install Cloud Native Runtimes, `shared.ingress_domain` or `cnrs.domain_name` property is required to be set when `ingress_issuer` property is set. For example:

```
shared:
  ingress_domain: "foo.bar.com"
```

or

```
cnrs:
  domain_name: "foo.bar.com"
```

If the domain name is not available or desired, domain name can be set to any valid value as long as no process is relying on the domain name resolving to the envoy IP. (Not recommended for production environments) Another alternative to bypass setting domain name is to disable auto-TLS. For more information, see [Disabling Automatic TLS Certificate Provisioning](#).

Install

To install Cloud Native Runtimes:

1. List version information for the package by running:

```
tanzu package available list cnrs.tanzu.vmware.com --namespace tap-install
```

For example:

```
tanzu package available list cnrs.tanzu.vmware.com --namespace tap-install

NAME                                VERSION  RELEASED-AT
cnrs.tanzu.vmware.com 2.3.1    2023-06-05 19:00:00 -0500 -05
```

2. (Optional) Make changes to the default installation settings:

1. Gather values schema.

```
tanzu package available get cnrs.tanzu.vmware.com/2.3.1 --values-schema -n tap-install
```

For example:

```
tanzu package available get cnrs.tanzu.vmware.com/2.3.1 --values-schema -n tap-install

KEY                                DEFAULT                                TYPE
DESCRIPTION

domain_config                      <nil>                                  <nil>
> Optional. Overrides the Knative Serving "config-domain" ConfigMap, allowing you to map Knative Services to specific domains. Must be valid YAML and conform to the "config-domain" specification.
namespace_selector                 string
> Specifies a LabelSelector which determines which namespaces should have a wildcard certificate provisioned. Set this property only if the Cluster issuer is type DNS-01 challenge.
pdb.enable                         true                                    <nil>
> Optional. Set to true to enable a PodDisruptionBudget for the Knative Serving activator and webhook deployments.
domain_name                        string
> Optional. Default domain name for Knative Services.
ingress.external.namespace         tanzu-system-ingress                 string
> Required. Specify a namespace where an existing Contour is installed on your cluster. CNRs will use this Contour instance for external services.
ingress.internal.namespace        tanzu-system-ingress                 string
> Required. Specify a namespace where an existing Contour is installed on your cluster. CNRs will use this Contour instance for internal services.
lite.enable                       false                                  <nil>
> Optional. Set to "true" to enable lite mode. Reduces CPU and Memory resource requests for all cnrs Deployments, Daemonsets, and StatefulSets by half. Not recommended for production.
domain_template                    {{.Name}}.{{.Namespace}}.{{.Domain}} string
> Optional. Specifies the goolang text template string to use when constructing the DNS name for a Knative Service.
kubernetes_distribution           <nil>                                  <nil>
> Optional. Type of K8s infrastructure being used. Supported Values: openshift
kubernetes_version                0.0.0                                  <nil>
> Optional. Version of K8s infrastructure being used. Supported Values
```

```

: valid Kubernetes major.minor.patch versions
allow_manual_configmap_update true bool
ean Specifies how updates to some CNRs ConfigMaps can be made. Set to Tr
ue, CNRs allows updates to those ConfigMaps to be made only manually. Set
to False, updates to those CNRs ConfigMaps can be made only using overla
ys. Supported Values: True, False.
ca_cert_data string
Optional. PEM Encoded certificate data to trust TLS connections with
a private CA.
default_external_scheme <nil> string
Optional. Specifies the default scheme to use for Knative Service UR
Ls, regardless of other TLS configurations. Supports either http or https
. Cannot be set along with default_tls_secret
default_tls_secret string
Optional. Specify a fallback TLS Certificate for use by Knative Serv
ices if autoTLS is disabled. Will set default external scheme for Knativ
e Service URLs to "https". Requires either "domain_name" or "domain_confi
g" to be set and cannot be set along with "default_external_scheme".
https_redirection true bool
CNRs ingress will send a 301 redirect for all http connections, aski
ng the clients to use HTTPS
ingress_issuer string
Cluster issuer to be used in CNRs. To use this property the domain_n
ame or domain_config must be set. Under the hood, when this property is s
et auto-tls is Enabled.

```

2. Create a `cnr-values.yaml` file by using the following sample as a guide to configure Cloud Native Runtimes:

Note: For most installations, you can leave the `cnr-values.yaml` empty, and use the default values.

```

---
# Configures the domain that Knative Services will use
domain_name: "mydomain.com"

```

Configuration Notes: * If you are running on a single-node cluster, such as minikube, set the `lite.enable: true` option to lower CPU and memory requests for resources. In case you also want to deactivate pod disruption budgets on Knative Serving and high availability is not indispensable in your development environment, you can set `pbk.enable` to `false`.

- Cloud Native Runtimes reuses the existing `tanzu-system-ingress` Contour installation for external and internal access when installed in the `light` or `full` profile. If you want to use a separate Contour installation for system-internal traffic, set `cnrs.ingress.internal.namespace` to the namespace of your separate Contour installation.
- If you install Cloud Native Runtimes with the default value of `true` for the `allow_manual_configmap_update` configuration, you will only be able to update some ConfigMaps (Ex: config-features manually. If you would like to update all ConfigMaps using overlays, please change this value to `false`. In a future release, `false` will be the default configuration. At some point after that, Cloud Native Runtimes will be released without the option to switch and `false` will be the permanent behavior.

3. Install the package by running:

```
tanzu package install cloud-native-runtimes -p cnrs.tanzu.vmware.com -v 2.3.1 -
n tap-install --values-file cnr-values.yaml --poll-timeout 30m
```

For example:

```
tanzu package install cloud-native-runtimes -p cnrs.tanzu.vmware.com -v 2.3.1 -
n tap-install -values-file cnr-values.yaml --poll-timeout 30m

| Installing package 'cnrs.tanzu.vmware.com'
| Getting package metadata for 'cnrs.tanzu.vmware.com'
| Creating service account 'cloud-native-runtimes-tap-install-sa'
| Creating cluster admin role 'cloud-native-runtimes-tap-install-cluster-role'
| Creating cluster role binding 'cloud-native-runtimes-tap-install-cluster-role
binding'
- Creating package resource
- Package install status: Reconciling

Added installed package 'cloud-native-runtimes' in namespace 'tap-install'
```

4. Verify the package install by running:

```
tanzu package installed get cloud-native-runtimes -n tap-install
```

For example:

```
tanzu package installed get cloud-native-runtimes -n tap-install

Retrieving installation details for cloud-native-runtimes...
NAME:                cloud-native-runtimes
PACKAGE-NAME:        cnrs.tanzu.vmware.com
PACKAGE-VERSION:     2.3.1
STATUS:              Reconcile succeeded
CONDITIONS:          [{ReconcileSucceeded True  }]
USEFUL-ERROR-MESSAGE:
```

Verify that **STATUS** is **Reconcile succeeded**.

Administrator Guide for Cloud Native Runtimes

The next several pages show you how to use, troubleshoot, integrate, upgrade and uninstall Cloud Native Runtimes, commonly known as CNRs.

Configure your External DNS with Cloud Native Runtimes

This topic describes how you can configure your external DNS with Cloud Native Runtimes, commonly known as CNRs.

Overview

Knative uses `svc.cluster.local` as the default 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 up a custom domain and an external DNS.

Configure custom domain

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

1. Configure your custom domain:

When your workloads are created, Knative will automatically create URLs for each workload based on the configuration in the domain ConfigMap.

- To set a default custom domain, edit your `cnr-values.yml` file to contain the following:

```
---
domain_name: "mydomain.com"
```

This will modify the Knative domain ConfigMap to use `domain_name` as the default domain.

Note: `domain_name` must be a valid DNS subdomain.

- **Advanced:** To overwrite the domain ConfigMap entirely, edit your `cnr-values.yml` file to contain your desired config-domain options, similar to the following:

```
---
domain_config: |
  ---
  mydomain.com: |
```

```
mydomain.org: |
  selector:
    app: nonprofit
```

This will replace the body of the Knative domain ConfigMap with `domain_config`. This will allow you to configure multiple custom domains, and configure a custom domain for a service depending on its labels.

See [Changing the default domain](#) for more information about the structure of the domain ConfigMap.

Note: `domain_config` must be valid YAML and a valid domain ConfigMap.

Note: You can only use one of `domain_config` or `domain_name` at a time. You may not use both.

2. Get the address of the cluster load balancer:

```
kubectl get service envoy -n EXTERNAL-CONTOUR-NS --output 'jsonpath={.status.loadBalancer.ingress}'
```

Where `EXTERNAL-CONTOUR-NS` is the namespace where a Contour serving external traffic is installed. If Cloud Native Runtimes was installed as part of a Tanzu Application Profile, this value will likely be `tanzu-system-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
```

If you chose to configure multiple custom domains above, you will need to create a wildcard DNS record for each domain.

Configure Knative Service Domain Template

Knative uses domain template which specifies the goolang text template string to use when constructing the Knative service's DNS name. The default value is `{{.Name}}.{{.Namespace}}.{{.Domain}}`. Valid variables defined in the template include Name, Namespace, Domain, Labels,

and Annotations.

To configure domain template for the created Knative Services, edit your `cnr-values.yml` file to contain the following:

```
---
domain_template: "{{.Name}}-{{.Namespace}}.{{.Domain}}"
```

This will modify the Knative `domain-template` ConfigMap to use `domain_template` as the default domain template.

Changing this value might be necessary when the extra levels in the domain name generated are problematic for wildcard certificates that only support a single level of domain name added to the certificate's domain. In those cases you might consider using a value of `{{.Name}}-{{.Namespace}}.{{.Domain}}`, or removing the Namespace entirely from the template.

When choosing a new value, be thoughtful of the potential for conflicts, such as when users the use of characters like `-` in their service or namespace names.

`{{.Annotations}}` or `{{.Labels}}` can be used for any customization in the go template if needed.

It is strongly recommended to keep namespace part of the template to avoid domain name clashes: eg. `{{.Name}}-{{.Namespace}}.{{ index .Annotations "sub" }}.{{.Domain}}` and you have an annotation `{"sub": "foo"}`, then the generated template would be `{Name}-{Namespace}.foo.{Domain}`.

Securing Your Web Workloads in Cloud Native Runtimes

This topic give you an overview of securing HTTP connections using TLS certificates in Cloud Native Runtimes, commonly known as CNRs, for VMware Tanzu Application Platform and helps you configure TLS (Transport Layer Security).

Prerequisites

Ensure that you have the Tanzu Application Platform, Cloud Native Runtimes for VMware Tanzu, Contour, and cert-manager installed.

Overview of Cloud Native Runtimes TLS Configurations

This section describes default configuration, custom configuration, obtaining and renewing TLS certificates with Cloud Native Runtimes.

Default TLS configuration in Cloud Native Runtimes

When installing Tanzu Application Platform by using [profiles](#), the [cert-manager package](#) is utilized to facilitate the acquisition, management and renewal of TLS certificates.

Cloud Native Runtimes automatically acquire TLS certificates for workloads through the shared ingress issuer integrated into the Tanzu Application Platform. The ingress issuer is specified by the `shared.ingress_issuer` configuration value in Tanzu Application Platform, and it refers to a `cert-manager.io/v1/ClusterIssuer`.

By default, the ingress issuer is self-signed and has limitations. For more information about the shared ingress issuer, see the Tanzu Application Platform documentation below:

- [Ingress certificates](#)
- [Shared ingress issuer](#)

The following TLS features are included in Cloud Native Runtimes by default:

- **Auto-TLS**

Cloud Native Runtimes has the Auto-TLS feature enabled by default. It uses the cert-manager package to automate the process of certificate issuance and management. Auto-TLS takes care of requesting, renewing, and configuring TLS certificates for each domain that you configure in your Cloud Native Runtimes settings.

- **Automatic HTTPS Redirection**

By default, Cloud Native Runtimes automatically redirects HTTP traffic to HTTPS for secured services. This ensures that all communication with your applications is encrypted and providing a secure experience for your users.

- **One certificate per hostname**

Cloud Native Runtimes issues a unique certificate for each hostname associated with a Knative Service.

Custom TLS configuration in Cloud Native Runtimes

While the default ingress issuer is suitable for testing and evaluation purposes, VMware highly recommends replacing it with your own issuer for production environments.

There are a few ways to customize TLS configuration in Cloud Native Runtimes:

Replace the shared ingress issuer at the Tanzu Application Platform's level

You have the flexibility to replace Tanzu Application Platform's default ingress issuer with any other [certificate authority](#) that is [compliant with cert-manager ClusterIssuer](#). For more information on how to replace the default ingress issuer, see [Replacing the default ingress issuer](#) documentation.

Cloud Native Runtimes will utilize the issuer specified by `shared.ingress_issuer` configuration value to issue certificates for your workload automatically.

Designate another ingress issuer for your workloads in Cloud Native Runtimes only

You can have a shared ingress issuer at the Tanzu Application Platform's level and also designate another issuer to be used by Cloud Native Runtimes to issue TLS certificates for your workloads. This allows you to customize TLS settings for Cloud Native Runtimes while maintaining a global configuration for other components.

You can designate an ingress issuer for Cloud Native Runtimes by specifying `cnrs.ingress_issuer` configuration value. The ingress/TLS configuration for Cloud Native Runtimes takes precedence over the shared ingress issuer.

If you wish to designate another ingress issuer for your workloads, navigate to [Configure Cloud Native Runtimes to use a custom Issuer or ClusterIssuer](#) for details.

Provide an existing TLS certificate for your workloads in Cloud Native Runtimes

If you manually generated a TLS certificate and wish to provide it to Cloud Native Runtimes instead of using an ingress issuer, you can follow the instructions in [Use your existing TLS Certificate for Cloud Native Runtimes](#).

Resources on custom TLS configuration for Cloud Native Runtimes:

- [Configure Cloud Native Runtimes to use a custom Issuer or ClusterIssuer](#)
- [Use wildcard certificates with Cloud Native Runtimes](#)
- [Use your existing TLS Certificate for Cloud Native Runtimes](#)
- [Deactivate HTTP-to-HTTPS redirection](#)
- [Opt out from any ingress issuer and deactivate automatic TLS feature](#)

Configure Cloud Native Runtimes to use a custom Issuer or ClusterIssuer

The ability to opt out of the shared ingress issuer and use a custom Issuer or ClusterIssuer for Cloud Native Runtimes provides greater flexibility, security, isolation, and integration with existing infrastructure, allowing you to tailor the TLS configurations to your specific needs and requirements.

We will explain in the following example how to opt out of the shared ingress issuer and use Let's Encrypt with the HTTP01 challenge type. The HTTP01 challenge requires that your load balancer be reachable from the internet by using HTTP. With the HTTP01 challenge type, a certificate is provisioned for each service.

To configure Cloud Native Runtimes to use a custom Issuer or ClusterIssuer with the HTTP01 challenge, follow these steps:

Configure a custom issuer

You have the flexibility to replace Tanzu Application Platform's default ingress issuer with any other [certificate authority](#) that is [compliant with cert-manager ClusterIssuer](#). For more information on how to replace the default ingress issuer, see [Replacing the default ingress issuer](#) documentation.

1. Create a custom Issuer or ClusterIssuer with the Certificate Authority (CA) that you want and configurations. Here's an example YAML configuration for a custom ClusterIssuer using Let's Encrypt with the HTTP01 challenge:

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-http01-issuer
spec:
  acme:
    email: YOUR-EMAIL
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: letsencrypt-http01-issuer-account-key
    solvers:
```

```
- http01:
  ingress:
    class: contour
```

Where `YOUR-EMAIL` is your email address.

Make sure to specify the ingress class you are using in your Tanzu Application Platform cluster, which is `contour`.

2. Save the configuration above in a file called `issuer-letsencrypt-http01.yaml`.



Note

If you want to test this feature, you might want to set `spec.acme.server` to `https://acme-staging-v02.api.letsencrypt.org/directory`. This is the staging url, which generates self-signed certs. It is useful for testing without worrying about hitting quotas for your actual domain.

3. Apply the Issuer or ClusterIssuer configuration to your cluster:

```
kubectl apply -f issuer-letsencrypt-http01.yaml
```

Configure Cloud Native Runtimes to use the custom issuer

1. Configure Cloud Native Runtimes to use the custom Issuer or ClusterIssuer for issuing certificates by updating your `tap-values.yaml` file with the following snippet of yaml.

```
cnrs:
  ingress_issuer: "letsencrypt-http01-issuer"
```

2. Update Tanzu Application Platform

To update the Tanzu Application Platform installation with the changes to the values file, run:

```
tanzu package installed update tap -p tap.tanzu.vmware.com -v ${TAP_VERSION} --
values-file tap-values.yaml -n tap-install
```

Verify the issuance of certificates

Verify that your ClusterIssuer was created and properly issuing certificates:

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

You can confirm the status of the certificate by running the command below. You should see the certificate in a `Ready` state.

```
kubectl get certificate -n DEVELOPER-NAMESPACE
```

Additionally, you can access your workload using the domain you specified with `curl` or a web browser, and verify that it is using a TLS certificate issued by the custom Issuer or ClusterIssuer.

```
tanzu apps workload get WORKLOAD-NAME --namespace DEVELOPER-NAMESPACE
kubectl get ksvc WORKLOAD-NAME -n DEVELOPER-NAMESPACE -o jsonpath='{.status.url}'
```

For details on how to troubleshoot failures related to the certificate, visit [cert-manager's Troubleshooting guide](#).

Use wildcard certificates with Cloud Native Runtimes

This section describes how to configure, use and verify wildcard certificates with Cloud Native Runtimes.

Cloud Native Runtimes utilizes the cert-manager package by default to automate the process of obtaining, managing, and renewing TLS certificates for your services.

Cert-manager is a Kubernetes-native component that works with different [certificate authorities](#) (CAs) like Let's Encrypt and others, to manage certificates within your Kubernetes cluster. As part of Tanzu Application Platform predefined profile installation, cert-manager package is deployed to the cluster.

Configure an issuer for wildcard certificates

Any other [cert-manager.io/v1/ClusterIssuer](#) can replace Tanzu Application Platform's default ingress issuer. This topic uses Let's Encrypt as an example of how to set up a custom ingress issuer that issues wildcard certificates.

The Let's Encrypt documentation states that the DNS01 challenge is required to validate wildcard domains. This topic provides instructions for configuring Cloud Native Runtimes to use wildcard certificates with the DNS01 challenge only.

1. Create a cert-manager custom Issuer or ClusterIssuer for the DNS01 challenge.

You must create a custom Issuer or ClusterIssuer with the DNS01 solver configured for your specific DNS provider. Visit cert-manager's documentation on [Supported DNS01 providers](#) for instructions on configuring cert-manager for all the supported DNS providers.

The following example uses Let's Encrypt and Google Cloud DNS:

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-dns-wildcard
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    # This will register an issuer with LetsEncrypt.
    email: YOUR-EMAIL
    privateKeySecretRef:
      # Set privateKeySecretRef to any unused secret name.
      name: letsencrypt-dns-wildcard-account-key
    solvers:
    - dns01:
        cloudDNS:
          # Set this to your GCP project id
          project: PROJECT_ID
          # This is the secret used to access the service account
```



```
serviceAccountSecretRef:
  name: cloud-dns-key
  key: key.json
```

Where `YOUR-EMAIL` is the email associated with your DNS provider.

Where `PROJECT_ID` is the ID of the GCP project.

2. Save the configuration above in a file called `issuer-wildcard.yaml`.

When using `cert-manager` to obtain wildcard certificates, you typically must provide credentials, especially when using the DNS01 challenge. The DNS01 challenge requires `cert-manager` to create and delete DNS records for domain validation during the certificate issuance process. To perform these actions, `cert-manager` needs access to your DNS provider's API, which requires authentication using API keys, access tokens, or other credentials. See [Supported DNS01 providers](#) in the `cert-manager` documentation.



Note

If you want to test this feature, you might want to set `spec.acme.server` to `https://acme-staging-v02.api.letsencrypt.org/directory`. This is the staging url, which generates self-signed certs. It is useful for testing without worrying about hitting quotas for your actual domain.

3. Apply the file you saved in the previous section to your cluster:

```
kubectl apply -f issuer-wildcard.yaml`
```

Configure Cloud Native Runtimes to use wildcard certificates

To use wildcard certificates:

1. Configure Cloud Native Runtimes to use the custom Issuer or ClusterIssuer and indicate in which namespaces to create wildcard certificates



Note

If no value is passed to `cnrs.namespace_selector`, only per service certificates are generated instead of wildcard certificates.

You achieve this by add the snippet below to your `tap-values.yaml` file.

```
cnrs:
  ingress_issuer: "letsencrypt-dns-wildcard"
  namespace_selector:
    matchExpressions:
      - key: apps.tanzu.vmware.com/tap-ns
        operator: Exists
```

This configuration tells Cloud Native Runtimes which custom Issuer or ClusterIssuer is used for issuing the wildcard certificate. When a Knative Service is created or updated with this

configuration, Cloud Native Runtimes requests and uses the wildcard certificate from the specified custom Issuer or ClusterIssuer.

In Cloud Native Runtimes, the per-namespace certificate manager operates by using the namespace labels to verify which namespaces require a certificate to be generated. This example specifies that only namespaces labeled with `apps.tanzu.vmware.com/tap-ns` have corresponding wildcard certificates created for them.

If you are using the suggested namespace selector, label your developer namespace with `apps.tanzu.vmware.com/tap-ns`. You can do so by running the following command:

```
kubectl label namespace DEV-NAMESPACE "apps.tanzu.vmware.com/tap-ns"
```

To remove a label from a namespace, run the following command:

```
kubectl label namespace DEV-NAMESPACE "apps.tanzu.vmware.com/tap-ns" --overwrite
```

2. Update Tanzu Application Platform.

To update the Tanzu Application Platform installation with the changes to the values file, run:

```
tanzu package installed update tap -p tap.tanzu.vmware.com -v ${TAP_VERSION} --values-file tap-values.yaml -n tap-install
```

Verify the issuance of wildcard certificates

Verify that your ClusterIssuer was created and properly issuing certificates:

```
kubectl get clusterissuer letsencrypt-dns-wildcard
```

You can confirm the status of the certificate by running the command below. You should see the certificate in a `Ready` state.

```
kubectl get certificate -n DEVELOPER-NAMESPACE
```

Additionally, you can access your workload using the domain you specified with `curl` or a web browser, and verify that it is using a TLS certificate issued by the custom Issuer or ClusterIssuer.

```
tanzu apps workload get WORKLOAD-NAME --namespace DEVELOPER-NAMESPACE
kubectl get ksvc WORKLOAD-NAME -n DEVELOPER-NAMESPACE -o jsonpath='{.status.url}'
```

For details on how to troubleshoot failures related to the certificate, visit [cert-manager's Troubleshooting guide](#).

Use your existing TLS Certificate for Cloud Native Runtimes

This topic tells you how to use your existing TLS Certificate for Cloud Native Runtimes, commonly known as CNRs.

Overview

Configure secure HTTPS connections to enable your web workloads and routes to terminate external TLS connections using an existing certificate.



Note

It is important to note that you have the flexibility to provide your own TLS certificate to Cloud Native Runtimes *instead of* relying on the shared ingress issuer for your Knative workloads. To utilize the feature explained in this document, you must configure Cloud Native Runtimes to bypass the cert-manager certificate issuer. For instance, if you have set `cnrs.default_tls_secret` in your `tap-values.yaml` file, you should set the `cnrs.ingress_issuer` configuration to an empty value. For detailed instructions on how to opt out and deactivate the automatic TLS feature, please refer to the documentation: [Opt out from any ingress issuer and deactivate automatic TLS feature](#).

Prerequisites

In order to configure TLS for Cloud Native Runtimes, you must first configure a Service Domain. For more information, see [Configuring External DNS with Cloud Native Runtimes](#).

To configure your TLS certificate for the created Knative Services, follow the steps:

- Create a Kubernetes Secret to hold your TLS Certificate

```
kubectl create -n DEVELOPER-NAMESPACE secret tls SECRET_NAME \
  --key key.pem \
  --cert cert.pem
```

- Create a delegation. To do so, create a `tlscertdelegation.yaml` file with following contents

```
apiVersion: projectcontour.io/v1
kind: TLSCertificateDelegation
metadata:
  name: default-delegation
  namespace: DEVELOPER-NAMESPACE
spec:
  delegations:
    - secretName: SECRET_NAME
      targetNamespaces:
        - "DEVELOPER-NAMESPACE"
```

Where `SECRET_NAME` is the name of the Kubernetes secret you created in the step above.

- Apply the above yaml file by running below command:

```
kubectl apply -f tlscertdelegation.yaml
```

- Include the following configuration in your `tap-values.yaml` file under Cloud Native Runtimes section and redeploy:

```
cnrs:
```

```
default_tls_secret: "DEVELOPER-NAMESPACE/SECRET_NAME"
```

Where `SECRET_NAME` is the name of the Kubernetes secret you created in the previous step.

Where `DEVELOPER-NAMESPACE` is the name of the namespace where the secret was created in the previous step.

- Update Tanzu Application Platform

To update the Tanzu Application Platform installation with the changes to the values file, run:

```
tanzu package installed update tap -p tap.tanzu.vmware.com -v ${TAP_VERSION} --
values-file tap-values.yaml -n tap-install
```

This will modify the Knative `config-contour` ConfigMap to use `default_tls_secret` as the default TLS certificate.

Your web workloads' URLs will use the scheme `https` by default when this secret is provided.

Deactivate HTTP-to-HTTPS redirection

When you designate an ingress issuer for your workloads by setting either the `shared.ingress_issuer` or `cnrs.ingress_issuer` configuration value, in your `tap-values.yaml` file, the auto-TLS feature is enabled in Cloud Native Runtimes. When the auto-TLS is enabled, Cloud Native Runtimes will automatically redirect traffic from HTTP to HTTPS. However, there may be situations where you want to opt out this behavior and continue serving content over HTTP. If this applies to your case, you must turn off the HTTPS redirection feature.

To deactivate HTTP-to-HTTPS redirection in Cloud Native Runtimes, you must modify your configuration values file and follow the steps below:

1. Configure Cloud Native Runtimes to deactivate https redirection

```
cnrs:
  https_redirection: false
```

2. Update your Tanzu Application Platform installation

```
tanzu package installed update tap -p tap.tanzu.vmware.com -v ${TAP_VERSION} --
values-file tap-values.yaml -n tap-install
```

3. Verify that the HTTP-to-HTTPS redirection is deactivated by accessing your workload using the HTTP protocol. You should be able to access the service without being redirected to HTTPS. Use a web browser or a tool like `curl` to test the behavior:

```
curl -I http://your-workload.your-domain.com
```

The response should show an HTTP status code without redirection to HTTPS.

Opt out from an ingress issuer and deactivate automatic TLS feature

This topic tells you how to opt out from an ingress issuer and deactivate automatic TLS feature for Cloud Native Runtimes, commonly known as CNRs.

Deactivate TLS

You can deactivate automatic TLS certificate provisioning in Cloud Native Runtimes by setting the `ingress_issuer` property to an empty string as follows:

```
cnrs:
  ingress_issuer: ""
```

Make sure to update your Tanzu Application Platform installation accordingly after following the step mentioned above.

```
tanzu package installed update tap -p tap.tanzu.vmware.com -v ${TAP_VERSION} --values-file tap-values.yaml -n tap-install
```

Installing Cloud Native Runtimes with your Existing Contour Installation

This topic describes how you can configure Cloud Native Runtimes, commonly known as CNRs, with your existing Contour instance. Cloud Native Runtimes uses Contour to manage internal and external access to the services in a cluster.

About Using Contour with Cloud Native Runtimes

The instructions on this page assume that you have an existing Contour installation on your cluster.

Follow the instructions on this page if:

- You have installed Contour as part of Tanzu Application Platform and wish to configure Cloud Native Runtimes to use it.
- You see an error about [an existing Contour installation](#) when you install the Cloud Native Runtimes package.

Cloud Native Runtimes needs two instances of Contour: one instance for exposing services outside the cluster, and another instance for services that are private in your network. If installed as part of a Tanzu Application Platform profile, by default Cloud Native Runtimes will use the Contour instance installed in the namespace `tanzu-system-ingress` for both internal and external traffic.

If you already use a Contour instance to route requests from clients outside and inside the cluster, you may use your own Contour if it matches the Contour version used by [Tanzu Application's Platform](#).

You may use the same single instance of Contour for both internal and external traffic. However, this will cause internal and external traffic will be handled the same way. For example, if the Contour instance is configured to be accessible from clients outside the cluster, then any internal traffic will also be accessible from outside the cluster. Note that currently Tanzu Application Platform only supports using a single Contour instance for both internal and external traffic.

In all of the above cases, Cloud Native Runtimes will use the Tanzu Application Platform's Contour

`CustomResourceDefinitionS`.

Prerequisites

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

- Contour version v1.22.0 (Contour version that is installed as part of Tanzu Application Platform). To identify your cluster's Contour version, see [Identify Your Contour Version](#) below.
- Contour `CustomResourceDefinitionS` versions:

Resource Name	Version
<code>contourdeployments.projectcontour.io</code>	v1alpha1
<code>contourconfigurations.projectcontour.io</code>	v1alpha1
<code>extensionservices.projectcontour.io</code>	v1alpha1
<code>httpproxies.projectcontour.io</code>	v1
<code>tlscertificatedelegations.projectcontour.io</code>	v1

Identify Your Contour Version

To identify your cluster's Contour version, run:

```
export CONTOUR_NAMESPACE=CONTOUR-NAMESPACE
export CONTOUR_DEPLOYMENT=$(kubectl get deployment --namespace $CONTOUR_NAMESPACE --output name)
kubectl get $CONTOUR_DEPLOYMENT --namespace $CONTOUR_NAMESPACE --output jsonpath="{.spec.template.spec.containers[].image}"
kubectl get crds extensionservices.projectcontour.io --output jsonpath="{.status.storedVersions}"
kubectl get crds httpproxies.projectcontour.io --output jsonpath="{.status.storedVersions}"
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 can add values to your `cnr-values.yml` file so that Cloud Native Runtimes will use your Contour instance.

An example of a `cnr-values.yml` file where you wish Cloud Native Runtimes to use the Contour version in a different namespace would look like this:

```
---
ingress:
  external:
```

```
namespace: "tanzu-system-ingress"
internal:
  namespace: "tanzu-system-ingress"
```

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

Configuring Eventing with RabbitMQ for Cloud Native Runtimes

This topic tells you how to use 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 for Cloud Native Runtimes, commonly known as CNRs.

Overview

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.

Install VMware Tanzu RabbitMQ for Kubernetes

Before you can use or test RabbitMQ eventing on Cloud Native Runtimes, you need to install VMWare Tanzu RabbitMQ for Kubernetes . Follow below steps to complete the installation:

1. [Accept End User License Agreement](#)
2. [Prerequisites:](#) This step installs kapp-controller and secretgen-controller. Skip this step if kapp-controller and secretgen-controller are already installed on your cluster.
3. Prepare for the Installation
 - ◆ [Provide imagePullSecrets](#)
 - ◆ [Install the PackageRepository](#)
 - ◆ [Create a Service Account](#)
 - ◆ [Install Cert-Manager:](#) Skip this step if cert-manager is already installed on your cluster.
4. [Install the Tanzu RabbitMQ Package:](#) This step will install the Tanzu RabbitMQ Cluster Operator, Message Topology Operator, and Standby Replication Operator on your cluster.

Next Steps

After completing these installations, you can:

- Verify your Knative Eventing installation using an example RabbitMQ broker. For instructions,

see [Verify Knative Eventing](#).

- [Bring your own RabbitMQ Cluster](#) you can plug in an existing RabbitMQ instance and start using it with the Tanzu and Knative Eventing resources.
- Create a broker, producer, and a consumer to use RabbitMQ and Knative Eventing with your own app.

Configuring Observability for Cloud Native Runtimes

This topic tells you how to configure observability for Cloud Native Runtimes, commonly known as CNRs.

Overview

You can set up integrations with third-party observability tools to use logging, metrics, and tracing with Cloud Native Runtimes. 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](#).

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](#). For information about viewing your Kubernetes namespaces, see [Viewing Namespaces](#). 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 VMware Aria Operations for Applications (formerly known as 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 about configuring this, see the [Prometheus 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 applications on Cloud Native Runtimes. To do this, you configure tracing for both Knative Serving and Eventing by editing the ConfigMap `config-tracing` for your Knative namespaces.

VMware recommends that you add any integrations 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 an Observability Platform or Data Visualization Tool

You can use the OpenTelemetry integration to forward trace data to a data visualization tool that can ingest data in Zipkin format. For more information about using Zipkin for tracing, see the [Zipkin](#)

documentation.

VMWare recommends integration with VMware Aria Operations for Applications (formerly known as Tanzu Observability by Wavefront). For information about forwarding trace data, see [Send OpenTelemetry Data to Tanzu Observability](#).

Sending Trace Data to VMware Aria Operations for Applications

You can send trace data to an observability and analytics platform such as VMware Aria Operations for Applications to view and monitor your trace data in dashboards. VMware Aria Operations for Applications offers several deployment options. During development, a single proxy is often sufficient for all data sources. See [Proxy Deployment Options](#) for more information on other deployment options.

Follow the steps below to configure Cloud Native Runtimes to send traces to the Wavefront proxy and then, configure the Wavefront proxy to consume Zipkin spans.

1. Deploy the Wavefront Proxy. For more information about Wavefront proxies, see [Install and Manage Wavefront Proxies](#).
2. Configure the namespace where the Wavefront Proxy was deployed with proper credentials to its image registry.

The example below utilizes the Namespace Provisioner package to automatically configure namespaces labeled with: `apps.tanzu.vmware.com/tap-ns`.

```
export WF_NAMESPACE=default
kubectl label namespace ${WF_NAMESPACE} apps.tanzu.vmware.com/tap-ns=""

export WF_REGISTRY_HOSTNAME=projects.registry.vmware.com
export WF_REGISTRY_USERNAME=<your-password>
export WF_REGISTRY_PASSWORD=<your-username>
tanzu secret registry add registry-credentials \
  --username ${WF_REGISTRY_USERNAME} --password ${WF_REGISTRY_PASSWORD} \
  --server ${WF_REGISTRY_HOSTNAME} \
  --export-to-all-namespaces --yes --namespace tap-install
```

Where: `WF_NAMESPACE` is the namespace where you deployed the Wavefront Proxy. `WF_REGISTRY_HOSTNAME` is the image registry where the Wavefront Proxy image is located. `WF_REGISTRY_USERNAME` is your username to access the image registry to pull the Wavefront Proxy image. `WF_REGISTRY_PASSWORD` is your password to access the image registry to pull the Wavefront Proxy image.

For more information on how to set up developer namespaces, see [Provision developer Namespace](#).

3. Configure the Wavefront Proxy to allow Zipkin/Istio traces.

You can uncomment the lines indicated in the yaml file for the Wavefront Deployment to enable consumption of Zipkin traces. You should edit the Wavefront Deployment to set the `WAVEFRONT_PROXY_ARGS` environment variable to the value `--traceZipkinListenerPorts 9411`. Also, edit the Wavefront Deployment to expose the containerPort `9411`.

4. Confirm that the Wavefront Proxy is running and working.

First, check if pods are running. For further information on how to test a proxy, see [Test a](#)

Proxy.

```
kubectl get pods -n ${WF_NAMESPACE}
```

Where: `WF_NAMESPACE` is the namespace where you deployed the Wavefront Proxy.

5. Edit the Serving ConfigMap `config-tracing` to enable the Zipkin tracing integration.

You can configure Cloud Native Runtimes to send traces 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`.



Note

There are two ways of editing a Knative ConfigMap on Cloud Native Runtimes. Depending on your installation, you may be able to edit the ConfigMap directly on the cluster or via overlays. Check the following documentation on how to edit ConfigMaps using overlays: [Configuring Cloud Native Runtimes](#)

The snippet below is an example of a Kubernetes secret containing a ytt overlay with the suggested changes to the ConfigMap `config-tracing`.

```
apiVersion: v1
kind: Secret
metadata:
  name: cnrs-patch
stringData:
  patch.yaml: |
    #@ load("@ytt:overlay", "overlay")
    #@overlay/match by=overlay.subset({"kind":"ConfigMap","metadata":{"name":"config-tracing","namespace":"knative-serving"}})
    ---
    data:
      #@overlay/match missing_ok=True
      backend: "zipkin"
      #@overlay/match missing_ok=True
      zipkin-endpoint: "http://wavefront-proxy.default.svc.cluster.local:9411/api/v2/spans"
```

Once you follow the steps in the [Customizing Cloud Native Runtimes](#) documentation to configure your installation to use the above overlay, you can check the ConfigMap on the cluster to confirm that the changes were applied.

```
kubectl get configmap config-tracing --namespace knative-serving --output yaml
```

The ConfigMap should then look like this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-tracing
data:
```

```

_example: |
  ...
  backend: "zipkin"
  zipkin-endpoint: "http://wavefront-proxy.default.svc.cluster.local:9411/api/v2/
  spans"

```

Other resources:

- [OpenTelemetry Integration](#)
- [Wavefront Proxies](#)
- [Deploy a Wavefront Proxy in Kubernetes \(Manual Install\)](#)
- [Managing API Tokens on Wavefront](#)

Configuring Cloud Native Runtimes with Avi Vantage

This topic tells you how to configure Cloud Native Runtimes, commonly known as CNRs, with Avi Vantage.

Overview

You can configure Cloud Native Runtimes 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 while the Avi Service Engines provide a data plane. Once set up, 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.

If you have not already installed Cloud Native Runtimes, see [Installing Cloud Native Runtimes](#). If you already have a Contour installation on your cluster, 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](#).

- (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.

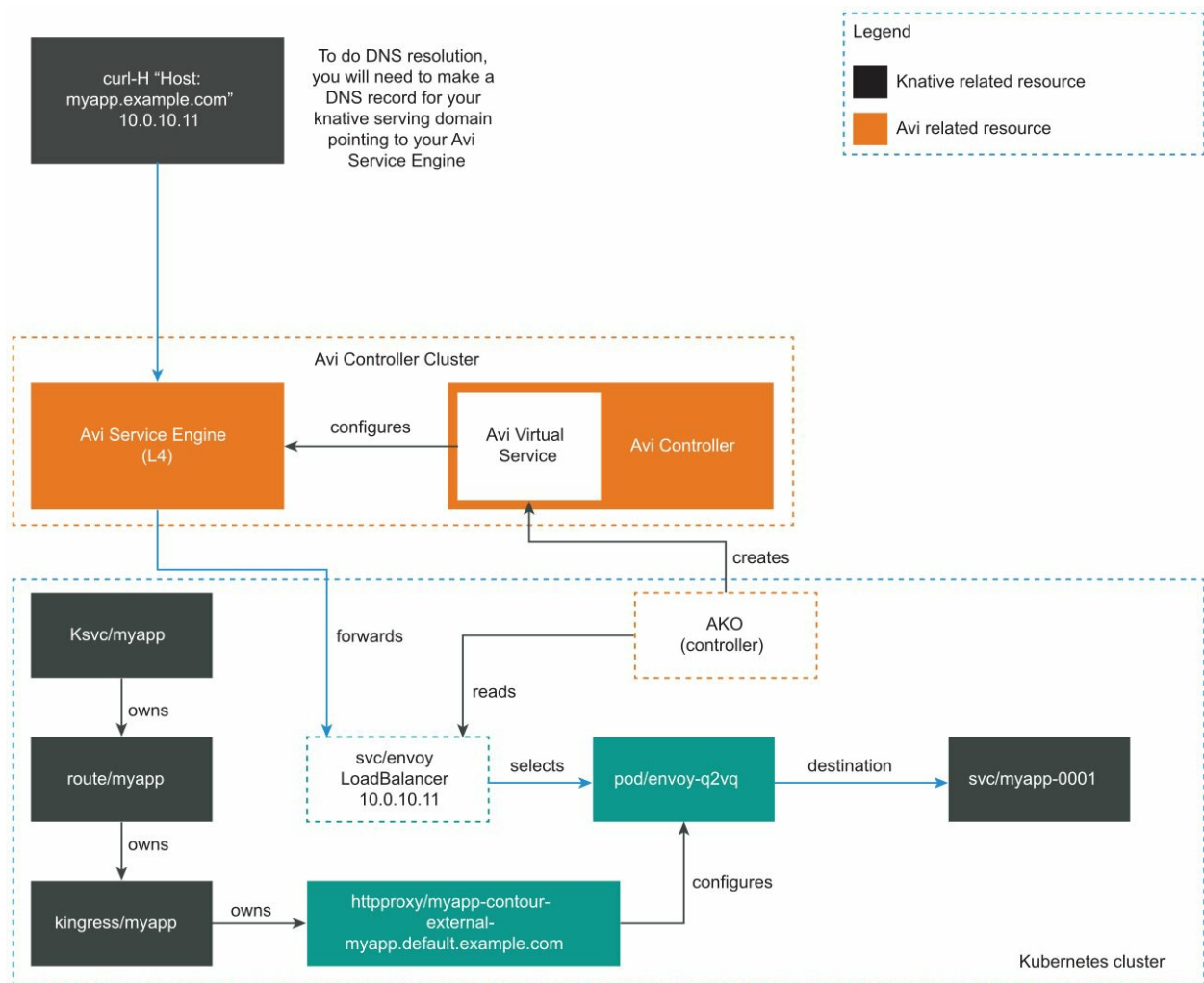
To get the KService URL, run:

```
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 with Tanzu Service Mesh

This topic tells you how to configure Cloud Native Runtimes, commonly known as CNRs, with Tanzu Service Mesh.

Overview

You cannot install Cloud Native Runtimes on a cluster that has Tanzu Service Mesh attached.

This workaround describes how Tanzu Service Mesh can be configured to ignore the Cloud Native Runtimes. 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-NS
 - ✦ knative-serving
 - ✦ knative-eventing

- ✦ knative-sources
- ✦ triggermesh
- ✦ vmware-sources
- ✦ tap-install
- ✦ rabbitmq-system
- ✦ kapp-controller
- ✦ The namespace or namespaces where you plan to run Knative workloads.

Where CONTOUR-NS is the namespace(s) where Contour is installed on your cluster. If Cloud Native Runtimes was installed as part of a Tanzu Application Profile, this value will likely be `tanzu-system-ingress`.

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.

Customizing Cloud Native Runtimes

There are many package configuration options exposed through data values that allows you to customize your Cloud Native Runtimes installation.

The following command yields all the configuration options available in a given Cloud Native Runtimes package version.

```
export CNR_VERSION=2.3.1
tanzu package available get cnrs.tanzu.vmware.com/${CNR_VERSION} --values-schema -n ta
p-install
```

Customizing Cloud Native Runtimes

Besides utilizing the out-of-the-box options to configure your package, you can use ytt overlays to further customize your installation. See [Customize your package installation](#) for instructions on how to customize any Tanzu Platform Application package.

This section will provide an example on how to update the Knative ConfigMap `config-logging` to override the logging level of the Knative Serving controller to `debug`.

1. Create a Kubernetes secret containing the ytt overlay by applying the configuration below to your cluster.

```
kubectl apply -n tap-install -f - << EOF
apiVersion: v1
kind: Secret
```



```

metadata:
  name: cnrs-patch
stringData:
  patch.yaml: |
    #@ load("@ytt:overlay", "overlay")
    #@overlay/match by=overlay.subset({"kind":"ConfigMap","metadata":{"name":"co
nfig-logging","namespace":"knative-serving"}})
    ---
    data:
      #@overlay/match missing_ok=True
      loglevel.controller: "debug"
EOF

```

To learn more about the Carvel tool `ytt` and how to write overlays, see their [official documentation](#).

2. Update your `tap-values.yaml` file to add the snippet below.

The section below informs the Tanzu Application Platform about the secret name where the overlay is stored and also, to apply the overlay to the `cnrs` package.

```

package_overlays:
- name: cnrs
  secrets:
  - name: cnrs-patch

```

Tip: You can retrieve your `tap-values.yaml` file by running the command below.

```

kubectl get secret tap-tap-install-values -n tap-install -ojsonpath="{.data.tap
-values\.yaml}" | base64 -d

```

3. Update the Tanzu Application Platform installation.

```

tanzu package installed update tap -p tap.tanzu.vmware.com -v ${TAP_VERSION} --
values-file tap-values.yaml -n tap-install

```

4. Confirm your changes were applied to the corresponding ConfigMap.

By running the command below, you can check if your changes were applied to the ConfigMap `config-logging` by ensuring `loglevel.controller` is set to `debug`.

```

kubectl get configmap config-logging -n knative-serving -oyaml

```

Troubleshooting Cloud Native Runtimes

This topic tells you how to troubleshoot Cloud Native Runtimes, commonly known as CNRs, installation or configuration.

Updates fail with error annotation value is immutable

Symptom

After upgrading to Tanzu Application Platform v1.6.4 or later, if you attempt to update a web workload created in Tanzu Application Platform v1.6.3 or earlier you see the following error:

```
API server says: admission webhook "validation.webhook.serving.knative.dev" denied the
request: validation failed: annotation value is immutable: metadata.annotations.servi
ng.knative.dev/creator (reason: BadRequest)
```

Explanation

Kapp controller, which is the orchestrator underneath workloads, deploys resources exactly as requested. However, Knative adds annotations to Knative Services to track the creator and last modified time of a resource. This conflict between kapp controller and Knative is a known issue and expected behavior that is mitigated by a kapp configuration that the supply chain defines and uses at deploy time. The kapp config specifies that the annotations Knative adds must not be modified during updates.

As of Tanzu Application Platform v1.6.4, the kapp configuration moved from the delivery supply chain to the build supply chain. When a web workload is being updated, the delivery supply chain no longer provides the kapp configuration, which causes the validation error. Although the kapp configuration exists on v1.6.4 in a different part of the supply chain, existing deliverables are not rebuilt to include it.

Solution

To workaroud this issue:

1. Deploy the following overlay as a secret to your Tanzu Application Platform installation namespace. In the following example, Tanzu Application Platform is installed to the `tap-install` namespace:

```
apiVersion: v1
kind: Secret
metadata:
  name: old-deliverables-patch
  namespace: tap-install #! namespace where tap is installed
stringData:
  app-deploy-overlay.yaml: |
    #@ load("@ytt:overlay", "overlay")

    #@ def kapp_config_replace(left, right):
    #@ return left + "\n" + right
    #@ end

    #@overlay/match by=overlay.subset({"kind": "ClusterDeploymentTemplate", "me
tadata": {"name": "app-deploy"}})
    ---
  spec:
    #@overlay/replace via=kapp_config_replace
    ytt: |
      #@ load("@ytt:overlay", "overlay")
      #@ load("@ytt:yaml", "yaml")

      #@ def kapp_config_temp():
      apiVersion: kapp.k14s.io/v1alpha1
      kind: Config
      rebaseRules:
        - path: [metadata, annotations, serving.knative.dev/creator]
```

```

    type: copy
    sources: [new, existing]
    resourceMatchers: &matchers
      - apiVersionKindMatcher: {apiVersion: serving.knative.dev/v1, kind:
Service}
      - path: [metadata, annotations, serving.knative.dev/lastModifier]
        type: copy
        sources: [new, existing]
        resourceMatchers: *matchers
    waitRules:
      - resourceMatchers:
        - apiVersionKindMatcher:
            apiVersion: serving.knative.dev/v1
            kind: Service
        conditionMatchers:
          - type: Ready
            status: "True"
            success: true
          - type: Ready
            status: "False"
            failure: true
    ownershipLabelRules:
      - path: [ spec, template, metadata, labels ]
        resourceMatchers:
          - apiVersionKindMatcher: { apiVersion: serving.knative.dev/v1, kind
: Service }
        #@ end

    #@overlay/match by=overlay.subset({"apiVersion": "kappctrl.k14s.io/v1alph
a1", "kind": "App", "metadata": { "name": data.values.deliverable.metadata.name
}})
    ---
    spec:
      fetch:
        #@overlay/append
        - inline:
          paths:
            overlay-config.yml: #@ yaml.encode(kapp_config_temp())

```

2. If you installed Tanzu Application Platform using a profile, apply the overlay to the `ootb-templates` package by following the instructions in [Customize a package that was installed by using a profile](#).

After you complete the steps, updates to the application will deploy.



Note

VMware plans to include a fix in future releases.

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.com
```

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 192.168.64.17:8443: connect: connection refused
kapp: Error: waiting on reconcile job/contour-certgen-v1.10.0 (batch/v1) namespace: CONTOUR-NS:
  Errored:
    Listing schema.GroupVersionResource{Group:"", Version:"v1", Resource:"pods"}, namespace: true:
      Get "https://192.168.64.17:8443/api/v1/pods?labelSelector=kapp.k14s.io%2Fassociation%3Dv1.572a543d96e0723f858367fcf8c6af4e": unexpected EOF
```

Where CONTOUR-NS is the namespace where Contour is installed on your cluster. If Cloud Native Runtimes was installed as part of a Tanzu Application Profile, this value will likely be `tanzu-system-ingress`.

Solution

Increase your available system RAM to at least 4 GB.

Pulling an image with `imgpkg` overwrites the `cloud-native-runtimes` directory

Symptom

When relocating an image to a private registry and later pulling that image with `imgpkg pull --lock LOCK-OUTPUT -o ./cloud-native-runtimes`, the contents of the `cloud-native-runtimes` are overwritten.

Solution

Upgrade the `imgpkg` version to v0.13.0 or later.

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) namespace: 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) namespace: cloud-native-runtimes
3:16:09PM: ^ Reconcile failed: (message: Deploying: Error (see .status.usefulErrorMessage 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="{.status.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 sub component 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 -o jsonpath="{.status.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-NS
6:30:22PM: ^ Load balancer ingress is empty
6:30:29PM: ---- waiting on 1 changes [322/323 done] ----
```

Where CONTOUR-NS is the namespace where Contour is installed on your cluster. If Cloud Native Runtimes was installed as part of a Tanzu Application Profile, this value will likely be `tanzu-system-ingress`.

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.io
(apiextensions.k8s.io/v1) cluster
10:51:58PM: fail: reconcile deployment/webhook (apps/v1) namespace: vmware-sources
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=v1.9621e0a793b4e925077dd557acedbcfe,pod-template-hash=6f5d979b7d,role=webhook,sources.tanzu.vmware.com/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.
```

- 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.

Knative Service Fails to Come up Due to Invalid HTTPProxy

Symptom

When creating a Knative Service, it does not reach ready status. The corresponding Route resource has the status `Ready=Unknown` with `Reason=EndpointsNotReady`. When you check the logs for the `net-contour-controller`, you see an error like this:

```
{ "severity": "ERROR", "timestamp": "2022-12-08T16:27:08.320604183Z", "logger": "net-contour-
-controller", "caller": "ingress/reconciler.go:313", "message": "Returned an error", "commi
t": "041f9e3", "knative.dev/controller": "knative.dev/net-contour.pkg/reconciler/contour.
Reconciler", "knative.dev/kind": "networking.internal.knative.dev/Ingress", "knative.dev/
traceid": "9d615387-f552-449c-a8cd-04c69dd1849e", "knative.dev/key": "cody/foo-java", "tar
getMethod": "ReconcileKind", "error": "HTTPProxy.projectcontour.io \"foo-java-contour-5f5
49ae3e6f584a5f33d069a0650c0d8foo-java.cody.\" is invalid: metadata.name: Invalid value
: \"foo-java-contour-5f549ae3e6f584a5f33d069a0650c0d8foo-java.cody.\": a lowercase RFC
1123 subdomain must consist of lower case alphanumeric characters, '-' or '.', and mu
st start and end with an alphanumeric character (e.g. 'example.com', regex used for va
lidation is '[a-z0-9]([-a-z0-9]*[a-z0-9])?(\\.[a-z0-9]([-a-z0-9]*[a-z0-9])?)*')", "stac
ktrace": "knative.dev/networking/pkg/client/injection/reconciler/networking/v1alpha1/in
gress.(*reconcilerImpl).Reconcile\n\tknative.dev/networking@v0.0.0-20221012062251-58f3
```

```
e6239b4f/pkg/client/injection/reconciler/networking/v1alpha1/ingress/reconciler.go:313
\nknative.dev/pkg/controller.(*Impl).processNextWorkItem\n\tknative.dev/pkg@v0.0.0-20221011175852-714b7630a836/controller/controller.go:542\nknative.dev/pkg/controller.(*Impl).RunContext.func3\n\tknative.dev/pkg@v0.0.0-20221011175852-714b7630a836/controller/controller.go:491"}
```

Solution

Due to a [known upstream Knative issue](#), certain combinations of Name + Namespace + Domain yield invalid names for HTTPProxy resources due to the way the name is hashed and trimmed to fit the size requirement. It can end up with non-alphanumeric characters at the end of the name.

Resolving this will be unique to each Knative service. It will likely involve renaming your app to be shorter so that after the hash + trim procedure, the name gets cut to end on an alphanumeric character.

For example, `foo-java.cody.iterate.tanzu-azure-lab.winterfell.fun` gets hashed and trimmed into `foo-java-contour-5f549ae3e6f584a5f33d069a0650c0d8foo-java.cody.`, leaving an invalid `.` at the end.

However, changing the app name to `foo-jav` will result in `foo-jav-contour-<some different hash>foo-jav.cody.it`, which is a valid name.

When using auto-tls, Knative Service Fails with `CertificateNotReady`.

Symptom

When creating a Knative Service, it does not reach ready status. The Knative Service has the status `CertificateNotReady`. When you check the status of the `kcert` resource that belongs to the Knative Service you see a message like this:

```
kubectl -n your-namespace get kcert route-76e387a2-cc35-4580-b2f1-bf7561371891 -ojsonpath='{.status}'
```

Output:

```
{
  "conditions": [
    {
      "lastTransitionTime": "2023-06-05T11:26:53Z",
      "message": "error creating Certmanager Certificate: cannot create valid length CommonName: (where-for-dinner.medium.longevityaks253.tapalong.cloudfocused.in) still longer than 63 characters, cannot shorten",
      "reason": "CommonName Too Long",
      "status": "False",
      "type": "Ready"
    }
  ],
  "observedGeneration": 1
}
```

Explanation

Due to a restriction imposed by cert-manager, CNs cannot be longer than 64 bytes. For more

information, see this [cert-manager issue](#) in GitHub. For Knative using cert-manager, this means that the FQDN for a Knative Service, usually comprised of `<ksvc name>.<namespace>.<domain>` but configurable using `domain_template` in Cloud Native Runtimes, must not exceed 64 bytes.

Recent improvements to Knative have been able to catch this in some cases. When `<ksvc name>.<namespace>` is longer than 25 characters, Knative will attempt to hash that value, and create a new common name in the form of `<hash>.<domain>`. However, if `<ksvc name>.<namespace>` is less than 25 characters long, it will not attempt to hash.

Knative is limited to a 25 character hash to preserve uniqueness in CommonNames. It also cannot shorten the domain portion, because that will break DNS resolution when performing [HTTP01 Challenges](#).

As a result, this catches some cases, but not all. It is possible that your `<domain>` portion is still too long.

There is an [issue in Knative Serving community](#) that aims to solve this.

Solution

The quickest way to avoid this is to disable TLS. See [Cloud Native Runtimes docs on disabling auto tls](#) for more details.

If you wish to continue using TLS, there are a few ways to resolve this on your own, though each comes with its own risks and limitations.

Option 1: Change the `domain_template`

Changing the `domain_template` alters how Knative will create FQDNs for Knative Services. See Cloud Native Runtimes instructions on [configuring External DNS](#).

You can use this option to shorten the template, either by shortening one of the fields:

```
{{.Name}}.{{slice .Namespace 0 3}}.{{.Domain}}
```

Note: Knative was not designed with shortening the name or namespace in mind. Due to a quirk in Knative's domain template validation, you can only slice up to a max of 3 characters.

Or by removing a field altogether:

```
{{.Name}}.{{.Domain}}
```

Warning: Removing the namespace from the `domain_template` makes it possible for Knative to create non-unique FQDNs for Knative Services across different namespaces. It will require manual care in naming Knative Services to make sure FQDNs remain unique.

Option 2: Shorten the names of Knative Services or Namespaces

Another option is to shorten the names of your Knative Services and/or Namespaces, if you have that ability. This will also require some manual calculation to make sure that the shortened Name, Namespace, and domain (including `.s`) come out to less than 64 bytes.

Verifying Your Installation

This topic tells you how to verify your Cloud Native Runtimes, commonly known as CNRs, installation. You can verify that your Cloud Native Runtimes installation was successful by testing Knative Serving, Knative Eventing, and TriggerMesh Sources for Amazon Web Services (SAWS).

Prerequisites

1. Create a namespace and environment variable where you want to create Knative services.
Run:

Note: This step covers configuring a namespace to run Knative services. If you rely on a SupplyChain to deploy Knative services into your cluster, skip this step because namespace configuration is covered in [Set up developer namespaces to use installed packages](#). Otherwise, you must complete the following steps for each namespace where you create Knative services.

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

2. Configure a namespace to use Cloud Native Runtimes. If during the Tanzu Application Platform installation you relocated images to another registry, you must grant service accounts that run Knative services using Cloud Native Runtimes access to the image pull secrets. This includes the `default` service account in a namespace, which is created automatically but not associated with any image pull secrets. Without these credentials, attempts to start a service fail with a timeout and the pods report that they are unable to pull the `queue-proxy` image.
 1. Create an image pull secret in the namespace Knative services will run and fill it from the `tap-registry` secret mentioned in [Add the Tanzu Application Platform package repository](#). Run the following commands to create an empty secret and annotate it as a target of the secretgen controller:

```
kubectl create secret generic pull-secret --from-literal=.dockerconfigjso
n={ } --type=kubernetes.io/dockerconfigjson -n ${WORKLOAD_NAMESPACE}

kubectl annotate secret pull-secret secretgen.carvel.dev/image-pull-secre
t="" -n ${WORKLOAD_NAMESPACE}
```

2. After you create a `pull-secret` secret in the same namespace as the service account, run the following command to add the secret to the service account:

```
kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name": "
pull-secret"}]}' -n ${WORKLOAD_NAMESPACE}
```

- Verify that a service account is correctly configured by running:

```
kubectl describe serviceaccount default -n ${WORKLOAD_NAMESPACE}
```

For example:

```
kubectl describe sa default -n cnr-demo
Name:                default
Namespace:          cnr-demo
Labels:              <none>
Annotations:        <none>
Image pull secrets: pull-secret
Mountable secrets:  default-token-xh6p4
Tokens:             default-token-xh6p4
Events:             <none>
```

The service account has access to the `pull-secret` image pull secret.

Verify that `STATUS` is `Reconcile succeeded`.

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

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

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

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

- 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

- Delete the namespace that you created for the demo. Run:

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

Verifying Knative Serving for Cloud Native Runtimes

This topic tells you how to verify that Knative Serving was successfully installed for Cloud Native Runtimes, commonly known as CNRs.

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 a namespace where you want to deploy Knative services. This namespace will be referred as `${WORKLOAD_NAMESPACE}` in this tutorial. See step 1 of [Verifying Your Installation](#) for more information.

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 `${WORKLOAD_NAMESPACE}` 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:

-
- la
 - ✦ Tanzu Kubernetes Grid on AWS
 - aS
 - ✦ Tanzu Mission Control on AWS
 - :
 - ✦ Amazon Elastic Kubernetes Service
-

```
R export EXTERNAL_ADDRESS=$(kubectl get service envoy -n tanzu-system-ingress -ojsonpath="
un {.status.loadBalancer.ingress[0].hostname}")
:
```

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

```
Ru export EXTERNAL_ADDRESS=$(kubectl get service envoy -n tanzu-system-ingress -ojsonpath="
n: {.status.loadBalancer.ingress[0].ip}")
```

- laaS:
 - ✦ Docker desktop
 - ✦ Minikube
-

```
Run: export EXTERNAL_ADDRESS='localhost:8080'
And set up port-forwarding in a separate terminal:
kubectl -n tanzu-system-ingress port-forward svc/envoy 8080:80
```

4. Connect to the app. Check the URL for the knative service.

Run:

```
kn service list -n ${WORKLOAD_NAMESPACE}
```

The result is something like this:

NAME	URL	LAT
EST	AGE CONDITIONS READY REASON	
hello-yeti	https://hello-yeti.\${WORKLOAD_NAMESPACE}.svc.cluster.local	hel
lo-yeti-00001	6s 3 OK / 3 True	

Now, take the host name from the URL and set it in an env variable `KSERVICE_HOSTNAME` like so:

```
export KSERVICE_HOSTNAME="hello-yeti.${WORKLOAD_NAMESPACE}.svc.cluster.local"
```

Then, connect to the app:

```
curl https://${KSERVICE_HOSTNAME} -k --resolve ${KSERVICE_HOSTNAME}:443:${EXTERNAL_ADDRESS}
```

Note: If you have configured DNS locally via `/etc/hosts` or externally, the `--resolve` flag can

be omitted, or you can use 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
unset KSERVICE_HOSTNAME
```

2. If you created port forwarding in step 3 above, then terminate that process.

Verify Knative Eventing with Cloud Native Runtimes

Eventing in Tanzu Application Platform is deprecated and marked for removal in Tanzu Application Platform v1.7.0.

This topic tells you how to verify that Knative Eventing was successfully installed with Cloud Native Runtimes, commonly known as CNRs.

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 a namespace where you want to deploy your Knative resources. This namespace will be referred to as `${WORKLOAD_NAMESPACE}` in this tutorial. See step 1 of [Verifying Your Installation](#) for more information.
- Create the following role binding in the `${WORKLOAD_NAMESPACE}` namespace. Run:

```
cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
```

```

kind: RoleBinding
metadata:
  name: ${WORKLOAD_NAMESPACE}-psp
  namespace: ${WORKLOAD_NAMESPACE}
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: eventing-restricted
subjects:
- kind: Group
  name: system:serviceaccounts:${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 [Configuring Eventing with RabbitMQ](#).
2. On the Kubernetes cluster where Eventing is installed, deploy a RabbitMQ cluster using the RabbitMQ Cluster Operator by running:

```

cat <<EOF | kubectl apply -f -
apiVersion: rabbitmq.com/v1beta1
kind: RabbitmqCluster
metadata:
  name: my-rabbitmq
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  replicas: 1
  override:
    statefulSet:
      spec:
        template:
          spec:
            securityContext: {}
            containers:
            - name: rabbitmq
              env:
              - name: ERL_MAX_PORTS
                value: "4096"
            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`.

3. Create a RabbitmqBrokerConfig

```

cat <<EOF | kubectl apply -f -
apiVersion: eventing.knative.dev/v1alpha1

```

```

kind: RabbitmqBrokerConfig
metadata:
  name: default-config
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  rabbitmqClusterReference:
    name: my-rabbitmq
    namespace: ${WORKLOAD_NAMESPACE}
  queueType: quorum
EOF

```

Verify Knative Eventing

To verify installation of Knative Eventing create and test a broker, procedure, and consumer in the `${WORKLOAD_NAMESPACE}` namespace:

1. Create a broker.

For the RabbitMQ broker. Run:

```

cat <<EOF | kubectl apply -f -
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: default
  namespace: ${WORKLOAD_NAMESPACE}
  annotations:
    eventing.knative.dev/broker.class: RabbitMQBroker
spec:
  config:
    apiVersion: eventing.knative.dev/v1alpha1
    kind: RabbitmqBrokerConfig
    name: default-config
    namespace: ${WORKLOAD_NAMESPACE}
EOF

```

For the in-memory broker. Run:

```

cat <<EOF | kubectl create -f -
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: apps/v1
kind: Deployment
metadata:
  labels:
    app: event-display
  name: event-display
  namespace: ${WORKLOAD_NAMESPACE}

```



```

spec:
  replicas: 1
  selector:
    matchLabels:
      app: event-display
  template:
    metadata:
      labels:
        app: event-display
    spec:
      containers:
        - name: user-container
          image: gcr.io/knative-releases/knative.dev/eventing-contrib/cmd/event
_display
      ports:
        - containerPort: 8080
          name: user-port
          protocol: TCP
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: event-display
  name: event-display-service
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 8080
  selector:
    app: event-display
EOF

```

3. Create a trigger. Run:

```

cat <<EOF | kubectl apply -f -
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: event-display
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  broker: default
  subscriber:
    ref:
      apiVersion: v1
      kind: Service
      name: event-display-service
      namespace: ${WORKLOAD_NAMESPACE}
EOF

```

4. Create a producer. This will send a message every minute. 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 deploy/event-display -n ${WORKLOAD_NAMESPACE} --since=10m --tail=50 -f

```

Setup RabbitMQ Broker as the default in the cluster (optional)

Eventing provides a `config-br-defaults` ConfigMap that contains the configuration setting that govern default Broker creation.

This example configuration will set RabbitMQ as the default broker on the cluster:

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: config-br-defaults
  namespace: knative-eventing
  labels:
    eventing.knative.dev/release: devel
data:
  default-br-config: |
    clusterDefault:
      brokerClass: RabbitMQBroker
      apiVersion: eventing.knative.dev/v1alpha1
      kind: RabbitmqBrokerConfig
      name: default-config
      namespace: ${WORKLOAD_NAMESPACE} # This should be the name of your namespace.
    delivery:
      retry: 3
      backoffDelay: PT0.2S
      backoffPolicy: exponential

```

To achieve this you can:

1. Run:

```

kubectl patch -n knative-eventing cm config-br-defaults --type merge --patch '{
  "data": {"default-br-config": "clusterDefault:\n      brokerClass: RabbitMQBroker\n      apiVersion: eventing.knative.dev/v1alpha1\n      kind: RabbitmqBrokerConfig\n      name: default-config\n      namespace: \''${WORKLOAD_NAMESPACE}'\n\n      delivery:\n        retry: 3\n        backoffDelay: PT0.2S\n        backoffPolicy: exponential"}
}'

```

```
ffPolicy: exponential\n"}}}'
```

2. Check that the ConfigMap looks as intended.

```
kubectl get -n knative-eventing cm config-br-defaults -oyaml
```

3. Now, to create a RabbitMQ broker you can run:

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

Eventing will automatically set the `brokerClass` to `RabbitMQBroker` and it will set up the `spec.config` to `RabbitmqBrokerConfig` with name `default-config`.

Setup RabbitMQ Broker as the default in a namespace (optional)

You can also use the `config-br-defaults` ConfigMap to set up the default broker configuration for a given namespace.

Let us suppose you want to have the `MTChannelBroker` as the default for the cluster and the `RabbitMQ Broker` as the default for your workload namespace.

To do this, we want that our `config-br-defaults` ConfigMap looks like this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-br-defaults
  namespace: knative-eventing
  labels:
    eventing.knative.dev/release: devel
data:
  default-br-config: |
    clusterDefault:
      brokerClass: MTChannelBasedBroker
      apiVersion: v1
      kind: ConfigMap
      name: config-br-default-channel
      namespace: knative-eventing
      delivery:
        retry: 10
        backoffDelay: PT0.2S
        backoffPolicy: exponential
    namespaceDefaults:
      ${WORKLOAD_NAMESPACE}: # This should be the name of your namespace.
        brokerClass: RabbitMQBroker
        apiVersion: eventing.knative.dev/v1alpha1
        kind: RabbitmqBrokerConfig
        name: default-config
        namespace: ${WORKLOAD_NAMESPACE} # This should be the name of your namespace.
```

```

delivery:
  retry: 3
  backoffDelay: PT0.2S
  backoffPolicy: exponential

```

To achieve this you can:

1. Run:

```

kubectl patch -n knative-eventing cm config-br-defaults --type merge --patch '{
"data": {"default-br-config": "clusterDefault:\n brokerClass: MTChannelBasedBr
oker\n apiVersion: v1\n kind: ConfigMap\n name: config-br-default-channel\n
namespace: knative-eventing\n delivery:\n retry: 10\n backoffDelay: PT0
.2S\n backoffPolicy: exponential\nnamespaceDefaults:\n "'${WORKLOAD_NAMESPA
CE}":\n brokerClass: RabbitMQBroker\n apiVersion: eventing.knative.dev/v
1alpha1\n kind: RabbitmqBrokerConfig\n name: default-config\n namespac
e: "'${WORKLOAD_NAMESPACE}""\n delivery:\n retry: 3\n backoffDelay
: PT0.2S\n backoffPolicy: exponential\n"}'}'

```

2. Check that the ConfigMap looks as intended.

```

kubectl get -n knative-eventing cm config-br-defaults -o yaml

```

3. With this configuration when you create a Broker in the `default` namespace it will be a `MTChannelBasedBroker`.

```

cat <<EOF | kubectl create -f -
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: broker-in-default-ns
  namespace: default
EOF

```

4. Check the type of this broker like so:

```

kubectl get -n default broker broker-in-default-ns -o yaml

```

It will show something like this:

```

apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  annotations:
    eventing.knative.dev/broker.class: MTChannelBasedBroker
  name: broker-in-default-ns
  namespace: default
spec:
  config:
    apiVersion: v1
    kind: ConfigMap
    name: config-br-default-channel
    namespace: knative-eventing
  delivery:
    backoffDelay: PT0.2S
    backoffPolicy: exponential
    retry: 10

```

Notice the `eventing.knative.dev/broker.class: MTChannelBasedBroker` annotation.

- Now try to create a Broker in the `${WORKLOAD_NAMESPACE}`.

```
cat <<EOF | kubectl create -f -
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: broker-in-workload-ns
  namespace: ${WORKLOAD_NAMESPACE}
EOF
```

- Check the type of this broker like so:

```
kubectl get -n ${WORKLOAD_NAMESPACE} broker broker-in-workload-ns -o yaml
```

It will show something like this:

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  annotations:
    eventing.knative.dev/broker.class: RabbitMQBroker
  name: broker-in-workload-ns
  namespace: ${WORKLOAD_NAMESPACE}
spec:
  config:
    apiVersion: eventing.knative.dev/v1alpha1
    kind: RabbitmqBrokerConfig
    name: default-config
    namespace: ${WORKLOAD_NAMESPACE}
  delivery:
    backoffDelay: PT0.2S
    backoffPolicy: exponential
    retry: 3
```

Notice the `eventing.knative.dev/broker.class: RabbitMQBroker` annotation.

Delete the Eventing Resources

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

- Delete the eventing resources:

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

- If you followed [Setup RabbitMQ Broker as the default in the cluster \(optional\)](#), delete the broker like so:

```
kubectl delete broker/broker-using-defaults -n ${WORKLOAD_NAMESPACE}
```

3. If you followed [Setup RabbitMQ Broker as the default in a namespace \(optional\)](#), delete the brokers like so:

```
kubectl delete broker/broker-in-default-ns -n default
kubectl delete broker/broker-in-workload-ns -n ${WORKLOAD_NAMESPACE}
```

4. If you created a RabbitMQ cluster:

```
kubectl delete rabbitmqbrokerconfig/default-config -n ${WORKLOAD_NAMESPACE}
kubectl delete rabbitmqcluster/my-rabbitmq -n ${WORKLOAD_NAMESPACE}
```

5. Delete the role binding:

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

Verifying TriggerMesh SAWS for Cloud Native Runtimes

This topic tells you how to verify that TriggerMesh Sources for Amazon Web Services (SAWS) was installed successfully for Cloud Native Runtimes, commonly known as CNRs.

Overview

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

Cloud Native Runtimes 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 [TriggerMesh](#) in GitHub.

The procedure below shows you how to test TriggerMesh SAWS using the example of an event source for Amazon CodeCommit.

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
- Have a namespace where you want to deploy Knative services. This namespace will be referred as `${WORKLOAD_NAMESPACE}` in this tutorial. See step 1 of [Verifying Your Installation](#) for more information.

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:

```

kubectl -n ${WORKLOAD_NAMESPACE} create secret generic awscreds \
--from-literal=aws_access_key_id=${AWS_ACCESS_KEY_ID} \
--from-literal=aws_secret_access_key=${AWS_SECRET_ACCESS_KEY}

```

Where:

- ◆ `AWS_ACCESS_KEY_ID` is the AWS access key ID for your AWS service account.
- ◆ `AWS_SECRET_ACCESS_KEY` is your AWS access key for your AWS service account.

5. Create the AWSCodeCommitSource:

```

kubectl apply -f - << EOF
apiVersion: sources.triggerrmesh.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. Patch the `awscodecommitsource-adapter` service account to pull images from the private registry using the `tap-registry` secret, created during the TAP installation. Note that the `awscodecommitsource-adapter` service account was created on the previous step during the creation of `AWSCodeCommitSource`.

```

kubectl patch serviceaccount -n ${WORKLOAD_NAMESPACE} awscodecommitsource-adapter -p '{"imagePullSecrets": [{"name": "tap-registry"}]}'

```

Note: It may be necessary to delete the current `awscodecommitsource-source` **Pod** so a new pod is created with the new `imagePullSecrets`.

7. Create an event by pushing a commit to your CodeCommit repository.
8. 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 ${WORKLOAD_NAMESPACE} --since=10m --tail=50

```


Upgrading Cloud Native Runtimes

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

New versions of Cloud Native Runtimes are available from the Tanzu Application Platform package repository, and can be upgraded to as part of [upgrading Tanzu Application Platform as a whole](#).

Prerequisites

The following prerequisites are required to upgrade Cloud Native Runtimes:

- An updated Tanzu Application Platform package repository with the version of Cloud Native Runtimes you wish to upgrade to. For more information, see the [documentation on adding a new package repository](#).

Upgrade Cloud Native Runtimes



Note

If you previously installed Cloud Native Runtimes v1.3 or prior and, you wish to upgrade to the latest version, you must first upgrade to Cloud Native Runtimes v2.0.1. You can do so by following the [Upgrade from a previous version to Cloud Native Runtimes v2.0.1](#) instructions.

If you have previously installed Cloud Native Runtimes v1.3 or an earlier version and wish to upgrade to the latest version, please be aware that the Tanzu Application Platform now includes a shared ingress issuer by default. If you are currently using a single certificate (for example, if you have set `cnrs.default_tls_secret` in your `tap-values.yaml` file) and want to opt out of the default shared ingress issuer, it is important to deactivate it. To learn how to opt out and deactivate the automatic TLS feature, please refer to the documentation: [Opt out from any ingress issuer and deactivate automatic TLS feature](#).

To upgrade the Cloud Native Runtimes PackageInstall specifically, run:

```
tanzu package installed update cloud-native-runtimes -p cnrs.tanzu.vmware.com -v CNR-VERSION --values-file cnr-values.yaml -n tap-install
```

Where `CNR-VERSION` is the latest version of Cloud Native Runtimes available as part of the new Tanzu Application Platform package repository.

Uninstalling Cloud Native Runtimes

This topic tells you how to uninstall Cloud Native Runtimes.

Overview

Cloud Native Runtimes is part of the Tanzu Application Platform package repository. For information on uninstalling the entire Tanzu Application Platform package repository, see the [Tanzu Application Platform uninstall documentation](#).

Uninstall

To uninstall Cloud Native Runtimes specifically:

1. Delete the installed package:

```
tanzu package installed delete cloud-native-runtimes --namespace tap-install
```