

CN2 Pipelines for GitOps Guide

Published
2023-09-28

Juniper Networks, Inc.
1133 Innovation Way
Sunnyvale, California 94089
USA
408-745-2000
www.juniper.net

Juniper Networks, the Juniper Networks logo, Juniper, and Junos are registered trademarks of Juniper Networks, Inc. in the United States and other countries. All other trademarks, service marks, registered marks, or registered service marks are the property of their respective owners.

Juniper Networks assumes no responsibility for any inaccuracies in this document. Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice.

CN2 Pipelines for GitOps Guide

Copyright © 2023 Juniper Networks, Inc. All rights reserved.

The information in this document is current as of the date on the title page.

YEAR 2000 NOTICE

Juniper Networks hardware and software products are Year 2000 compliant. Junos OS has no known time-related limitations through the year 2038. However, the NTP application is known to have some difficulty in the year 2036.

END USER LICENSE AGREEMENT

The Juniper Networks product that is the subject of this technical documentation consists of (or is intended for use with) Juniper Networks software. Use of such software is subject to the terms and conditions of the End User License Agreement ("EULA") posted at <https://support.juniper.net/support/eula/>. By downloading, installing or using such software, you agree to the terms and conditions of that EULA.

Table of Contents

About This Guide | vi

1

Install and Manage

CN2 Pipelines | 2

CN2 Pipelines Overview | 2

CN2 Configuration | 2

CN2 and GitOps | 4

Prerequisites | 7

CN2 Pipelines Installation and Setup | 7

Before You Install CN2 Pipelines | 9

Install Helm | 9

Verify Kubeconfig | 9

CN2 Cluster Configuration | 10

Create a Personal Access Token for GitLab | 10

Mountpath and Profiles | 11

Create a Sample ConfigMap in Git Server Folder | 11

Update /etc/hosts for an OpenShift Deployment | 12

Install CN2 Pipelines | 12

Download CN2 Pipelines | 12

Install the CN2 Pipelines Helm Chart | 13

Verify the CN2 Pipelines Helm Chart Installation | 14

Argo CD and Helm Configuration | 15

Argo Log In | 16

CN2 and Workflows | 17

CN2 Pipelines Service | 18

CN2 Pipelines Configurations | 18

Create Custom Workflows for the CN2 Pipelines | 20

Explanation of values.yaml | 23

2

Architecture and Design

CN2 Pipelines Solution Test Architecture and Design | 27

Overview | 27

Use Case | 27

Test Workflows | 28

Profiles | 29

Test Environment Configuration | 36

Test Execution with Micro-Profiles | 37

Logging and Reporting | 38

3

Test Cases

CN2 Pipelines Test Management | 40

Trigger the CN2 Pipelines Test | 40

Support for Queuing Test Execution | 40

Change Commit Threshold Trigger | 41

Change Test Profiles | 41

Identify CN2 Pipelines Test Trigger | 42

Access Test Results | 43

Uninstall the CN2 Pipelines | 43

CN2 Pipelines Test Cases | 44

CN2 Pipelines Test Case Descriptions | 53

Architect Onboard | 54

Architect Execute | 54

Architect Teardown | 55

SRE Onboard | 55

SRE Execute | 56

SRE teardown | 58

About This Guide

This guide provides an understanding of the features and test cases for Juniper Cloud-Native Contrail® Networking™ (CN2) Pipelines Release 23.2.

1

CHAPTER

Install and Manage

[CN2 Pipelines | 2](#)

[Before You Install CN2 Pipelines | 9](#)

[Install CN2 Pipelines | 12](#)

[Explanation of values.yaml | 23](#)

CN2 Pipelines

SUMMARY

Juniper Cloud-Native Contrail Networking (CN2) Pipelines is a CI/CD tool to enable GitOps-based workflows to automate CN2 configuration, testing, and qualification. CN2 Pipelines runs alongside CN2 clusters starting with CN2 Release 23.1.

IN THIS SECTION

- [CN2 Pipelines Overview | 2](#)
- [CN2 Configuration | 2](#)
- [CN2 and GitOps | 4](#)
- [Prerequisites | 7](#)
- [CN2 Pipelines Installation and Setup | 7](#)

CN2 Pipelines Overview

GitOps is a deployment methodology centralized around a Git repository, where the GitOps workflow pushes a configuration through testing, staging, and production. Many customers run a staging environment or staging cluster. The GitOps process supports automatic configuration to deploy and test CN2 network configurations using test case YAML files.

After you (the administrator) configure the CN2 Pipelines and GitOps, CN2 Pipelines will:

- Sync with the GitOps repository and auto-provision CN2 configurations to the Kubernetes cluster.
- Provision CN2 configurations with the capability to test and verify the deployed CN2 configurations in each Kubernetes cluster.
- Provide auto-revision monitoring and updates.

CN2 Configuration

IN THIS SECTION

- [GitOps | 3](#)

CN2 uses Kubernetes Custom Resource Definitions (CRDs) configurations written in YAML or JSON format. These CRDs are stored and managed in the Git repository, which makes the Git repository the source of truth for all of the network configurations.

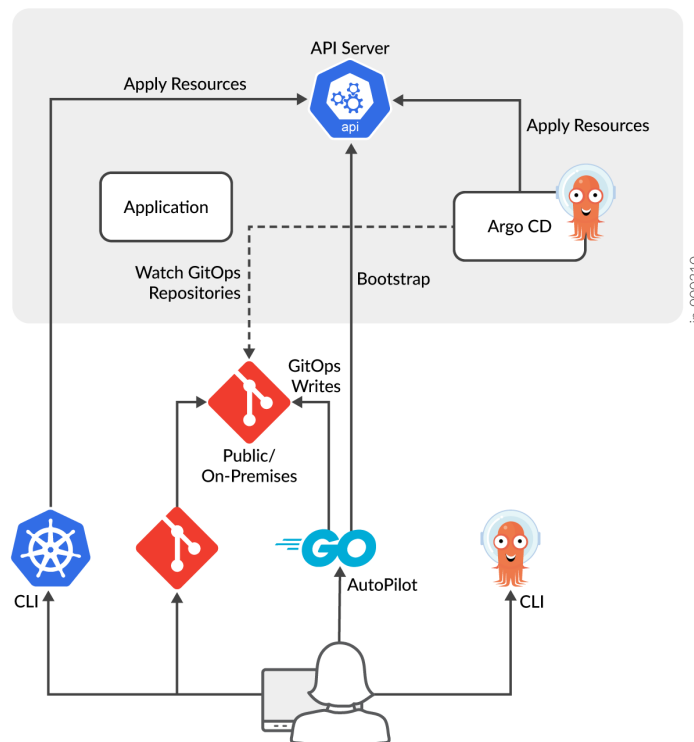
GitOps

"GitOps is a paradigm or a set of practices that empowers developers to perform tasks which typically fall under the purview of IT operations. GitOps requires us to describe and observe systems with declarative specifications that eventually form the basis of continuous everything." Quote is from *CloudBees*.

To achieve the GitOps mode of operation, the Argo CD application is used. The CN2 Git repository is configured to be used as the Argo CD application. Argo CD is a declarative, GitOps continuous delivery tool for Kubernetes. See [Figure 1 on page 3](#).

The GitOps engine also runs a repository server that caches all of the application files from the Git repository. These files are verified and monitored for any CN2 configuration changes received to the Git repository.

Figure 1: Argo CD with Git Repository and Kubernetes



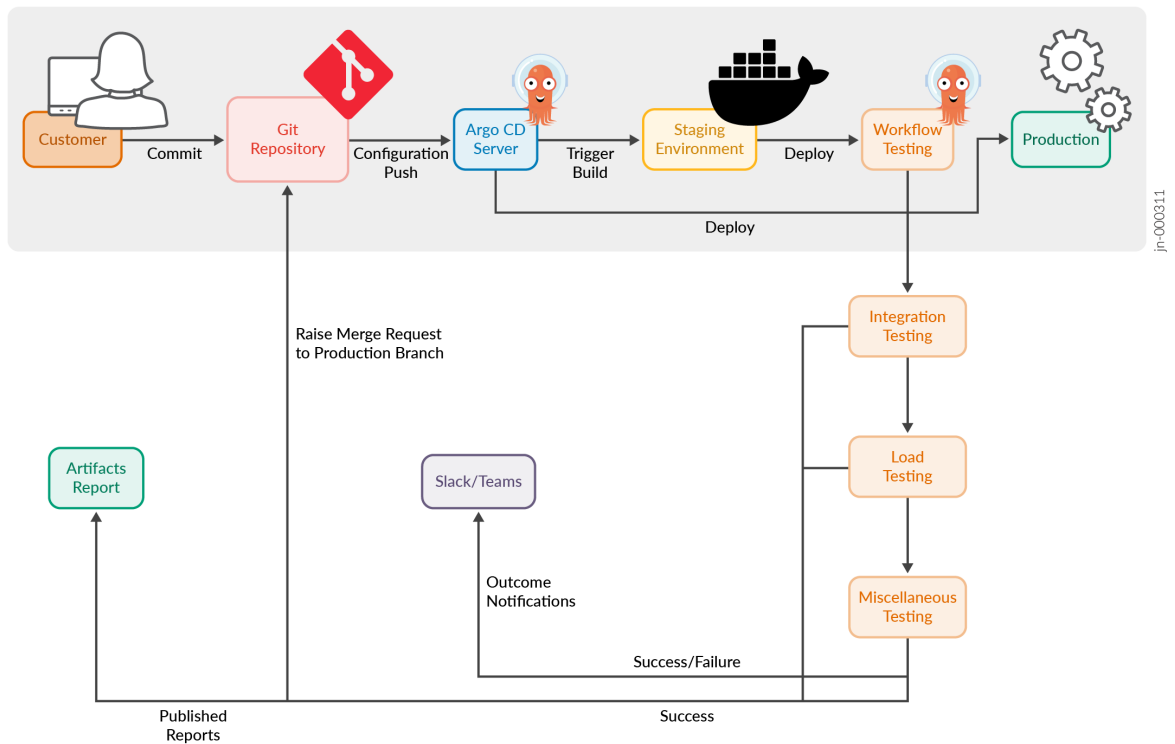
CN2 and GitOps

IN THIS SECTION

- CN2 Pipelines Configuration Flow | 5
- GitOps Server | 5
- Workflow and Tests | 6

The primary benefit of supporting GitOps for CN2 is to achieve automatic configuration deployment and testing of CN2 network configurations. CN2 configurations are custom resource definitions (CRDs) which are maintained in a Git repository. These CRDs are applied to the CN2 cluster whenever there is a change to the CN2 configurations in the Git repository. To test and apply these changes, the GitOps applications Argo CD and Argo Workflows are utilized.

Figure 2: GitOps Pipelines Workflow

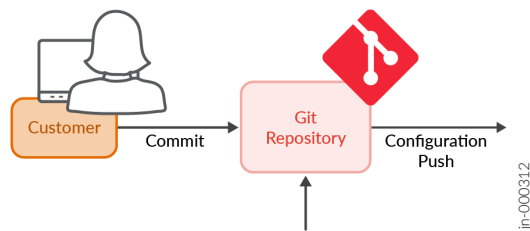


CN2 Pipelines Configuration Flow

Your CN2 configurations are maintained in the CN2 Git repository. The CN2 Git repository is configured to be used as the Argo CD application. The CN2 Pipelines configuration flow is as follows:

1. CN2 configurations are initially pushed to the staging repository by you (the administrator).
2. Any changes to the configurations in the repository triggers a Git synchronization to the GitOps server.
3. The GitOps server looks for any changes by comparing the current configuration and the new configuration fetched from the CN2 Git repository.
4. If any new changes are pushed, the GitOps server applies these changes to the CN2 environment.

Figure 3: CN2 Configurations in Customer Git Repository



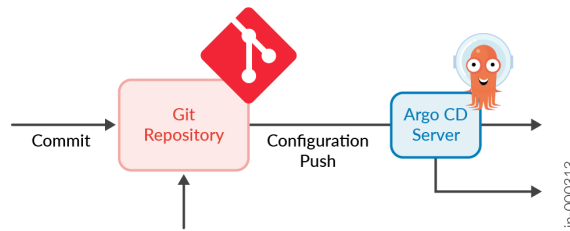
GitOps Server

The GitOps server confirms that the configuration in the CN2 environment is always synchronized with the Git repository. CN2 Pipelines supports two branches:

- One for the staging environment.
- One for the production environment.

Many customers run a staging environment or staging cluster. The staging branch is where you (the administrator) push any configurations required to be pushed to the staging CN2 cluster. These configurations are then tested by the workflow engine before the configurations are merged to the production branch.

Figure 4: GitOps Server

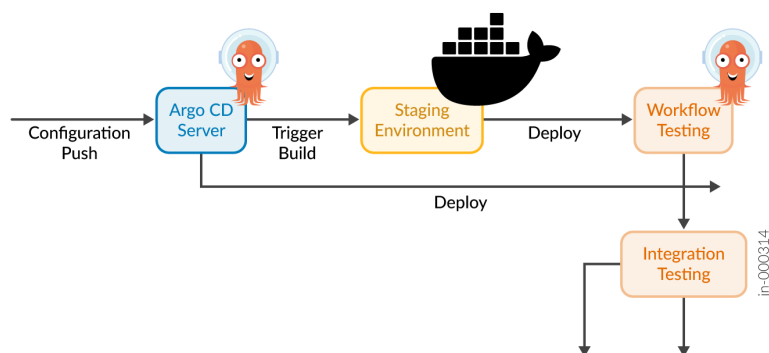


Workflow and Tests

The GitOps server pushes all configuration changes to your CN2 setup as follows:

1. This push triggers the workflow cycle to run test cases. These test cases validate and verify the CN2 setup against the configuration you applied in the staging setup.
2. If the test cases are successful, you are notified about the test completion and a merge request is presented to the production branch.
3. Next, you need to validate the changes in the merge request and approve the changes to be merged to the production branch.
4. After the new configurations are merged to the production branch, the GitOps server synchronizes the configurations and applies the configurations to the CN2 production cluster.

Figure 5: Workflow and Tests



Prerequisites

Before you install CN2 Pipelines, verify you have the following:

- Management Kubernetes cluster (where you will install CN2 Pipelines)
- CN2 Kubernetes cluster
- Connectivity from the management Kubernetes cluster to the CN2 cluster
- GitLab repository with CN2 configuration folder with sample ConfigMap file
See ["Create a Sample ConfigMap in Git Server Folder" on page 11](#)
- MountPath folder
- Connectivity from the management Kubernetes cluster to outside, needed to access Argo CD, Argo Workflows, and test results
- Notes:
 - If you are using Red Hat OpenShift with CN2 Pipelines, install ingress from the files `/ingress/openshift/public` on the OpenShift cluster
 - CN2 Pipelines needs GitLab or GitLab for Open Source as an event source
 - CN2 pipeline requires a separate GitLab project per CN2 cluster. So, each CN2 cluster requires a separate GitLab project to be created for storing the CN2 configuration (config).
 - In the case of file deletion, if the commit-processing workflow fails, you are required to do a dummy commit.

CN2 Pipelines Installation and Setup

IN THIS SECTION

- [Components | 8](#)
- [CN2 Components | 8](#)
- [Kubernetes | 8](#)

Components

CN2 Pipelines installs and configures the following components:

- Argo CD
- Argo Workflows
- Argo Events
- CN2 Pipelines services
- Configure, upload and trigger CN2 testing workflows
- Supports customer container network functions (CNFs)

CN2 Components

All CN2 Pipelines components are installed and configured as part of the CN2 Pipelines Helm chart installation. Argo CD is one of the components in CN2 Pipelines and it is installed in the management cluster.

Argo CD is configured with the following details during the initial setup:

- CN2 cluster environment details
- Git repository access details
- CN2 GitOps engine application configuration

See ["Install the CN2 Pipelines Helm Chart" on page 13](#).

Kubernetes

You can use any native Kubernetes or Red Hat OpenShift with CN2 or another Container Network Interface (CNI) to provision CN2 Pipelines.

Before You Install CN2 Pipelines

SUMMARY

The following procedures will help you obtain some prerequisites and some values used to fill the `values.yaml` file for the CN2 Pipelines Helm chart.

IN THIS SECTION

- [Install Helm | 9](#)
- [Verify Kubeconfig | 9](#)
- [CN2 Cluster Configuration | 10](#)
- [Create a Personal Access Token for GitLab | 10](#)
- [Mountpath and Profiles | 11](#)
- [Create a Sample ConfigMap in Git Server Folder | 11](#)
- [Update `/etc/hosts` for an OpenShift Deployment | 12](#)

Install Helm

Before installing the CN2 Pipelines chart, you need to install Helm 3 in the management cluster. Helm helps you manage Kubernetes applications. Helm charts help you define, install, and upgrade even the most complex Kubernetes application.

Run the following command to download and install the latest version of Helm 3:

```
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3
chmod 700 get_helm.sh
./get_helm.sh
```

Verify Kubeconfig

Before creating the kubeconfig file as base64, verify kubeconfig works from the management cluster.

1. Copy the kubeconfig file from CN2 to the management cluster. You can do this with a copy and paste.
2. Run the following command to view the nodes on the CN2 cluster:

```
kubect1 get nodes --kubeconfig=<kubeconfig_file_of_CN2_cluster>
```

3. Run the following command to view the all the pods on the CN2 cluster:

```
kubect1 get pods -A --kubeconfig=<kubeconfig_file_of_CN2_cluster>
```

CN2 Cluster Configuration

CN2 cluster configuration performs the following actions in the CN2 cluster:

- Creates CN2 Pipelines namespace if namespace does not exist already.
- Creates a service account named cn2pipelines.
- Applies the cluster role and role bindings.

Based on the above items, CN2 Pipelines creates a dynamic bearer token to communicate with the CN2 cluster immediately during the provisioning of CN2 Pipelines with Argo CD.

Mountpath for CN2 Cluster Configuration

Place the CN2 cluster configuration with the name config in the mountpath specified in the values.yaml.

For example:

```
mountpath: /opt/cn2_workflows/config
```

Create a Personal Access Token for GitLab

To create a personal access token, use the following procedure from GitLab:

1. Select **Edit profile**.
2. In the left pane, select **Access Tokens**.

3. Enter a name and (optional) expiration date for the token.
Default expiration is 30 days.
4. Select the desired scopes.
See [GitLab Personal Access Token Scopes](#).
5. Select **Create personal access token**.
6. Save the personal access token somewhere safe. After you leave the page, you no longer have access to the token.
For more information, see [GitLab Personal Access Token](#).

Mountpath and Profiles

You need to put the mountpath in a mountpath folder, then create your profiles in the mountpath folder. For example, if your mountpath is `/opt/cn2_workflows` as defined in the `values.yaml`, you will create a folder named `/opt/cn2_workflows`.

Create a Sample ConfigMap in Git Server Folder

You need to create a sample ConfigMap before installing the CN2 Pipelines. Create and add the sample ConfigMap to the CN2 network configuration folder identified in your GitLab server branch. The same branch and folder also needs to be added in the `values.yaml`. This ConfigMap gets applied by Argo CD as part of the CN2 Pipelines installation.

1. Run the following command to create a ConfigMap with the filename `cn2configmap`:

```
cat <<'EOF'>> cn2configmap.yaml
```

Output:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: pipelines-config
  namespace: default
data:
  cn2pipeline: "true"
```

2. Commit the ConfigMap file to the CN2 configuration folder identified in your Git server branch.

Update /etc/hosts for an OpenShift Deployment

On the CN2 cluster, check the ingress components.

Verify that /etc/hosts contains entries from the OpenShift cluster. For example:

```
192.168.19.571 api.ocp-ss-571.net
```

Install CN2 Pipelines

SUMMARY

This section guides you through installing CN2 Pipelines.

IN THIS SECTION

- [Download CN2 Pipelines | 12](#)
- [Install the CN2 Pipelines Helm Chart | 13](#)
- [Verify the CN2 Pipelines Helm Chart Installation | 14](#)
- [Argo CD and Helm Configuration | 15](#)
- [Argo Log In | 16](#)
- [CN2 and Workflows | 17](#)
- [CN2 Pipelines Service | 18](#)
- [CN2 Pipelines Configurations | 18](#)
- [Create Custom Workflows for the CN2 Pipelines | 20](#)

Download CN2 Pipelines

Download the CN2 Pipelines files to update the files with the needed tokens prior to installation.

To download the CN2 Pipelines tar file:

1. Download CN2 Pipelines Deployer files from [Juniper Networks Downloads](#).
2. Untar the downloaded files to the management server.

Install the CN2 Pipelines Helm Chart

The CN2 Pipelines Helm chart is used to install and configure the CN2 Pipelines management cluster.

To install the CN2 Pipelines Helm chart on your management cluster:

1. In your downloaded CN2 Pipelines Deployer files, locate the values.yaml in the folder **contrail-pipelines-x.x.x/values**.
2. Input the chart values. For parameter descriptions, see ["Explanation of values.yaml" on page 23](#).

Example CN2 Pipelines values.yaml for the management cluster:

```
#####
#           Common Configuration (global vars)           #
#####
global:
  docker_image_repo: docker.io # Global docker registry for non Juniper images
  registry: enterprise-hub.juniper.net/contrail-container-prod/ # Global image registry to
pull Juniper artifacts
  imagePullSecret: <base64 imagePullSecret> # Image pull secret for authenticated registry ##
Keep this commented for nonAuthenticated registry
  deployment_type: 'k8s' # deployment_type: k8s for CN2 kubernetes cluster (or)
deployment_type: "openshift" for CN2 openshift cluster
  managementServer: <managementServer> # CN2 pipeline management server IP

  gitServer:
    access_token: <access_token> # eg: eTE1Y0p1Ml04TGhiWFpfLTFSVEg=
    gitlabBaseURL: <gitlabBaseURL> # eg: https://cnf-gitlab.net
    project: <project> # eg: devops/cn2/cn2-pipelines
    folderName: <folderName> # eg: cn2networkconfig
    branch: <branch> # eg: master

  cn2ClusterDetails:
    name: <cluster name> # CN2 cluster name
    server: <kubeAPI IP> # CN2 cluster kubeapi server (should be reachable from management
server)
```

```
kubeconfig: cn2-cluster-kubeconfig # CN2 kubeconfig name, leave as default
mountpath: /opt/cn2_workflows # CN2 test profile folder location
```

```
workflow-objects:
  ssl_enabled: True # True if CN2 cluster deployed with SSL enabled else it is False
  ## ''' Enable below OCP keys only for deployment_type is openshift ''' ##
  #ocp_api_host_ip: <ocp_api_host_ip> # eg: '192.167.19.571'
  #ocp_api_host_name: <ocp_api_host_name> # eg: 'api.ocp-ss-571.net'
```

3. Run the following command to install the CN2 Pipelines Helm chart with the release name cn2-pipeline:

```
helm install cn2-pipeline . --timeout=20m
```

Verify the CN2 Pipelines Helm Chart Installation

To verify the CN2 Pipelines Helm chart Installation, run the following commands:

1. List the Helm release in the current namespace.

```
helm ls
```

Output:

NAME	NAMESPACE	REVISION	UPDATED
STATUS	CHART	APP VERSION	
cn2-pipeline	default	1	2023-16-10 11:44:29.380155158 +0000 UTC
deployed	cn2-pipeline-1	23.2.0	

2. Display all pods in all namespaces.

```
kubectl get pods -A
```

Output:

NAMESPACE	NAME	READY	STATUS
RESTARTS	AGE		

argo-events	controller-manager-844d44-vf6r8	1/1	Running	
0	2m23s			
argo-events	eventbus-default-stan-0	2/2	Running	
0	2m18s			
argo-events	eventbus-default-stan-1	2/2	Running	
0	2m6s			
argo-events	eventbus-default-stan-2	2/2	Running	
0	2m4s			
argo-events	events-webhook-64dc49f456-p6rmw	1/1	Running	
0	2m23s			
argo-events	gitlab-eventsources-qhznz8-74c4c785dc-ggmpr	1/1	Running	2 (118s
ago)	2m17s			
argo-events	gitlab-sensor-xc5s6-74c65564b8-m5cld	1/1	Running	3 (115s
ago)	2m17s			
argo	argo-server-65566599f8-tv99s	1/1	Running	
0	2m23s			
argo	workflow-controller-77c44779bf-9b42k	1/1	Running	
0	2m23s			
argocd	argocd-application-controller-0	1/1	Running	
0	2m23s			
argocd	argocd-dex-server-76d5bc7dc6-r5rnw	1/1	Running	1 (2m15s
ago)	2m23s			
argocd	argocd-notifications-controller-5ff9495c68-8z58l	1/1	Running	
0	2m23s			
argocd	argocd-redis-857ddfd67b-2lfd2	1/1	Running	
0	2m23s			
argocd	argocd-repo-server-6dcd4856d4-hjv95	1/1	Running	
0	2m23s			
argocd	argocd-server-7cf45b4594-cntd5	1/1	Running	
0	2m23s			

Argo CD and Helm Configuration

This topic lists the Argo components and configurations that are automated as part of the CN2 Pipelines Helm chart install.

- **Argo CD External Service**—Creates a Kubernetes service with service type as NodePort or LoadBalancer. This creates the Argo CD external service that provides access to the Argo CD API server and the Argo CD GUI.

- **Register Git Repository with CN2 Configurations**—Configures repository credentials and connects your Git repository to Argo CD. Argo CD is configured to your Git repository to watch and pull the configuration changes from your Git repository. This Git repository should only contain Kubernetes resources. Argo CD does not understand any other type of YAML or files.
- **Register Kubernetes Clusters**—Registers a Kubernetes cluster to Argo CD. This process configures Argo CD to provision the Kubernetes resources in any Kubernetes cluster. Multiple Kubernetes clusters can be configured in Argo CD.
- **Create an Argo CD Application**—Creates an application using the Argo CD GUI. Any application created in Argo CD needs to be associated with a Git repository and one Kubernetes cluster.

Argo Log In

IN THIS SECTION

- [Access Argo Workflow UI | 16](#)
- [Access Argo CD GUI | 17](#)

After installing the CN2 Pipelines Helm chart, you have access to the Argo Workflow GUI and the Argo CD GUI.

Access Argo Workflow UI

To access the Argo CD GUI, you need connectivity from the management cluster to access the GUI using the NodePort service. The Argo CD GUI is accessed using the management server IP address and port 30550.

1. Access the Argo CD GUI from your browser.

```
https://<management-api-ip>:30550
```

2. On the management node, run the following command to receive the token.

```
kubectl -n argo exec $(kubectl get pod -n argo -l 'app=argo-server' -o
jsonpath='{.items[0].metadata.name}') -- argo auth token
```

Access Argo CD GUI

To access the Argo CD GUI, you need connectivity from the management cluster to access the GUI using the NodePort service. The Argo CD GUI is accessed using the management server IP address and port 30551.

1. Access the Argo CD GUI from your browser.

```
https://<management-api-ip>:30551
```

2. On the management node, run the following command to receive the token. The username is **admin**.

```
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" |
base64 -d
```

CN2 and Workflows

IN THIS SECTION

- [Why Workflows Are Needed | 18](#)
- [How Workflows Work and How CN2 Uses Workflows | 18](#)

Argo Workflows is an open source container-native workflow engine for orchestrating parallel jobs on Kubernetes. You can define workflows where each step in the workflow is a container. You can also model multi-step workflows as a sequence of tasks or capture dependencies between tasks using a directed acyclic graph (DAG).

Why Workflows Are Needed

Workflows are used to invoke and run CN2 test cases after provisioning CN2 resources by using the GitOps engine. These workflows qualify the CN2 application configurations and generates test results for the configuration that is being deployed.

How Workflows Work and How CN2 Uses Workflows

Workflows are triggered whenever a CN2 resource is provisioned by the GitOps engine. Each of the CN2 resources or a group of CN2 resources are mapped to a specific workflow test DAG. After successful completion of these test suites, the CN2 configurations are qualified to be promoted to production environments from the Staging or Test environments.

CN2 Pipelines Service

The pipeline service listens for notifications from Argo Events for any changes in Kubernetes resources. The pipeline service exposes a service which is used by Argo Events to consume and trigger the service with the data related to the CN2 configuration that you applied. It is the responsibility of the CN2 Pipelines service to identify the test workflow to be triggered for the type of CN2 configuration that you applied. Workflows change dynamically depending on the objects being notified. The CN2 Pipelines listener service invokes the respective workflow dependent on the CN2 configuration that gets applied.

CN2 Pipelines Configurations

IN THIS SECTION

- [Pipeline Configuration | 19](#)
- [Test Workflow Template Parameter Configuration | 19](#)
- [Workflow to Kind Map | 19](#)

This topic shows examples for the CN2 Pipelines configurations.

Pipeline Configuration

The pipeline configuration is used by the pipeline engine and includes:

- Pipeline commit threshold
- Config map: cn2pipeline-configs
- Namespace: argo-events

Example CN2 Pipelines configuration:

```
apiVersion: v1
data:
  testcase_trigger_threshold: "10"
kind: ConfigMap
labels:
  app.kubernetes.io/managed-by: Helm
name: cn2pipeline-configs
namespace: argo-events
```

Test Workflow Template Parameter Configuration

All workflow template inputs are stored as configuration maps. These configuration maps are dynamically selected during the execution by the pipeline service.

Workflow to Kind Map

This mapping configuration contains the workflow template to CN2 resource kind mapping. Only one template is selected for execution and the first map that matches has the higher priority. An asterisk (*) in kind: ['*'] indicates that template has higher priority than any other kind matches and overrides every mapping.

A workflow template for a CN2 resource kind mapping template includes:

- Config map: cn2tmpl-to-kind-map
- Namespace: argo-events

Following is an example configuration for the workflow template to CN2 resource kind mapping. Note the asterisk (*) in kind: ['*'] kindmap.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: cn2tmpl-to-kind-map
  namespace: argo-events
data:
  kindmap: |
    - workflow: it-cloud
      kind: ['*']
    - workflow: custom-cnf-sample-test
      kind: ['namespace']
```

Create Custom Workflows for the CN2 Pipelines

You can create custom workflows tests to test your container network functions (CNFs).

To create a custom workflow, you can use the example custom test workflow templates provided with the CN2 Pipelines files. Every workflow has a set of input parameters, volume mounts, container creation, and so on. To understand the workflow template creation see [Argo Workflows](#).

The following example custom test workflow templates are provided:

- Input parameters to workflow
- Mount volumes
- Create Kubernetes resource using workflow (Template name: create-cnf-tf and create-cnf-service-tf)
- Embedded code in workflow (Template name: test-access-tf)
- Pull external code and execute within a container (Template name: test-service-tf)

To automate the inputs to the workflow during the pipeline run, a workflow parameter configuration map is created which has the inputs for the workflow. The configuration map must have the same name as the workflow template.

In the following example, the template name is custom-cnf-sample-test. A configuration map is created automatically with the same name. As a part of the pipeline run, the pipeline service looks for the

configuration map with the template name, gets the inputs, which are then automatically added to the workflow when the pipeline triggers the workflow.

Another update that happens in the test case which triggers the custom workflow is to change the configuration map name to *<cn2tmpl-to-kind-map>*.

```
- workflow: custom-cnf-sample-test
  kind: ['namespace']
```

The following is an example workflow configuration for the template custom-cnf-sample-test:

```
apiVersion: argoproj.io/v1alpha1
kind: WorkflowTemplate           # new type of k8s spec
metadata:
  name: custom-cnf-sample-test  # name of the workflow spec
  namespace: argo-events
spec:
  serviceAccountName: operate-workflow-sa
  entrypoint: cnf-test-workflow # invoke the workflows template
  hostNetwork: true
  arguments:
    parameters:
      - name: image              # the path to a test docker image
        value: not_provided
      - name: kubeconfig_secret  # eg: kubeconfig-989348
        value: not_provided
      - name: report_dir         # eg: /root/SolutionReports
        value: not_provided
  volumes:
    - name: kubeconfig
      secret:
        secretName: "{{ `{{workflow.parameters.kubeconfig_secret}}` }}"
    - name: reportdir
      hostPath:
        path: "{{ `{{workflow.parameters.report_dir}}` }}"
  templates:
    - name: create-cnf-tf
      resource:
        action: apply
        #successCondition: status.succeeded > 0
        #failureCondition: status.failed > 3
        manifest: |
```

```

    apiVersion: v1
    kind: Pod
    metadata:
      name: webapp-cnf
      namespace: argo-events
      labels:
        app.kubernetes.io/name: proxy
    spec:
      containers:
        - name: nginx
          image: {{ .Values.global.docker_image_repo }}/nginx:stable
          ports:
            - containerPort: 80
              name: http-web-svc
- name: create-cnf-service-tf
  resource:
    action: apply
    #successCondition: status.succeeded > 0
    #failureCondition: status.failed > 3
    manifest: |
      apiVersion: v1
      kind: Service
      metadata:
        name: webapp-service
        namespace: argo-events
      spec:
        selector:
          app.kubernetes.io/name: proxy
        ports:
          - name: webapp-http
            protocol: TCP
            port: 80
            targetPort: http-web-svc
- name: test-access-tf
  script:
    image: "{{ `{{workflow.parameters.image}}` }}"
    command: [python]
    source: |
      import time
      print('--Test access to CNF--')
      url = 'webapp-service.argo-events.svc.cluster.local'
      retry_max = 3
      retry_cnt = 0

```

```

        while retry_cnt < retry_max:
            print('Response status code: {}'.format('200'))
            time.sleep(1)
            retry_cnt += 1
            print('Monitoring access count: {}'.format(retry_cnt))
        print('Completed')
- name: test-service-tf
  inputs:
    artifacts:
      - name: pyrunner
        path: /usr/local/src/cn2_py_runner.py
        mode: 0755
      http:
        url: https://raw.githubusercontent.com/roshpr/argotest/main/cn2-experiments/
cn2_py_runner.py
    script:
      image: "{{ workflow.parameters.image }}"
      command: [python]
      args: ["/usr/local/src/cn2_py_runner.py", "4"]
- name: cnf-test-workflow
  dag:
    tasks:
      - name: create-cnf
        template: create-cnf-tf
      - name: create-cnf-service
        template: create-cnf-service-tf
      - name: test-connectivity
        template: test-access-tf
        dependencies: [create-cnf-service]
      - name: test-load
        template: test-service-tf
        dependencies: [create-cnf-service]

```

Explanation of values.yaml

The following table describes the configuration parameters listed in the values.yaml file. You will need the parameter values obtained in ["Before You Install CN2 Pipelines" on page 9](#). See an example values.yaml file in ["Install the CN2 Pipelines Helm Chart" on page 13](#)

Table 1: Parameters for Values.yaml in CN2 Release 23.2

Name	Description	Value	Accepted Values
Global Parameters			
global.registry	Global image registry to pull Juniper artifacts	""	enterprise-hub.juniper.net/contrail-container-prod/
global.docker_image_repo	Global docker registry for non-Juniper images	""	docker.io
global.imagePullSecret	Image pull secret for authenticated registry in base64 format.	""	base64 formatted secret
global.deployment_type	CN2 cluster installed on Kubernetes or OpenShift	""	k8s, openshift
global.managementServer	Management server KubeAPI IP	""	Example: 192.168.1.4
Global GitServer Parameters			
global.gitServer.access_token	GitLab personal access token	""	Example: 1di9sd23cpsadadsaaasd
global.gitServer.gitlabBaseURL	Base server for your GitLab server	""	Example: https://cnf-gitlab.net
global.gitServer.project	Repository or project name in GitLab	""	Example: devops/cn2config
global.gitServer.folderName	Folder name where all the CN2 configurations are located	""	Example: cn2networkconfig

Table 1: Parameters for Values.yaml in CN2 Release 23.2 (Continued)

Name	Description	Value	Accepted Values
global.gitServer.branch	CN2 config Git branch name	""	Example: master
Global cn2ClusterDetails			
global.cn2ClusterDetails.name	CN2 cluster name from kubeconfig file	""	Example: cluster.local
global.cn2ClusterDetails.server	CN2 cluster kubeapi server, which is accessible from the management server	""	Example: 10.1.2.3
global.cn2ClusterDetails.kubeconfig	CN2 kubeconfig name	""	cn2-cluster-kubeconfig
global.cn2ClusterDetails.mountpath	CN2 test profile folder location	""	Example: /opt/cn2
Workflow Object Parameters			
workflow-objects.ssl_enabled	True if SSL enabled. Otherwise, False	False	False, True
Enable the following OpenShift Container Platform (OCP) when deployment_type is "openshift"			
workflow-objects.ocp_api_host_ip	Only for OpenShift Container Platform (OCP) kubeapi IP address	""	Example: 192.168.19.571
workflow-objects.ocp_api_host_name	Only for OpenShift Container Platform (OCP) kubeapi name	""	Example: api.ocp-ss-571.net

2

CHAPTER

Architecture and Design

CN2 Pipelines Solution Test Architecture and Design | 27

CN2 Pipelines Solution Test Architecture and Design

SUMMARY

Learn about Cloud-Native Contrail® Networking™ (CN2) Pipelines architecture and design.

IN THIS SECTION

- [Overview | 27](#)
- [Use Case | 27](#)
- [Test Workflows | 28](#)
- [Profiles | 29](#)
- [Test Environment Configuration | 36](#)
- [Test Execution with Micro-Profiles | 37](#)
- [Logging and Reporting | 38](#)

Overview

Solution Test Automation Framework (STAF) is a common platform developed for automating and maintaining solution use cases mimicking the real-world production scenarios.

- STAF can granularly simulate and control user-personas, actions, timing at scale and thereby exposing the software to all real-world scenarios with long-running traffic.
- STAF architecture can be extended to allow the customer to plug-in GitOps artifacts and create custom test workflows.
- STAF is implemented in Python and pytest test frameworks.

Use Case

STAF emulates Day 0, Day 1, and Day-to-Day operations in a customer environment. Use case tests are performed as a set of test workflows by user-persona. Each user-persona has its own operational scope.

- Operator—Performs global operations, such as cluster setup and maintenance, CN2 deployment, and so on.

- Architect—Performs tenant related operations, such as onboarding, teardown, and so on.
- Site Reliability Engineering (SRE)—Performs operations in the scope of a single tenant only.

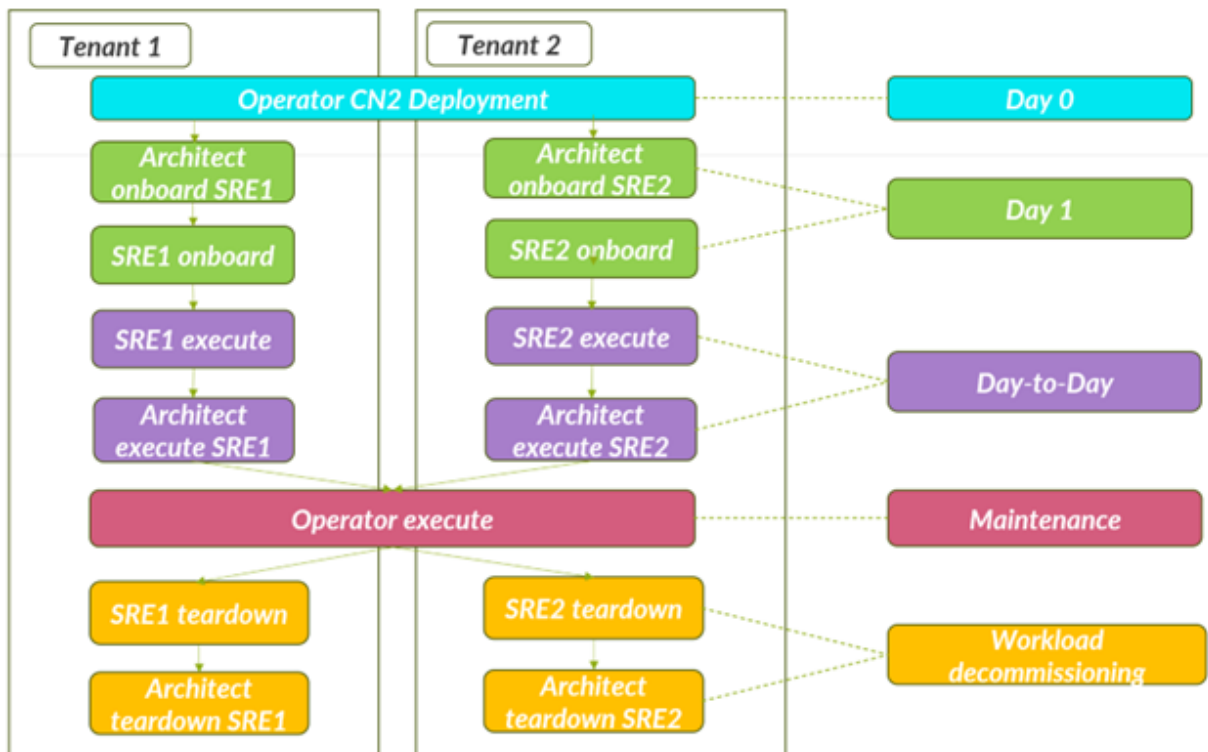
Currently, STAF supports IT Cloud webservice and telco use cases.

Test Workflows

Workflows for each tenant are executed sequentially only. Several tenants' workflows can be executed in parallel, with the exclusion of Operator tests.

Day 0 operation or CN2 deployment is currently independent from test execution. The rest of the workflows are executed as Solution Sanity Tests. In pytest, each workflow is represented by a test suite.

Figure 6: Typical Use Case Scenario



For test descriptions, see ["CN2 Pipelines Test Case Descriptions"](#) on page 53.

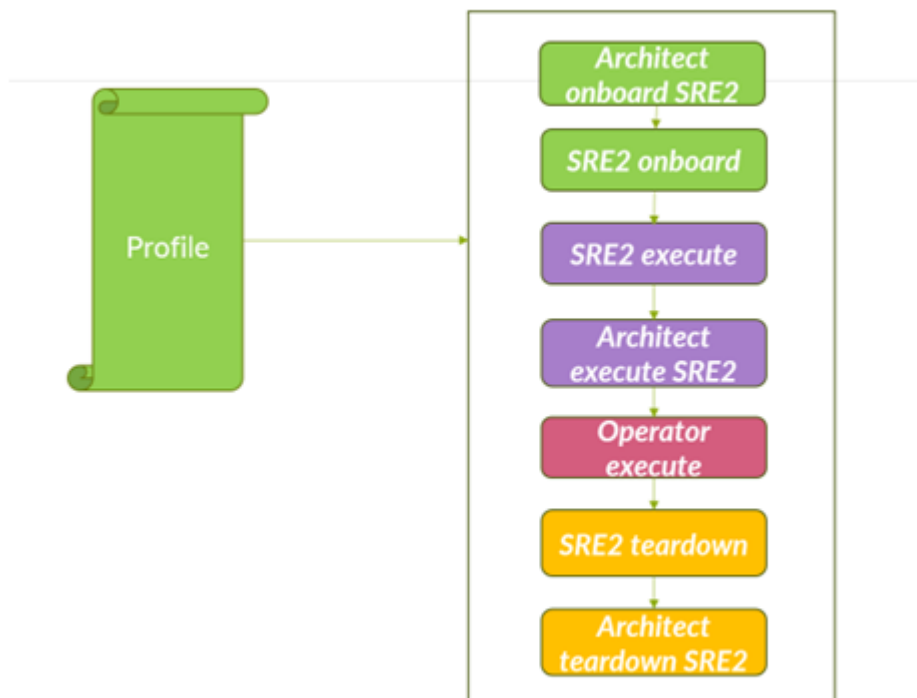
Profiles

IN THIS SECTION

- [Example Profiles | 30](#)

Profile workflows are executed for a use case instance described in a profile YAML file. The profile YAML describes the parameters for namespaces, application layers, network policies, service type, and so on.

Figure 7: Profile Workflow



The profile file is mounted outside of a test container to give you flexibility with choice of scale parameters. For CN2 Pipelines, you can update the total number of pods only.

You can access the complete set of profiles from the downloaded CN2 Pipelines tar file in the folder: **charts/workflow-objects/templates**.

Example Profiles

The following sections have example profiles.

Isolated LoadBalancer Profile

The `IsolatedLoadBalancerProfile.yml` configures a three-tier webservice profile as follows:

- Frontend pods are deployed with a replica count of two (2). These frontend pods are accessed from outside of the cluster through the LoadBalancer service.
- Middleware pods are deployed with a replica count of two (2) and an allowed address pair is configured on both the pods. These pods are accessible through the ClusterIP service from the frontend pods.
- Backend pods are deployed with a replica count of two (2). Backend pods are accessible from middleware pods through the ClusterIP service.
- Policies are created to allow traffic on configured ports on each tier.

`IsolatedLoadbalancerProfile.yml`

```
isl-lb-profile:
  WebService:
    isolated_namespace: True
    count: 1
    frontend:
      external_network: custom
      n_pods: 2
      services:
        - service_type: LoadBalancer
      ports:
        - service_port: 21
          target_port: 21
          protocol: TCP
    middleware:
      n_pods: 2
      aap: active-standby
      services:
        - service_type: ClusterIP
```

```

    ports:
      - service_port: 80
        target_port: 80
        protocol: TCP
  backend:
    n_pods: 2
    services:
      - service_type: ClusterIP
        ports:
          - service_port: 3306
            target_port: 3306
            protocol: UDP

```

Isolated NodePort Profile

The IsolatedNodePortProfile.yml configures a three-tier webservice profile as follows:

- Frontend pods are deployed with a replica count of two (2). These frontend pods are accessed from outside of the cluster using haproxy node port ingress service.
- Middleware pods are deployed with a replica count of two (2) and an allowed address pair is configured on both the pods. These pods are accessible through the ClusterIP service from the frontend pods.
- Backend pods are deployed with a replica count of two (2). Backend pods are accessible from middleware pods through the ClusterIP service.
- Policies are created to allow traffic on configured ports on each tier. Isolated namespace is enabled in this profile.

IsolatedNodePortProfile.yml

```

isl-np-web-profile-w-haproxy-ingress:
  WebService:
    count: 1
    isolated_namespace: True
    frontend:
      n_pods: 2
      anti_affinity: true
      liveness_probe: HTTP
      ingress: haproxy_nodeport
      services:
        - service_type: NodePort

```

```

    ports:
      - service_port: 443
        target_port: 443
        protocol: TCP
      - service_port: 80
        target_port: 80
        protocol: TCP
  middleware:
    n_pods: 2
    liveness_probe: command
    services:
      - service_type: ClusterIP
        ports:
          - service_port: 80
            target_port: 80
            protocol: TCP
  backend:
    n_pods: 2
    services:
      - service_type: ClusterIP
        ports:
          - service_port: 3306
            target_port: 3306
            protocol: UDP

```

Multi-Namespace Contour Ingress LoadBalancer Profile

The MultiNamespaceContourIngressLB.yml configures a three-tier webservice profile as follows:

- Frontend pods are launched using deployment with a replica count of two (2). These frontend pods are accessed from outside of the cluster using haproxy node port ingress service.
- Middleware pods are launched using deployment with a replica count of two (2) and an allowed address pair is configured on both the pods. These pods are accessible through the ClusterIP service from the frontend pods.
- Backend pods are deployed with a replica count of two (2). Backend pods are accessible from middleware pods through the ClusterIP service.
- Policies are created to allow traffic on configured ports on each tier. Isolated namespace is enabled in this profile.

MultiNamespaceContourIngressLB.yml

```
multi-ns-contour-ingress-profile:
  WebService:
    isolated_namespace: True
    multiple_namespace: True
    fabric_forwarding: True
    count: 1
    frontend:
      n_pods: 2
      ingress: contour_loadbalancer
      services:
        - service_type: ClusterIP
          ports:
            - service_port: 6443
              target_port: 6443
              protocol: TCP
    middleware:
      n_pods: 2
      services:
        - service_type: ClusterIP
          ports:
            - service_port: 80
              target_port: 80
              protocol: TCP
    backend:
      n_pods: 2
      services:
        - service_type: ClusterIP
          ports:
            - service_port: 3306
              target_port: 3306
              protocol: UDP
```

Multi-Namespaced Isolated LoadBalancer Profile

The MultiNamespacedIsolatedLB.yml profile configures a three-tier webservice profile as follows:

- Frontend pods are deployed with a replica count of two (2). These frontend pods are accessed from outside of the cluster using a LoadBalancer service.

- Middleware pods are deployed with a replica count of two (2) and an allowed address pair is configured on both the pods. These middleware pods are accessible through the ClusterIP service from the frontend pods.
- Backend pods are deployed with a replica count of two (2). Backend pods are accessible from middleware pods through the ClusterIP service.
- Policies are created to allow traffic on configured ports on each tier. Isolated namespace is enabled in this profile in addition to multiple namespace for frontend, middleware, and backend deployments.

MultiNamespaceIsolatedLB.yml

```
multi-ns-lb-profile:
  WebService:
    isolated_namespace: True
    multiple_namespace: True
    count: 1
  frontend:
    n_pods: 2
    services:
      - service_type: LoadBalancer
    ports:
      - service_port: 443
        target_port: 443
        protocol: TCP
      - service_port: 6443
        target_port: 6443
        protocol: TCP
  middleware:
    n_pods: 2
    aap: active-standby
    services:
      - service_type: ClusterIP
    ports:
      - service_port: 80
        target_port: 80
        protocol: TCP
  backend:
    n_pods: 2
    services:
      - service_type: ClusterIP
    ports:
      - service_port: 3306
```

```
target_port: 3306
protocol: UDP
```

Non-Isolated Nginx Ingress LoadBalancer Profile

The NonIsolatedNginxIngressLB.yml profile configures a three-tier webservice profile as follows:

- Frontend pods are deployed with a replica count of two (2). These frontend pods are accessed from outside of the cluster using a NGINX ingress LoadBalancer service.
- Middleware pods are deployed with a replica count of two (2) and allowed address pair is configured on both the pods. These middleware pods are accessible through the ClusterIP service from the frontend pods.
- Backend pods are deployed with a replica count of two (2). Backend pods are accessible from middleware pods through the ClusterIP service.
- Policies are created to allow traffic on configured ports on each tier.

NonIsolatedNginxIngressLB .yml

```
non-isl-nginx-ingress-lb-profile:
  WebService:
    isolated_namespace: False
    count: 1
    frontend:
      ingress: nginx_loadbalancer
      n_pods: 2
      liveness_probe: HTTP
      services:
        - service_type: ClusterIP
          ports:
            - service_port: 80
              target_port: 80
              protocol: TCP
    middleware:
      n_pods: 2
      services:
        - service_type: ClusterIP
          ports:
            - service_port: 443
              target_port: 443
              protocol: TCP
```

```

backend:
  n_pods: 2
  is_deployment: False
  liveness_probe: command
  services:
  - service_type: ClusterIP
    ports:
      - service_port: 3306
        target_port: 3306
        protocol: UDP

```

Test Environment Configuration

IN THIS SECTION

- [Configuration File | 36](#)
- [Kubeconfig File | 37](#)

Starting in CN2 Release 23.2, the test environment requires that the pods in the Argo cluster have reachability to the network on which CN2 is deployed.

Configuration File

A test configuration file is a file in YAML format which describes a test execution environment.

Starting with CN2 Release 23.2:

- the test configuration file is provided either as an input parameter for Argo Workflow or read from the ConfigMap.
- The test environment is automatically configured when the ConfigMap is updated, deploying the test configuration file which contains parameters describing the test execution environment for Kubernetes or OpenShift.
- CN2 cluster nodes are discovered automatically during test execution.

Kubeconfig File

The kubeconfig file data is used for authentication. The kubeconfig file is stored as a secret on the Argo host Kubernetes cluster.

Enter the following data in the kubeconfig file:

- Server: Secret key must point to either the server IP address or host name.
- For Kubernetes setups, point to the master node IP address: server:

```
https://xx.xx.xx.xx:6443
```

- For OpenShift setups, point to the OpenShift Container Platform API server, extension, and server:

```
https://api.ocp.xxxx.com:6443
```

- Client certificate: Kubernetes client certificate.

Test Execution with Micro-Profiles

A micro-profile is a logical subset of tests from a standard workflow profile. Tests are executed using micro-profile kind markers.

How these markers work:

- In pytest, for the SRE execute and Architect execute test suites, each test case has markers to indicate the applicable profile to use, as well as the Kubernetes and CN2 resource kind mapping.
- The mapping profile kind marker is automatically chosen by the `trigger_pytest.py` script.
- Only tests that match the marker kind and profile requirements are executed. All applicable profiles are triggered in parallel by Argo Workflow.

All profiles are triggered from Argo Workflow, then test(s) execution is decided for each of the steps in the profile.

- If no tests for a kind marker are found in any profile, then all tests are executed.

Figure 8: Example Pytest Markers with Applicable Profiles

```
@pytest.mark.globalvrouterconfig
# applicable profiles
@pytest.mark.isolatednodeportprofile
@pytest.mark.isolatedloadbalancerprofile
@pytest.mark.nonisolatednginxingresslb
@pytest.mark.multinamespacecontouringresslb
@pytest.mark.multinamespaceisolatedlb
def test_validate_link_local_service(self, record_property, request, set_test_case_tr_properties):
    """
```

Logging and Reporting

Two types of log files are created during each test run:

- Pytest session log file—One per session
- Test suite log file—One per test suite

Default file size is 50 MB. Log file rotation is supported.

3

CHAPTER

Test Cases

CN2 Pipelines Test Management | 40

CN2 Pipelines Test Cases | 44

CN2 Pipelines Test Case Descriptions | 53

CN2 Pipelines Test Management

SUMMARY

This section covers the tasks that are specific to managing the test cases.

IN THIS SECTION

- [Trigger the CN2 Pipelines Test | 40](#)
- [Support for Queuing Test Execution | 40](#)
- [Change Commit Threshold Trigger | 41](#)
- [Change Test Profiles | 41](#)
- [Identify CN2 Pipelines Test Trigger | 42](#)
- [Access Test Results | 43](#)
- [Uninstall the CN2 Pipelines | 43](#)

Trigger the CN2 Pipelines Test

Any valid commit automatically triggers the CN2 Pipelines test. For the commit to be valid, you need to do the following:

1. Apply the commit to the correct branch mentioned in the values.yaml.
2. Verify that Argo CD is in sync state. You can verify by logging into the Argo GUI and checking the application status. See ["Argo Log In" on page 16](#).
3. Verify the commit will cross the threshold trigger.

You can verify the threshold trigger setting in the configuration map named cn2-pipeline-configs.

Support for Queuing Test Execution

CN2 Pipelines creates a test workflow execution queue when a commit threshold is reached, while previously triggered test workflows are in Running phase. As soon as a workflow is completed, the next test workflow execution is triggered. CN2 Pipelines has a push mechanism from a test workflow to detect the test completion.

1. Any successful commit (which gets successfully deployed by ArgoCD) triggers the CN2 test workflow.

2. The commit should cross the threshold set in the ConfigMap `cn2pipeline-configs` parameter `testcase_trigger_threshold`.
3. If the commit fails to apply configuration using Argo CD, that commitID will not get registered to ConfigMap.
4. The ConfigMap has data about the commitID with timestamp, test to run based on kind, and status of the test.

Change Commit Threshold Trigger

Use this procedure to change the commit threshold trigger.

1. To access the ConfigMap named `cn2-pipeline-configs`, run this command:

```
kubectl get cm -n argo-events cn2pipeline-configs -o yaml
```

Output:

```
apiVersion: v1
data:
  argocd_check_retry: "5"
  argocd_retry_interval: "120"
  testcase_trigger_threshold: "0"
```

2. Change the `testcase_trigger_threshold` value for the number of commits you want to ignore. By default, this is set to 0.

Change Test Profiles

Use this procedure to change test profiles. This mapping configuration contains the workflow template to CN2 resource kind mapping. Only one template is selected for execution and the first map that matches has the higher priority. An asterisk (*) in kind: `[*]` has higher priority than any other kind matches and overrides every mapping.

1. To access the configuration map named `cn2tmpl-to-kind-map`, run this command:

```
kubectl get cm -n argo-events cn2tmpl-to-kind-map -o yaml
```

Output:

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: cn2tmpl-to-kind-map
  namespace: argo-events
data:
  kindmap: |
    - workflow: it-cloud-arch
      kind: ['virtualRouter','subnet']
    - workflow: it-cloud
      kind: ['*']
    - workflow: it-cloud-arch-sre
      kind: ['namespace']
    - workflow: custom-cnf-sample-test
      kind: ['namespace']
```

CN2 Pipelines will trigger the test where the `kind:['*']` value exists. By default, the profile `it-cloud` is triggered.

2. To trigger another profile, change profile kind to `kind:['*']`.

NOTE: Only one workflow can have an asterisk (*) as the `kind:` value.

Identify CN2 Pipelines Test Trigger

After installing CN2 Pipelines, every valid commit in a given branch and folder will trigger the CN2 test. There is a commit processing workflow triggered after every commit that starts from the resource asterisk (*), for example where the `kind:['*']` value exists. This processes the commit and validates the commit against the threshold value and Argo CD synchronize.

- The commit processing workflow looks like this:

resource-workflow-5547v-3345460271	0/2	Completed	0	11h
------------------------------------	-----	-----------	---	-----

- The actual test workflow starts from CN2 and looks like this:

cn2-test-workflow-zfhtf-4049010958	0/2	Completed	0	13h
------------------------------------	-----	-----------	---	-----

Access Test Results

You can access the test results in HTML format from the GUI.

- To access test results, enter the following in your browser:

```
https://<management_api_ip>:30552
```

Uninstall the CN2 Pipelines

Use this procedure to uninstall and delete the CN2 Pipelines.

- To uninstall CN2 Pipelines, run this command:

```
helm uninstall cn2-pipeline
```

Output:

```
kubectl patch -n argo-events eventsource/gitlab -p '{"metadata":{"finalizers":[]}}' --type=merge
kubectl patch -n argo-events sensor/gitlab -p '{"metadata":{"finalizers":[]}}' --type=merge
kubectl patch -n argo-events eventbus/default -p '{"metadata":{"finalizers":[]}}' --type=merge
kubectl patch -n argocd applications.argoproj.io/cn2networkconfig -p '{"metadata":{"finalizers":[]}}' --type=merge
```

CN2 Pipelines Test Cases

The following table lists the test suites in the profiles, the associated test cases, and object markers. Each test case has markers to indicate which object test is applicable.

Table 2: CN2 Pipelines Tests

Profile	Test Case Steps	Object Markers
IsolatedNodePortProfile		
Test Suite	Test Case	
architect1_onboard_sre1	create_namespaces	
sre1_onboard	onboard_services	
sre1_execute	modify_liveness_probe	pytest.mark.virtualmachineinterface pytest.mark.pod pytest.mark.deployment
sre1_execute	update_cluster_ip_service	pytest.mark.floatingip pytest.mark.service
sre1_execute	update_nodeport_service	pytest.mark.floatingip pytest.mark.service
sre1_execute	update_ingress_network_policy	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy

Table 2: CN2 Pipelines Tests (Continued)

Profile	Test Case Steps	Object Markers
sre1_execute	update_egress_network_policy	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
sre1_execute	update_network_policy_policy_types	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
sre1_execute	update_ingress_service	pytest.mark.floatingip pytest.mark.service
architect1_execute_sre1	validate_link_local_service	pytest.mark.globalvrouterconfig
sre1_tearardown	teardown_services	
architect1_tearardown_sre1	teardown_namespaces	
IsolatedLoadbalancerProfile		
Test Suite	Test Case	
architect1_onboard_sre2	create_namespaces	
sre2_onboard	onboard_services	
sre2_execute	update_cluster_ip_service	pytest.mark.floatingip pytest.mark.service

Table 2: CN2 Pipelines Tests *(Continued)*

Profile	Test Case Steps	Object Markers
sre2_execute	update_service_type	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
sre2_execute	update_ingress_network_policy	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
sre2_execute	update_egress_network_policy	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
sre2_execute	update_network_policy_policy_types	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
sre2_execute	update_loadbalancer_service_general_properties	pytest.mark.floatingip pytest.mark.service pytest.mark.mx_required
sre2_execute	validate_allowed_address_pair_failover	pytest.mark.virtualnetwork pytest.mark.virtualmachineinterface pytest.mark.pod pytest.mark.deployment

Table 2: CN2 Pipelines Tests (Continued)

Profile	Test Case Steps	Object Markers
sre2_execute	validate_allowed_address_pair_update	pytest.mark.virtualnetwork pytest.mark.virtualmachineinterface pytest.mark.pod pytest.mark.deployment
sre2_execute	update_lb_service_static_public_vn	pytest.mark.virtualnetwork pytest.mark.virtualmachineinterface pytest.mark.floatingip pytest.mark.pod pytest.mark.deployment pytest.mark.service pytest.mark.mx_required
sre2_execute	update_label_of_public_network	pytest.mark.virtualnetwork pytest.mark.virtualmachineinterface pytest.mark.floatingip pytest.mark.service pytest.mark.pod pytest.mark.deployment pytest.mark.mx_required
architect1_execute_sre2	validate_link_local_service	pytest.mark.globalvrouterconfig
sre2_tearardown	teardown_services	
architect1_tearardown_sre2	teardown_namespaces	
NonIsolatedNginxIngressLB		

Table 2: CN2 Pipelines Tests *(Continued)*

Profile	Test Case Steps	Object Markers
Test Suite	Test Case	
architect2_onboard_sre3	create_namespaces	
sre3_onboard	onboard_services	
sre3_execute	modify_liveness_probe	pytest.mark.virtualmachineinterface pytest.mark.pod pytest.mark.deployment
sre3_execute	update_cluster_ip_service	pytest.mark.floatingip pytest.mark.service
sre3_execute	update_ingress_network_policy	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
sre3_execute	update_egress_network_policy	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
sre3_execute	update_network_policy_policy_types	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy

Table 2: CN2 Pipelines Tests (Continued)

Profile	Test Case Steps	Object Markers
sre3_execute	update_ingress_service	pytest.mark.floatingip pytest.mark.service
sre3_execute	update_label_of_pods	pytest.mark.applicationpolicyset pytest.mark.virtualmachineinterfac e pytest.mark.pod pytest.mark.deployment pytest.mark.networkpolicy
architect2_execute_sre3	validate_link_local_service	pytest.mark.globalvrouterconfig
sre3_tearardown	teardown_services	
architect2_tearardown_sre3	teardown_namespaces	
MultiNamespaceContourIngressLB		
Test Suite	Test Case	
sre4_onboard	onboard_services	
sre4_execute	update_cluster_ip_service	pytest.mark.floatingip pytest.mark.service
sre4_execute	update_ingress_network_policy	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy

Table 2: CN2 Pipelines Tests *(Continued)*

Profile	Test Case Steps	Object Markers
sre4_execute	update_egress_network_policy	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
sre4_execute	update_network_policy_policy_types	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
architect2_execute_sre4	validate_link_local_service	pytest.mark.globalvrouterconfig
architect2_execute_sre4	update_namespace_label	pytest.mark.virtualnetworkrouter pytest.mark.namespace
architect2_execute_sre4	validate_mesh_virtual_network_router	pytest.mark.virtualnetworkrouter pytest.mark.mx_required
architect2_execute_sre4	validate_hub_spoke_virtual_network_router	pytest.mark.virtualnetworkrouter pytest.mark.mx_required
architect2_execute_sre4	update_forwarding_mode_on_namespace	pytest.mark.virtualnetworkrouter
architect2_onboard_sre4	create_namespaces	
sre4_tearardown	teardown_services	
architect2_tearardown_sre4	teardown_namespaces	

Table 2: CN2 Pipelines Tests *(Continued)*

Profile	Test Case Steps	Object Markers
MultiNamespacesIsolatedLB		
Test Suite	Test Case	
architect2_onboard_sre5	create_namespaces	
sre5_onboard	onboard_services	
architect2_execute, 1, sre5	validate_link_local_service	pytest.mark.globalvrouterconfig
architect2_execute_sre5	update_namespace_label	pytest.mark.virtualnetworkrouter pytest.mark.namespace
architect2_execute_sre5	validate_mesh_virtual_network_router	pytest.mark.virtualnetworkrouter pytest.mark.mx_required
architect2_execute_sre5	validate_hub_spoke_virtual_network_router	pytest.mark.virtualnetworkrouter pytest.mark.mx_required
architect2_execute_sre5	update_forwarding_mode_on_namespace	pytest.mark.virtualnetworkrouter
architect2_execute_sre5	update_fabric_forwarding_on_external_vn	pytest.mark.virtualnetwork pytest.mark.mx_required
sre5_execute	update_cluster_ip_service	pytest.mark.floatingip pytest.mark.service

Table 2: CN2 Pipelines Tests *(Continued)*

Profile	Test Case Steps	Object Markers
sre5_execute	update_service_type	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
sre5_execute	update_ingress_network_policy	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
sre5_execute	update_egress_network_policy	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
sre5_execute	update_network_policy_policy_types	pytest.mark.floatingip pytest.mark.applicationpolicyset pytest.mark.service pytest.mark.networkpolicy
sre5_execute	update_loadbalancer_service_general_properties	pytest.mark.floatingip pytest.mark.service pytest.mark.mx_required
sre5_execute	validate_allowed_address_pair_failover	pytest.mark.virtualnetwork pytest.mark.virtualmachineinterface pytest.mark.pod pytest.mark.deployment

Table 2: CN2 Pipelines Tests *(Continued)*

Profile	Test Case Steps	Object Markers
sre5_execute	validate_allowed_address_pair_update	pytest.mark.virtualnetwork pytest.mark.virtualmachineinterface pytest.mark.pod pytest.mark.deployment
architect2_execute_sre5	update_lb_service_static_public_vn	pytest.mark.virtualnetwork pytest.mark.virtualmachineinterface pytest.mark.floatingip pytest.mark.pod pytest.mark.deployment pytest.mark.service pytest.mark.mx_required
sre5_tearardown	teardown_services	
architect2_tearardown_sre5	teardown_namespaces	

CN2 Pipelines Test Case Descriptions

SUMMARY

This section provides the complete list of solution test cases with descriptions of what each test case is performing.

IN THIS SECTION

- [Architect Onboard | 54](#)
- [Architect Execute | 54](#)
- [Architect Teardown | 55](#)
- [SRE Onboard | 55](#)

●	SRE Execute 56
●	SRE teardown 58

For more details about the test operations in a customer environment, see ["CN2 Pipelines Solution Test Architecture and Design" on page 27](#).

Architect Onboard

Architect onboard is a Day 1 operation and includes the following test case.

test_create_namespaces: : all profiles

For each profile, this test creates a webservice instance and all of the required namespaces.

Architect Execute

Architect execute is a Day-to-Day operation and includes the following test cases.

test_validate_link_local_service

This test updates the GlobalVrouterConfig (GVC) with the localhost service, other multiple services, and one service with multiple fabric IP addresses. This test validates the link local service. When test is complete, original link local service set is restored.

test_validate_mesh_virtual_network_router

This test removes all imports from all virtual network routers (VNR)s, so that routes are not advertised. Then all the other VNRs are added to each other's VNRs. This creates a mesh of all three layers—frontend, middleware, and backend. The test updates the NetworkPolicy, VNR type, and VN label selector. When test is complete, reset the VNRs as specified by the profile.

test_update_namespace_label

This test updates namespace label from name=ns1 to name=ns2. The corresponding VirtualNetworkRouter objects and namespaceSelector are updated. When test is complete, original settings are restored.

test_validate_hub_spoke_virtual_network_router

This test converts middleware layer to a hub VNR. Also, converts frontend and backend layers to spoke VNRs. Next the test validates updates to the hub VNR, backend VNR, meta label, and import statements. When test is complete, the configurations are reset to the baseline profile defined.

test_update_forwarding_mode_on_namespace

This test validates forwarding mode on namespace by setting forwarding mode to ip-fabric and fabric-snat. When test is complete, forwarding mode is reset to original mode.

test_update_fabric_forwarding_on_external_vn

This test enables fabric forwarding on the external virtual network (VN). When fabric forwarding is validated, fabric forwarding is reset on the original external VN.

Architect Teardown

Architect teardown is a maintenance and workload decommissioning operation. Includes the following test case.

test_teardown_namespaces: all profiles

This test tears down the namespaces for each profile in the use case profiles.

SRE Onboard

Site Reliability Engineering (SRE) onboard is a Day 1 operation and includes the following test case.

test_onboard_services: all profiles

For each profile in the use case profiles, create a number of instances and sets up deployment, services, and traffic generator for each instance. This test validates onboarding services for all profiles.

SRE Execute

Site Reliability Engineering (SRE) execute is a Day-to-Day operation and includes the following test cases.

test_modify_liveness_probe

This test validates the HTTP liveness probe failure and Exec liveness probe failure.

test_update_cluster_ip_service

This test creates new middleware pods as a replica of existing ones. The Selector and NetworkPolicy are updated with additional ports. Target port, service port, and service mappings are updated and tested. The ClusterIP service is validated for updates. When test is complete, the newly created set of middleware pods are deleted and the ClusterIP selector is reset.

test_update_nodeport_service

This test creates new frontend pods as a replica of existing ones. The Selector and NetworkPolicy are updated with additional ports. Target port, service port, and service mappings are updated and tested. The NodePort service is validated for updates. When test is complete, the newly created set of middleware pods are deleted and the NodePort selector is reset.

test_update_service_type

Update service type from LoadBalancer to NodePort. This test validates updating the service type. When test is complete, service type is reset back to LoadBalancer.

test_update_ingress_network_policy

This test changes the NetworkPolicy ingress rules from match on podLabel to match on namespaceLabel. All traffic is denied, then allowed. A new NetworkPolicy is added with an exception rule, such as exception for specific IP address. Rule includes three filters—namespaceSelector, podSelector, and AddressBlock. This test validates updating the ingress network policy. When test is complete, rules are reset to the rules specified by the profile.

test_update_egress_network_policy

This test changes the NetworkPolicy egress rules from match on ip_block to match on namespaceLabel. All traffic is denied, then allowed. A new NetworkPolicy is added with an exception rule, such as an exception for a specific IP address. The rule includes three filters—namespaceSelector, podSelector, and

AddressBlock. The test validates updating the egress network policy. When test is complete, rules are reset to the rules specified by the profile.

test_update_network_policy_policy_types

This test modifies PolicyTypes to deny all incoming, deny all outgoing, then deny both incoming and outgoing traffic. This test validates updating the network policy PolicyTypes. When test is complete, rules are reset to the rules specified by the profile.

test_update_loadbalancer_service_general_properties

This test creates new frontend pods as a replica of existing ones. The Selector is updated with new labels and the NetworkPolicy is updated with additional ports. Target port, service port, session affinity, external policy, service mappings are updated for testing. When the test of updates to the load balancer general properties is complete, the newly created set of frontend pods are deleted and Selectors are reset to previously existing labels as a test case teardown.

test_validate_allowed_address_pair_failover

This test triggers Virtual Router Redundancy Protocol (VRRP) master switchover. The test configures Allowed Address Pairs (AAP) mode as active-active. This test validates allowed address pair failover. When complete, AAP and VRRP are reset to original settings.

test_validate_allowed_address_pair_update

This test updates AAP IP from x to y and updates AAP IP to have multiple addresses. This test validates allowed address pair updates.

test_update_lb_service_static_public_vn

This test creates a VN (`new-public-vn`) under the service namespace with a custom public RT1 assigned. Also, configures Juniper Networks® MX Series 5G Universal Routing Platform (MX) with routing instance and route targets with the same VN properties. A new LoadBalancer service is created and updated with the ExternalIP (both IPv4 and IPv6). This test then validates traffic, validates service, and validates updating the namespace annotation. When test is complete, the namespace annotations are reset to the original value. All services created during the test are deleted.

test_update_ingress_service

This test creates an additional service for the frontend pods, then updates the service selector label of ingress backend from old service to a new service. Multiple service paths are included in the ingress

specification. After the test validates the update to the ingress service, the newly created ingress service is deleted. The original ingress service is restored.

test_update_label_of_public_network

This test updates the custom public networks label from `local==public-test` to `local==unselect-public-vn`. Existing service should not be affected. Next the test creates a new LoadBalancer service and resets the label to `local==public-test`. The new LoadBalancer service gets a public IP address and is accessible from the Internet endpoint. This tests updating the public networks label and the new service is deleted.

test_update_label_of_pods

This test updates the pod label, corresponding service, and NetworkPolicy selectors. After the test is complete, the pod label, corresponding service, and NetworkPolicy selectors are reset to original settings.

SRE teardown

Site Reliability Engineering (SRE) teardown is a maintenance and workload decommissioning operation. This phase tears down and deletes the SRE objects created during the onboard phase.

test_teardown_services

This test deletes SRE objects created during the SRE onboard phase.