Pods and Services Reference

Feature Summary and Revision History

Summary Data

Table 1: Summary Data

Applicable Product(s) or Functional Area	cnSGW-C
Applicable Platform(s)	SMI
Feature Default Setting	Enabled - Always-on
Related Documentation	Not Applicable

Revision History

Revision Details	**Release**
First introduced	2020.07

Feature Description

cnSGW-C is built on the Kubernetes cluster strategy, adopting the native concepts of containerization, high availability, scalability, modularity, and ease of deployment. cnSGW-C uses the components, such as pods and services offered by Kubernetes.

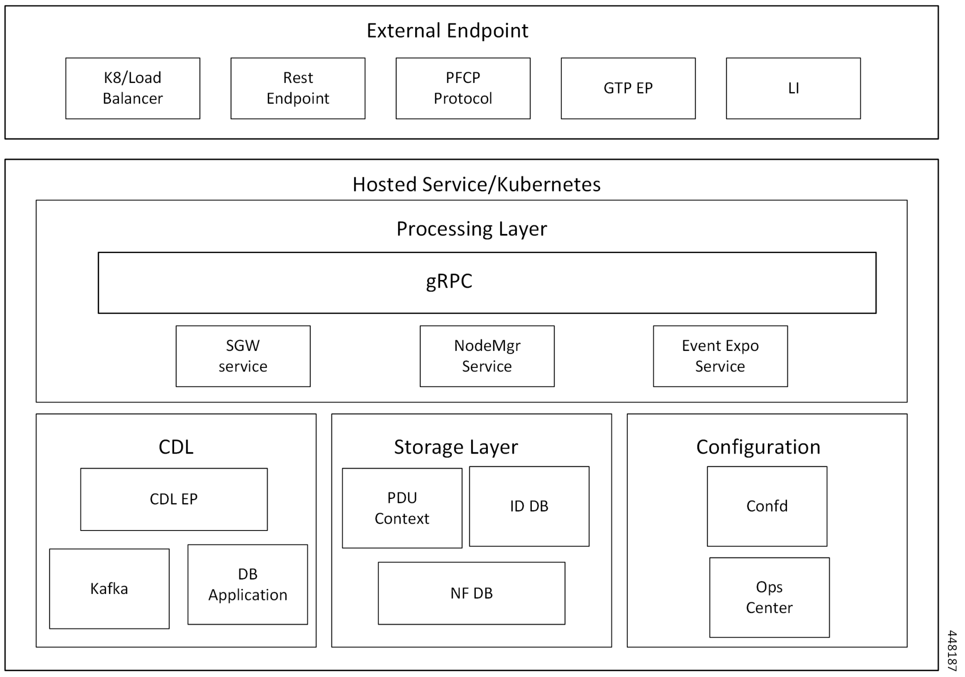
Depending on your deployment environment, the cnSGW-C deploys the pods on the configured virtual machines(VM) that you have configured. Pods operate through the services that are responsible for the intrapod communications. If the machine hosting the pods fail or experiences network disruption, the pods are terminated or deleted. However, this situation is transient and k8s, create new pods to replace the invalid pods.

The following workflow provides high-level information about:

- Host machines
- Associated pods and services
- Interaction among pods

The representation might defer based on your deployment infrastructure.

Figure 1: Communication Workflow of Pods



Kubernetes deployment includes the kubectl command-line tool to manage the Kubernetes resources in the cluster. You can manage the pods, nodes, and services.

For generic information on the Kubernetes concepts, see the Kubernetes documentation.

Pods

A pod is a process that runs on Kubernetes cluster. Pod encapsulates a granular unit known as a container. A pod can contains one or more containers.

Kubernetes deploys one or multiple pods on a single node which can be a physical or a virtual machine. Each pod has a discrete identity with an internal IP address and port number. The containers within the pod shares the storage and network resources.

The following tables list the cnSGW-C and Common Execution Environment (CEE) pod names and the hosts on which they are deployed depending on the labels that you assign. See the following table for information on how to assign the labels.

Table 2: cnSGW-C Pods

Pod Name	**Description**	**Host Name**
api-sgw-ops-center	Functions as *confD* API pod for the cnSGW-C Ops Center.	OAM
base-entitlement-sgw	Operates to support smart licensing feature.	OAM **Note** Currently not suppor
bgpspeaker	Operates to support dynamic routing for L3 route management and BFD monitoring.	Protocol
cache-pod	Operates to support cache system information that is used by other pods as applicable.	Protocol
cdl-ep-session-c1	Provides an interface to the CDL.	Session
cdl-index-session-c1	Preserves the mapping of keys to the session pods.	Session
cdl-slot-session-c1	Operates as the CDL session pod to store the session data.	Session
documentation	Contains the documentation.	OAM
etcd-sgw-etcd-cluster	Hosts the etcd for the cnSGW-C OAM application to store information such as pod instances, leader information, endpoints.	OAM
georeplication	Operates to support cache, ETCD replication across sites, and site role management.	Protocol
grafana-dashboard-app-infra	Contains the default dashboard of app-infra metrics in Grafana.	OAM
grafana-dashboard-cdl	Contains the default dashboard of CDL metrics in Grafana.	OAM
grafana-dashboard-sgw	Contains the default dashboard of cnSGW-C service metrics in Grafana.	OAM
gtpc-ep-n0	Operates as GTPC endpoint of cnSGW-C.	Protocol
kafka	Hosts the Kafka details for the CDL replication.	Protocol
li-ep-n0	Operates as Lawful Intercept endpoint of cnSGW-C.	Protocol
oam-pod	Operates as the pod to facilitate Ops Center actions, such as show commands, configuration commands, monitor protocol monitor subscriber.	OAM

Pod Name	Description	Host Name
ops-center-sgw-ops-center	Acts as the cnSGW-C Ops Center.	OAM
smart-agent-sgw-ops-center	Operates as the utility pod for the cnSGW-C Ops Center.	OAM
nodemgr-n0	Performs node level interactions, such as Sxa link establishment and management (heartbeat). It generates unique identifiers, such as UE IP address and SEID.	Service
protocol-n0	Operates as encoder and decoder of application protocols (PFCP, whose underlying transport protocol is UDP).	Protocol
rest-ep-n0	cnSGW-C uses REST-EP as Notification client.	Protocol
service-n0	Contains main business logic of cnSGW-C.	Service
udp-proxy	Operates as proxy for all UDP messages. Owns UDP client and server functionalities.	Protocol
swift-sgw-ops-center	Operates as the utility pod for the cnSGW-C Ops Center.	OAM
zookeeper	Assists Kafka for topology management.	OAM

CEE Pods

For details, see the "CEE pods" topic from the UCC Common Execution Environment - Configuration and Administration Guide.

UDP Proxy Pod

Feature Description

The cnSGW-C has UDP interfaces towards the UP (Sxa), MME (S11), and PGW (S5 or S8). With the help of the protocol layer pods, the messages are encoded, decoded, and exchanged on these UDP interfaces.

For achieving the functionalities mentioned on the 3GPP specifications:

- It is mandatory for the protocol layer pods to receive the original source and destination IP address and port number. But the original IP and UDP header is not preserved when the incoming packets arrive at the UDP service in the Kubernetes (K8s) cluster.
- Similarly, for the outgoing messages, the source IP set to the external IP address of the UDP service (published to the peer node) is mandatory. But the source IP is selected as per the egress interface when different instances of protocol layer pods send outgoing messages from different nodes of the K8s cluster.

The protocol layer pod spawns on the node, which has the physical interface configured with the external IP address to achieve the conditions mentioned earlier. However, spawning the protocol layer pods has the following consequences:

- It is not possible to achieve the node level HA (High Availability) as the protocol pods are spawned on the same node of the K8s cluster. Any failure to that node may result in loss of service.
- The protocol pods must include their own UDP client and server functionalities. In addition, each protocol layer pod may require labeling of the K8s nodes with the affinity rules. This restricts the scaling requirements of the protocol layer pods.

The cnSGW-C addresses these issues with the introduction of a new K8s pod called udp-proxy. The primary objectives of this pod are:

- The udp-proxy pod acts as a proxy for all kinds of UDP messages. It also owns the UDP client and server functionalities.
- The protocol pods perform the individual protocol (PFCP, GTP, Radius) encoding and decoding, and provide the UDP payload to the udp-proxy pod. The udp-proxy pod sends the UDP payload out after it receives the payload from the protocol pods.
- The udp-proxy pod opens the UDP sockets on a virtual IP (VIP) instead of a physical IP. This ensures that the udp-proxy pod does not have any strict affinity to a specific K8s node (VM), thus enabling node level HA for the UDP proxy.

Note One instance of the udp-proxy pod is spawned by default in all the worker nodes in the K8s cluster.

The UDP proxy for cnSGW-C feature has functional relationship with the Virtual IP Address feature.

Architecture

The udp-proxy pod is placed in the worker nodes in the K8s cluster.

1. Each of the K8s worker node contains one instance of the udp-proxy pod. However, only one of the K8s worker node owns the virtual IP at any time. The worker node that owns the virtual IP remains in the active mode while all the other worker nodes remain in the standby mode.
2. The active udp-proxy pod binds to the virtual IP and the designated ports for listening to the UDP messages from the peer nodes (UPF and SGW).
3. The UDP payload received from the peer nodes are forwarded to one instance of the protocol, gtp-ep, or radius-ep pods. The payload is forwarded either on the same node or different node for further processing.
4. The response message from the protocol, gtp-ep, or radius-ep pods is forwarded back to the active instance of the udp-proxy pod. The udp-proxy pod sends the response message back to the corresponding peer nodes.
5. The cnSGW-C-initiated messages are encoded at the protocol, gtp-ep, or radius-ep pods. In addition, the UDP payload is sent to the udp-proxy pod. Eventually, the udp-proxy pod comprises of the complete IP payload and sends the message to the peer. When the response from the peer is received, the UDP payload is sent back to the same protocol pod from which the message originated.

Protocol Pod Selection for Peer-Initiated Messages

When the udp-proxy pod receives the peer node (for instance UPF) initiated messages, it is load-balanced across the protocol instances to select any instance of the protocol pod. An entry of this instance number is stored along with the source IP and source port number of the peer node. This ensures that the messages form the same source IP and source port are sent to the same instance that was selected earlier.

High Availability for the UDP Proxy

The UDP proxy's HA model is based on the keepalived virtual IP concepts. A VIP is designated to the N4 interface during the deployment. Also, a keepalived instance manages the VIP and ensures that the IP address of the VIP is created as the secondary address of an interface in one of the worker nodes of the K8s cluster.

The udp-proxy instance on this worker node binds to the VIP and assumes the role of the active udp-proxy pod. All udp-proxy instances in the other worker nodes remain in the standby mode.

Services

The cnSGW-C configuration is composed of several microservices that run on a set of discrete pods. These Microservices are deployed during the cnSGW-C deployment. cnSGW-C uses these services to enable communication between the pods. When interacting with another pod, the service identifies the pod's IP address to initiate the transaction and acts as an endpoint for the pod.

The following table describes the cnSGW-C services and the pod on which they run.

Table 3: cnSGW-C Services and Pods

Service Name	Pod Name	Description
base-entitlement-sgw	base-entitlement-sgw	Operates to support sma
bgpspeaker-pod	bgpspeaker	Operates to support dyn route management and I
datastore-ep-session	cdl-ep-session-c1	Responsible for the CDI
datastore-notification-ep	smf-rest-ep	Responsible for sending the CDL to the *sgw-servi* **Note** cnSGW-C uses REST-E client.
datastore-tls-ep-session	cdl-ep-session-c1	Responsible for the secu
documentation	documentation	Responsible for the cnS
etcd	etcd-sgw-etcd-cluster-0, etcd-sgw-etcd-cluster-1, etcd-sgw-etcd-cluster-2	Responsible for pod dis namespace.
etcd-sgw-etcd-cluster-0	etcd-sgw-etcd-cluster-0	Responsible for synchro the *etcd* cluster.
etcd-sgw-etcd-cluster-1	etcd-sgw-etcd-cluster-1	Responsible for synchro the *etcd* cluster.
etcd-sgw-etcd-cluster-2	etcd-sgw-etcd-cluster-2	Responsible for synchro the *etcd* cluster.

Service Name	Pod Name	Description
grafana-dashboard-app-infra	grafana-dashboard-app-infra	Responsible for the app-infra metrics in
grafana-dashboard-cdl	grafana-dashboard-cdl	Responsible for the metrics in Grafana.
grafana-dashboard-sgw	grafana-dashboard-sgw	Responsible for the cnSGW-C service m
gtpc-ep	gtpc-ep-n0	Responsible for inter GTP-C pod.
helm-api-sgw-ops-center	api-sgw-ops-center	Manages the Ops Ce
kafka	kafka	Processes the Kafka
li-ep	li-ep-n0	Responsible for law
local-ldap-proxy-sgw-ops-center	ops-center-sgw-ops-center	Responsible for leve credentials by other
oam-pod	oam-pod	Responsible to facili Ops Center.
ops-center-sgw-ops-center	ops-center-sgw-ops-center	Manages the cnSGW
ops-center-sgw-ops-center-expose-cli	ops-center-sgw-ops-center	To access cnSGW-C IP address.
smart-agent-sgw-ops-center	smart-agent-sgw-ops-center	Responsible for the c
smf-nodemgr	smf-nodemgr	Responsible for inter *smf-nodemgr* pod.
smf-protocol	smf-protocol	Responsible for inter smf-protocol pod.
sgw-service	sgw-service	Responsible for inter cnSGW-C service p
swift-sgw-ops-center	swift	Operates as the utili Ops Center.
zookeeper	zookeeper	Assists Kafka for to
zookeeper-service	zookeeper	Assists Kafka for to

Open Ports and Services

The cnSGW-C uses different ports for communication. The following table describes the default open ports and the associated services.

Table 4: Open Ports and Services

Port	Type	Service	Usage
22	tcp	SSH	SMI uses TCP port to communicate with the virtual machines.

Port	Type	Service	Usage
53	tcp	domain	DNS port.
80	tcp	HTTP	SMI uses TCP port for providing Web access to CLI, Documentation, and TAC.
111	tcp	rpcbind	Open Network Computing Remote Procedure Call.
179	tcp	bgp	Border Gateway Protocol (BGP)
443	tcp	SSL/HTTP	SMI uses TCP port for providing Web access to CLI, Documentation, and TAC.
2379	tcp	etcd-client	CoreOS etcd client communication.
6443	tcp	http	SMI uses port to communicate with the Kubernetes API server.
7472	tcp	unknown	speaker, used by Grafana.
8083	tcp	us-srv	Kafka connects REST interface.
8850	tcp	unknown	udp-proxy
8879	tcp	unknown	udp-proxy
9100	tcp	jetdirect	SMI uses TCP port to communicate with the Node Exporter. Node Exporter is a Prometheus exporter for hardware and OS metrics with pluggable metric collectors. It allows you to measure various machine resources, such as memory, disk, and CPU utilization.
10250	tcp	SSL/HTTP	SMI uses TCP port to communicate with Kubelet. Kubelet is the lowest level component in Kubernetes. It is responsible for what is running on an individual machine. It is a process watcher or supervisor focused on active container. It ensures the specified containers are up and running.
10251	tcp	-	SMI uses TCP port to interact with the Kube scheduler. Kube scheduler is the default scheduler for Kubernetes and runs as part of the control plane. A scheduler watches for newly created pods that have no node assigned. For every pod that the scheduler discovers, the scheduler becomes responsible for finding the best node for that pod to run on.
10252	tcp	apollo-relay	SMI uses this TCP port to interact with the Kube controller. The Kubernetes controller manager is a daemon that embeds the core control loops shipped with Kubernetes. The controller is a control loop that watches the shared state of the cluster through the API server and makes changes to move the current state to the desired state.

Port	Type	Service	Usage
10256	-	HTTP	SMI uses TCP port to interact with the Kube proxy. Kube proxy is a network proxy that runs on each node in your cluster. Kube proxy maintains network rules on nodes. These network rules allow network communication to your pods from network sessions inside or outside of your cluster.
50051	tcp	unknown	gRPC service listen port.
53	udp	domain ISC BIND (Fake version: 9.11.3- 1ubuntu1.9- Ubuntu)	DNS port
111	udp	rpcbin	Open Network Computing Remote Procedure Call
2123	udp	gtpc	GTP control
8805	udp	pfcp	Packet Forwarding Control Protocol (PFCP)

Associating Pods to the Nodes

This section describes how to associate a pod to the node.

After configuring a cluster, you can associate the pods to the nodes through labels. This association enables the pods to get deployed on the appropriate node, based on the key-value pair.

Labels are required for the pods to identify the nodes where they must be deployed and to run the services. For example, when you configure the protocol-layer label with the required key-value pair, the pods are deployed on the nodes that match the key-value pair.

1. To associate pods to the nodes through the labels, use the following configuration:

```
config
 k8
   label
     cdl-layer
       key key_value
       value value
     oam-layer
       key key_value
       value value
     protocol-layer
       key key_value
       value value
     service-layer
       key key_value
```

```
value value
end
```

NOTES:

- If you don't configure the labels, cnSGW-C assumes the labels with the default key-value pair.
  - **label { cdl-layer { key** *key_value* **| value** *value* **}**—Configures the key value pair for CDL.
  - **oam-layer { key** *key_value* **| value** *value* **}**—Configures the key value pair for OAM layer.
  - **protocol-layer { key** *key_value* **| value** *value* **}**—Configures the key value pair for protocol layer.
  - **service-layer { key** *key_value* **| value** *value* **}**—Configures the key value pair for the service layer.

Viewing the Pod Details and Status

If the service requires additional pods, cnSGW-C creates and deploys the pods. You can view the list of available pods in your deployment through the cnSGW-C Ops Center.

You can run the kubectl command from the master node to manage the Kubernetes resources.

Pod Details

1. To view the comprehensive pod details, use the following command.

   ```
   kubectl get pods -n sgw pod_name -o yaml
   ```

The output of this command provides the pod details in YAML format with the following information:

- The IP address of the host where the pod is deployed.
- The service and the application that is running on the pod.
- The ID and the name of the container within the pod.
- The IP address of the pod.
- The present state and phase of the pod.
- The start time from which pod is in the present state.

Use the following command to view the summary of the pod details.

```
kubectl get pods -n sgw_namespace -o wide
```

States

The following table describes the state of a pod.

Table 5: Pod States

State	Description
Running	The pod is healthy and deployed on a node. It contains one or more containers.
Pending	The application is in the process of creating the container images for the pod.
Succeeded	Indicates that all the containers in the pod are successfully terminated. These pods cann't be restarted.
Failed	One ore more containers in the pod have failed the termination process. The failure occurred as the container either exited with non-zero status or the system terminated the container.

Runtime pod scales for server additions

Table 6: Feature History

Feature Name	Release Information	Description
Scaling Converged Core Cluster with Node Labeling and Required Affinity	2025.03.0	This feature enables the dynamic addition of servers to a running cluster without disrupting existing deployments. It allows seamless scalability to meet growing demands and minimizes downtime, enhancing operational continuity.

The runtime pod scales feature manages scaling efficiently when a new server is added.

The key features and capabilities include:

- Enables dynamic recalibration of resources during new server addition.
- Utilizes new resources at runtime in a standalone converged core cluster.
- Supports Kubernetes version 1.33.

Key advantages of the pod scaling

The pod scaling feature has key advantages:

- Dynamic server addition.
- Usage of internal Kubernetes labels and affinity rules for workload management.
- Efficient load distribution.

Addition and scaling of runtime servers in converged core process

When a new server is added to the Kubernetes cluster using the cluster manager, ensure that the replica count for the “smf service” and “sgw-service” endpoints is increased, resulting in application pods being scheduled on the new node.

Complete these actions:

- Updating the Helm charts for both smf-service and sgw-service pods to use the required affinity rules.
- Labeling the new node.
- Increasing replica counts in the endpoint configuration.
- Removing the label and performing any other necessary cleanup

Prerequisite

The prerequisites for this feature are:

- Decommission the CF or SF Node from VPC-DI.
- Clean up the Virtual Drive of this CF or SF node.
- Connect the node towards the 5G cluster by physically attaching PCIe slots.
- Ensure the SMI CNDP version is the July 2025 release version.
- Ensure that the OpsCenter is set to the "large-all" deployment model before proceeding.
- Run Full Cluster Sync to get this node added to the cluster.
- Set the node count for the endpoint service and sgw-service to 0 or 1.

Configure new deployment

supports a new deployment model "large-all". By default, "large" model is the default value used on existing deployments.

Use these Day 1 configuration steps for new deployment.

1. Enter the configuration mode.

   `configure`

2. Specify the deployment model using the **deployment model** *deployment-model* command.

   Example

   ```
   [dev-perf/smf] smf# config
   [dev-perf/smf] smf(config)# deployment model large-all
   [dev-perf/smf] smf(config)# commit
   ```

3. Enter **commit** to commit the changes.

Pod scaling processes

The process includes these steps:

1. Log into the Ops Center using the **cn-ops-center** command, to label the new node.

Enter **cluster label node** *node_name* **pod-list** *pod-list* to label the node for specific pod scaling.

Example

```
cluster label node dev-perf-worker-4 pod-list {sgw-service|smf-service}
```

2. Enter into the configuration mode and increase the replica count under the endpoint service (for) or increase the replica count for sgw-service.
3. Wait for the system status to be 100%.
4. Clear the node label for specific pod scaling using the **cluster unlabel node** *node_name* **pod-list** *pod-list*

   example

   ```
   cluster unlabel node dev-perf-worker-4 pod-list {sgw-service|smf-service}
   ```

5. Verify the node labels status using the **show cluster label** command.

```
dev-perf/smf] smf# show cluster label
NAME               STATUS
--------------------------------------------
dev-perf-master-1
dev-perf-master-2
dev-perf-master-3
dev-perf-worker-1
dev-perf-worker-2
dev-perf-worker-3
dev-perf-worker-4  smf-service,sgw-service
```