

Pods and Services Reference

- Feature Summary and Revision History, on page 1
- Feature Description, on page 2
- Associating Pods to the Nodes, on page 12
- Viewing the Pod Details and Status, on page 13
- Multi-server Control Plane (CP) scaling, on page 14
- GTPC Protocol Endpoint Merge with UDP Proxy Bypass, on page 26
- ULB traffic routing architecture, on page 26
- UDP Proxy Functionality Merge into Protocol Micro-services, on page 27
- Runtime pod scales for server additions, on page 30
- All-in-One deployment models , on page 32

Feature Summary and Revision History

Summary Data

Table 1: Summary Data

Applicable Products or Functional Area	SMF
Applicable Platform(s)	SMI
Feature Default Setting	Enabled – Always-on
Related Changes in this Release	Not Applicable
Related Documentation	Not Applicable

Revision History

Table 2: Revision History

Revision Details	Release
The node-monitor pod is supported to enable monitoring of all K8 pods.	2021.02.0
The grafana-dashboard-app-infra pod is removed.	2021.02.3.t3
First introduced.	Pre-2020.02.0

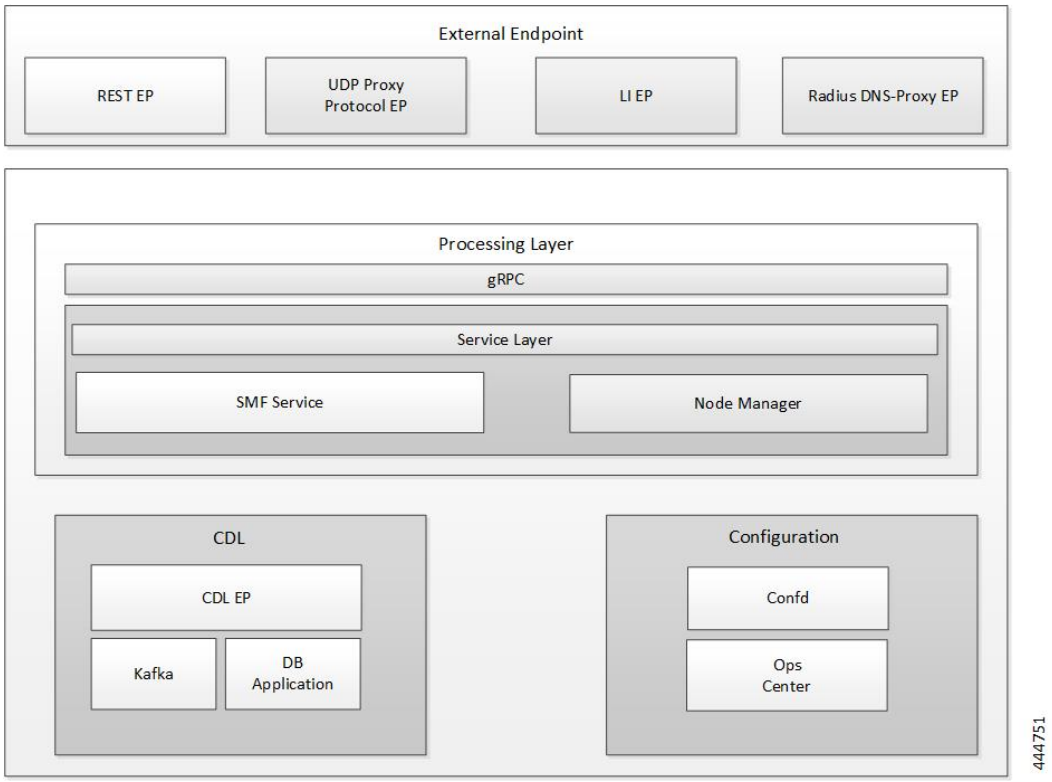
Feature Description

The SMF is built on the Kubernetes cluster strategy, which implies that it has adopted the native concepts of containerization, high availability, scalability, modularity, and ease of deployment. To achieve the benefits offered by Kubernetes, SMF uses the construct that includes the components, such as pods and services.

Depending on your deployment environment, the SMF deploys the pods on the virtual machines that you have configured. Pods operate through the services that are responsible for the intrapod communications. If the machine hosting the pods fails or experiences network disruption, the pods are terminated or deleted. However, this situation is transient and Kubernetes spins new pods to replace the invalid pods.

The following workflow provides a high-level visibility into the host machines, and the associated pods and services. It also represents how the pods communicate with each other. The representation may differ 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 information on the Kubernetes concepts, see the Kubernetes documentation.

For more information on the Kubernetes components in SMF, see the following:

- Pods
- Services

Pods

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

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

The following table lists the SMF pod names and the hosts on which they are deployed depending on the labels that you assign. For information on how to assign the labels, see Associating Pods to the Nodes.

Table 3: SMF Pods

Pod Name	Description	Virtual Machine Name
api-smf-ops-center	Functions as the *confD* API pod for the SMF Ops Center.	OAM
base-entitlement-smf	Supports Smart Licensing feature.	OAM
bgpspeaker	Dynamic routing for L3 route management and BFD monitoring	Protocol
cache-pod	Operates as the pod to cache any sort of system information that will be used by other pods as applicable.	Protocol
cdl-ep-session	Provides an interface to the CDL.	Session
cdl-index-session	Preserves the mapping of keys to the session pods.	Session
cdl-slot-session	Operates as the CDL session pod to store the session data.	Session
dns-proxy	Operates as DNS endpoint of SMF	Protocol
diameter-ep-gx-client	Operates as a Diameter endpoint to enable communication between SMFand PCRF through Gx interface	Protocol
diameter-ep-gy-client	Operates as a Diameter endpoint to enable communication between SMF and PCRF through Gy interface	Protocol
documentation	Contains the documentation.	OAM
edr-monitor pod	Contains the EDR files that are maintained in a persistent volume.	OAM
etcd-smf-etcd-cluster	Hosts the etcd for the SMF application to store information, such as pod instances, leader information, NF-UUID, endpoints, and so on.	OAM
georeplication	Responsible for cache, etcd replication across sites, and site role management	Protocol
grafana-dashboard-cdl	Contains the default dashboard of CDL metrics in Grafana.	OAM
grafana-dashboard-smf	Contains the default dashboard of SMF service metrics in Grafana.	OAM
gtpc-ep	Operates as GTPC endpoint of SMF.	Protocol
kafka	Hosts the Kafka details for the CDL replication.	Protocol
li-ep	Operates as Lawful Intercept endpoint of SMF.	Protocol
nodemgr	Performs node level interactions, such as N4 link establishment, management (heart-beat), and so on. Also, generates unique identifiers, such as UE IP address, SEID, CHF-ID, Resource URI, and so on.	Service

Pod Name	Description	Virtual Machine Name
node-monitor pod	Monitors all the K8 nodes and performs self-reboot on encountering an issue, which in turn triggers an GR to other rack.	NA
oam-pod	Operates as the pod to facilitate Ops Center actions like show commands, configuration commands, monitor protocol monitor subscriber, and so on.	OAM
ops-center-smf-ops-center	Acts as the SMF Ops Center.	OAM
protocol	Operates as encoder and decoder of application protocols (PFCP, GTP, RADIUS, and so on) whose underlying transport protocol is UDP.	Protocol
radius-ep	Operates as RADIUS endpoint of SMF	Protocol
rest-ep	Operates as REST endpoint of SMF for HTTP2 communication.	Protocol
service	Contains main business logic of SMF.	Service
smart-agent-smf-ops-center	Operates as the utility pod for the SMF Ops Center.	OAM
udp-proxy	Operates as proxy for all UDP messages. Owns UDP client and server functionalities.	Protocol
swift-smf-ops-center	Operates as the utility pod for the SMF Ops Center.	OAM
zookeeper	Assists Kafka for topology management.	OAM

For details on UDP proxy, see the UDP Proxy Pod, on page 6 section.

These SMF pods communicate with the Common Execution Environment (CEE) pods. For the complete list of CEE pods, see the *UCC CEE Configuration and Administration Guide*.

Replicas

Each pod runs on a single instance of an application. To provide more resources by running more instances, you can use multiple Pods, one for each instance. This concept in Kubernetes is referred to as replication. Replicated Pods or replicas are usually created and managed as a group by a workload resource and its controller.

With multiple replicas, Kubernetes can distribute the load between them. During node failures, replicas can be used.

Note Replicas are based on the hardware and deployed call model.

UDP Proxy Pod

Feature Description

The SMF has UDP interfaces toward the UPF (N4) and SGW (s5 or s8 for EPS interworking). With the help of the protocol layer pods (smf-protocol and gtp-ep), the messages are encoded and 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) because 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 (smf-protocol, gtp-ep, and radius-ep) 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 SMF 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 SMF 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 SMF-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 smf-protocol, gtp-ep, or radius-ep 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 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 other worker nodes remain in the standby mode.

Node Monitoring Pod

Node monitor pod runs on all Kubernetes nodes, such as master and worker nodes to periodically check the operating state of other nodes. The nodes might become unreachable due to network issue or hardware transient issue.

Depending on the operating state of nodes, the node monitor pod switches to different modes. SMF uses **nodemonitor** CLI command in Global Configuration mode to switch between the modes.

The following configuration section provides more information on the command and modes.

Configure Node Monitoring Pod

The node monitoring pod switches to different modes to resolve the hardware transient issues.

To switch between the modes, use the following sample configuration:

```
config
   nodemonitor mode { 0 | 1 | 2 | 3 interval wait_time }
   end
```

NOTES:

- **nodemonitor mode { 0 | 1 | 2 | 3 interval** *wait_time* **}**
  - **mode 0**—Disables the node monitoring functionality.
  - **mode 1**—Enables the node monitoring and performs self-reboot only after reaching a hardcoded value of 2 seconds when two or more nodes are not reachable. This is the default setting.
  - **mode 2**—Enables the node monitoring and performs self-reboot when two or more nodes are not reachable but not all the nodes.
  - **mode 3 interval** *wait_time*—Specify the time interval in seconds, after which the node monitoring pod is rebooted when two or more nodes are not reachable.

    wait_time must be an integer in the range of 5–300.

Configuration Example

The following is an example of node monitoring pod configuration.

```
config
   nodemonitor mode 3 interval 15
   end
```

As per this example, the node monitoring pod waits for 15 seconds and then performs self-reboot when two or nodes are not reachable.

Configuration Verification

To verify the configuration, use the **show running-config nodemonitor** command.

The output of this show command displays the configuration related to mode of the node monitoring pod.

```
smf# show running-config nodemonitor
nodemonitor mode 3 interval 15
```

Services

The SMF configuration consists of several microservices that run on a set of discrete pods. Microservices are deployed during the SMF deployment. SMF 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 SMF services and the pod on which they run.

Table 4: SMF Services and Pods

Service Name	Pod Name	Description
base-entitlement-smf	base-entitlement-smf	Supports Smart Licensing feature.
bgpspeaker-pod	bgpspeaker	Dynamic routing for L3 route management and BFD monitoring
datastore-ep-session	cdl-ep-session	Responsible for the CDL session.
datastore-notification-ep	smf-rest-ep	Responsible for sending the notifications from t CDL to the *smf-service* through *smf-rest-ep*.

Service Name	Pod Name	Description
datastore-tls-ep-session	cdl-ep-session	Responsible for the secure CDL connection.
diameter-ep-gx-client	diameter-ep-gx-client	Responsible for sending gRPC messages to Diameter endpoint for Credit Control messa which are converted to Diameter CCR mess by Diameter endpoint and sent to the Gx ser
diameter-ep-gy-client	diameter-ep-gy-client	Responsible for sending gRPC messages to Diameter endpoint for Credit Control Messa which are converted to Diameter CCR mess by Diameter endpoint and sent to the Gy ser
documentation	documentation	Responsible for the SMF documents.
edr-monitor	edr-monitor	Responsible for maintaning EDR files in pers volume
etcd	etcd-smf-etcd-cluster-0, etcd-smf-etcd-cluster-1, etcd-smf-etcd-cluster-2	Responsible for pod discovery within the namespace.
etcd-smf-etcd-cluster-0	etcd-smf-etcd-cluster-0	Responsible for synchronization of data amor *etcd* cluster.
etcd-smf-etcd-cluster-1	etcd-smf-etcd-cluster-1	Responsible for synchronization of data amor *etcd* cluster.
etcd-smf-etcd-cluster-2	etcd-smf-etcd-cluster-2	Responsible for synchronization of data amor *etcd* cluster.
grafana-dashboard-app-infra	grafana-dashboard-app-infra	Responsible for the default dashboard of app metrics in Grafana.
grafana-dashboard-cdl	grafana-dashboard-cdl	Responsible for the default dashboard of CD metrics in Grafana.
grafana-dashboard-smf	grafana-dashboard-smf	Responsible for the default dashboard of SMF-service metrics in Grafana.
gtpc-ep	gtpc-ep	Responsible for inter-pod communication w GTP-C pod.
helm-api-smf-ops-center	api-smf-ops-center	Manages the Ops Center API.
kafka	kafka	Processes the Kafka messages.
li-ep	li-ep	Responsible for lawful-intercept interactions
local-ldap-proxy-smf-ops-center	ops-center-smf-ops-center	Responsible for leveraging Ops Center crede by other applications like Grafana.
oam-pod	oam-pod	Responsible to facilitate Exec commands on Ops Center.
ops-center-smf-ops-center	ops-center-smf-ops-center	Manages the SMF Ops Center.
ops-center-smf-ops-center-expose-cli	ops-center-smf-ops-center	To access SMF Ops Center with external IP address.

Service Name	Pod Name	Description
smart-agent-smf-ops-center	smart-agent-smf-ops-center	Responsible for the SMF Ops Center API.
smf-sbi-service	smf-rest-ep	Responsible for routing incoming HTTP2 messag to REST-EP pods.
smf-n10-service	smf-rest-ep	Responsible for routing incoming N10 message to REST-EP pods.
smf-n11-service	smf-rest-ep	Responsible for routing incoming N11 message to REST-EP pods.
smf-n40-service	smf-rest-ep	Responsible for routing incoming N40 message to REST-EP pods.
smf-n7-service	smf-rest-ep	Responsible for routing incoming N7 messages REST-EP pods.
smf-nrf-service	smf-rest-ep	Responsible for routing incoming NRF messag to REST-EP pod.
smf-nodemgr	smf-nodemgr	Responsible for inter-pod communication with *smf-nodemgr* pod.
smf-protocol	smf-protocol	Responsible for inter-pod communication with smf-protocol pod
smf-radius-dns	smf-radius-dns	Responsible for inter-pod communication with smf-radius-dns pod
smf-rest-ep	smf-rest-ep	Responsible for inter-pod communication with smf-rest-ep pod
smf-service	smf-service	Responsible for inter-pod communication with smf-service pod
swift-smf-ops-center	swift	Operates as the utility pod for the SMF Ops Cent
zookeeper	zookeeper	Assists Kafka for topology management
zookeeper-service	zookeeper	Assists Kafka for topology management

Open Ports and Services

The SMF uses different ports for communication purposes. The following table describes the default open ports and the associated services.

Table 5: Open Ports and Services

Port	Service	Usage
9003	gRPC	This is the default gRPC port for all the application pods that aren't using the host network.
8080	TCP/HTTP	This is the default prometheus server port for all the application pods that aren't using the host network.
8090	TCP/HTTP	SMF uses the HTTP2 REST endpoint for the SBI interface endpoint.

Port	Service	Usage
2024	SSH	SMF Ops Center uses this port to provide the ConfD CLI access.
2379-2380	TCP	SMF uses the etcd server client endpoint to store and retrieve system data.
8882	gRPC	This is the CDL session DB endpoint used by all other application pods.
8890	gRPC	Application REST endpoint uses this port to receive the call-back timer or notification from the CDL.
8883	gRPC	This is the TLS - CDL session DB endpoint used by all other application pods.
2123	UDP	SMF uses the 3GPP standard GTP-C port for the signaling of GTP-C protocol messages (control path).
2152	UDP	SMF uses the 3GPP standard GTP-U port for the signaling of GTP-U protocol messages (data path).
8805	UDP	SMF uses the 3GPP standard PFCP port for the signaling of PFCP protocol messages.
8765	TCP6	SMF uses this port as the readiness probe port for the UDP proxy pod.
7179, 7189	TCP/HTTP	Pprof Server endpoint serves runtime profiling data from GTP-C endpoint interface-specific pods, such as S11 and S5.
23300, 23310	TCP/gRPC	SMF uses the TLS GRPC IPC server endpoint for the GTP-C endpoint interface-specific pods.
9005, 9015	TCP/gRPC	Non-TLS GRPC IPC server port for the GTP-C endpoint interface-specific pods.
27000, 27010	TCP	SMF uses the KeepAliveD endpoint for GTP-C endpoint interface-specific pods.
7279, 7289	TCP	This port is the admin endpoint internal port that is used in the GTP-C endpoint interface-specific pods. This port is used for liveness and readiness probe.
7079, 7089	TCP/HTTP	SMF uses the prometheus server port for the GTP-C endpoint interface-specific pods.
8083	TCP/HTTP	SMF uses the prometheus server port for the protocol pod.
9006	TCP/gRPC	SMF uses the non-TLS GRPC IPC Server endpoint for the protocol pod.
8003	TCP/HTTP	SMF uses the prometheus server port for the protocol pod.
23100	TCP/gRPC	SMF uses the TLS GRPC IPC server endpoint for the protocol pod.
7679	TCP/HTTP	The Pprof server endpoint serves runtime profiling data from the protocol pod.
7879	TCP	This port is the admin endpoint internal port that is used in the protocol pod. This port is used for the liveness and readiness probe.
27500	TCP	SMF uses this KeepAliveD endpoint for the protocol pod.
28000	TCP	SMF uses this KeepAliveD endpoint for the UDP proxy pod.
23200	TCP/gRPC	SMF uses the TLS GRPC IPC server endpoint for the UDP proxy pod.

Port	Service	Usage
9004	TCP/gRPC	Non-TLS GRPC IPC server endpoint for the UDP proxy pod.
7879	TCP	This port is the admin endpoint internal port that is used in the UDP proxy pod. This port is used for the liveness and readiness probe.
8850	TCP/HTTP	The Pprof server endpoint serves runtime profiling data from the UDP proxy pod.

In addition to the preceding ports, SMF uses the ports that are destined for SMI for routing information between hosts. For information on SMI ports, see the *Ultra Cloud Core Subscriber Microservices Infrastructure Operations Guide*.

Associating Pods to the Nodes

This section describes how to associate a pod to the node based on their labels.

After you have configured a cluster, you can associate 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 get 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.

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

```
config
  k8 label vm_group key label_key value label_value
  end
```

NOTES:

- **k8 label** *vm_group* **key** *label_key* **value** *label_value*: Configures the K8 node affinity label parameters.
  - *vm_group*: Specify the VM group. It must be one of the following:
    - cdl-layer
    - oam-layer
    - protocol-layer
    - service-layer

Note The **service-layer** label is applicable only to the large deployment model and is not applicable to the large-all deployment model.

  - **key** *label_key*: Specify the label key. *label_key* must be a string.
  - **value** *label_value*: Specify the label value. *label_value* must be a string.

- If you choose not to configure the labels, then SMF assumes the labels with the default key-value pair.

Viewing the Pod Details and Status

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

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

The pod details are available in YAML format.

Use the following sample configuration to view the comprehensive pod details:

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

The output of this command displays the following information:

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

To view all the pods in the SMF namespace, use the following sample configuration:

```
kp get pods -n smf_namespace -o wide
```

States

Understanding the pod's state lets you determine the current health and prevent the potential risks. The following table describes the pod's states.

Table 6: 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 cannot 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.
Unknown	The state of the pod could not be determined. Typically, this could be observed because the node where the pod resides was not reachable.

Multi-server Control Plane (CP) scaling

The multi-server control-plane (CP) scaling support provides a framework to enhance scalability and maintainability in deploying protocol pods across master nodes.

Active-standby pod deployment model for protocol pods

Each protocol pod is deployed with one active instance and two standby instances distributed across all master nodes. This simplifies pod placement and scheduling. Only the active pod consumes CPU and memory resources, while the standby pods remain idle until needed. The SMI keepalive configuration is centralized, with virtual IPs (VIPs) distributed evenly from a single control point. This approach is considered more scalable and maintainable.

Replica counts are set to three for selected pods if configured with fewer than three replicas. Protocol pods are split into Sx and N4 interfaces, and host network pods are deployed on all master nodes. Protocol-layer pods are deployed on all master nodes using Kubernetes protocol-layer labels, while host network pods are configured to run in one active and two standby modes. The active pod is selected by the Kubernetes scheduler based on preferred labels defined in the endpoint configuration. BGP speaker pods run actively on all master nodes, advertising their MED values to influence routing decisions. SMI keepalive priorities are distributed across the three master nodes, and keepalived ensures that the VIP is hosted on the highest-priority node. Initial CPU requests are set to 1 CPU for protocol pods, 0.8 CPU for service pods, and 2 CPUs for CDL pods. The standalone configuration under the Sx interface applies only to host networking, and there is no change in behavior for existing deployment models.

The following table describes the pod layout in the **large-all** deployment model.

Pod	Master-1	Master-2	Master-3	Worker-node	Node and replica configuration
smf-service	-	-	-	-	• **Endpoint Service Node:** Any number of nodes and replicas. Recommended node: 0/1 • **Extended Service:** Any number of nodes and replicas. Recommended node: 0/1
sgw-service	-	-	-	-	• **Endpoint Service Node:** Any number of nodes and replicas. Recommended node: 0/1 • **Extended Service:** Any

					number of nodes and replicas. Recommended node: 0/1
nodemgr	-	-	-	-	Active in any two nodes Node: 2, Replicas: 1 (Default: Node: 1, Replicas: 1)
protocol(Sx&N4)	Stand-By-1	Active	Stand-By-2	NA	Node:3, Replicas:1 (Default: Node:3,Replicas:1)
gtpc-ep(S5)	Active	Stand-By-1	Stand-By-2	NA	Node:3, Replicas:1 (Default: Node:3,Replicas:1)
gtpc-ep(S11)	Stand-By-1	Stand-By-2	Active	NA	Node:3, Replicas:1 (Default: Node:3,Replicas:1)
rest-ep	Active pods	NA	NA	NA	Node:3, Replicas:2
udp-proxy	Stand-By	Stand-By	Active	NA	Node:3, Replicas:1 (Default: Node:3, Replicas:1)
diameter-ep	Stand-By	Stand-By	Active	NA	Node:3, Replicas:1 (Default: Node:3, Replicas:1)
li-ep	Active	Stand-By	Stand-By	NA	Node:3, Replicas:1 (Default: Node:3, Replicas:1)
dns-proxy	Active pods	NA	NA	NA	Node and Replicas: Any value (Recommended: Node:3, Replicas:1)
gtpp-ep	1 Active/1 Stand-By	NA	NA	NA	Hard Coded to Node:2, Replicas:1
radius-ep	Active pods	NA	NA	NA	Node and Replicas: Any value (Recommended: Node:3, Replicas:1)
bgp-speaker	Active	Active	Active	NA	Node: 3, Replicas: 1 (no default)

Verifying the node label

In the multi-server control plane architecture, specific Kubernetes labels are assigned to nodes to ensure that various functional layers (OAM, Protocol, and CDL) are distributed correctly across the cluster. This allows for better resource utilization and high availability by utilizing master nodes for specific protocol and service tasks. You can verify if the nodes are labeled correctly by using the command:

kubectl get nodes --show-labels

Note If labels are missing, the pods will remain in a `Pending` state, as they cannot find their required placement node.

The following labels are used to manage pod placement in this configuration:

- **smi.cisco.com/node-type=oam**: Identifies nodes designated for Operations, Administration, and Maintenance (OAM) functions.
- **smi.cisco.com/proto-layer=protocol**: Identifies nodes that will host protocol-related pods (e.g., Sx, N4, S5, S11).
- **smi.cisco.com/proto-preferred**: Used for deterministic placement of protocol instances across the master nodes.
- **smi.cisco.com/sess-type=cdl-node**: Identifies nodes hosting the Common Data Layer (CDL) components.

Configuration and restrictions

To support a multi-server deployment, specific labels must be assigned to the Kubernetes nodes, and the protocol endpoints must be configured to operate in standalone mode with defined resource limits.

The deployment must use the **large-all** model. Labels are used to designate which nodes handle specific functional layers (OAM, Protocol, or CDL).

```
deployment
  model large-all

instance instance-id 1
   Existing PFCP/GTP configuration
  endpoint pfcp
    interface sxa
      standalone true
      internal-vip xx.1x.xx.1x
      cpu request xxx0
      cpu max-process 1x
      memory request xxx0
      memory limit xxxx0
      preferred-labels key smi.cisco.com/proto-preferred value node1
    exit
    interface n4
      preferred-labels key smi.cisco.com/proto-preferred value node2
    exit
  exit

  Existing Virtual IP configuration
virtual-ips sxi1
  hosts master-1
   priority      100
```

```
   preempt-delay 30
  exit
  hosts master-2
   priority      99
   preempt-delay 0
  exit
  hosts master-3
   priority      98
   preempt-delay 2
  exit
 exit

virtual-ips n4i1
  hosts master-1
    priority 99
  hosts master-2
    priority 100
  hosts master-3
    priority 99
exit
```

The **smf-service** and **sgw-service** pods are prioritized to deploy on worker nodes over master nodes. The following two CLI command options are introduced for both the endpoint service and extended-service StatefulSets:

- **preferred-labels key smi.cisco.com/service-key value service-value**
- **required-labels key smi.cisco.com/service-key value service-value**. (For extended-service, the keys are `service-key1` and `service-value1`.)



Note Only one of these label options should be configured per service. If both are configured, **required-labels** take precedence and **preferred-labels** are ignored. If neither command is configured, the system defaults to the existing behavior where Kubernetes schedules pods across all nodes based on resource availability.

The sample configuration structure illustrates using **preferred-labels** as an example, you can use **required-labels** based on your deploymnent need, but do not configure both simultaneously. Here are some considerations:

- Ensure that the Kubernetes nodes intended for these services are labeled with the corresponding keys (`smi.cisco.com/service-key` and `smi.cisco.com/service-key1`) to ensure the scheduler can correctly identify the target zones.
- If using `required-labels`, verify that nodes with the matching labels are active and have sufficient resources; otherwise, the pods will remain in a `Pending` state.
- These configurations are specific to the `large-all` model and will not impact other deployment types.

```
! New Service Specifications
endpoint service
  replicas 6
  Choose either preferred-labels OR required-labels
```

```
   preferred-labels key smi.cisco.com/service-key value service-value
 exit


extended-service
   replicas 10
   Choose either preferred-labels OR required-labels
   preferred-labels key smi.cisco.com/service-key1 value service-value1
 exit
```

Upgrade and resiliency for multi-server HA deployment in SMF

This section highlights the upgrade and resiliency strategies for multi-server large-all deployment model. It also discusses the importance of In-Service Software Upgrade (ISSU) in the upgrade process for the large-all deployment.

Table 7: Feature history

Feature name	Release information	Description
Upgrade and resiliency for Large-all deployment in SMF	2026.03.0	This feature helps maintain high uptime and session continuity in the large-all deployment model by scaling the control plane across multi-server clusters. It enables the large-all deployment model to have node resiliency with minimal session loss using in-service software upgrades. This feature allows to upgrade or recover nodes with lesser disruption, supporting ongoing service availability at scale.

In-Service Software Upgrade

The large-all deployment model supports In-Service Software Upgrade (ISSU) for both application and SMI software upgrades to ensure continuous service availability with minimal disruption.

The In-Service Software Upgrades (ISSU) and high availability enhancements feature minimizes session loss during application and firmware upgrades, provides node resiliency to ensure system stability and uninterrupted operation during single-server outages, and supports robust redundancy requirements in large-scale high availability deployments.

Application rolling upgrade

The rolling upgrade optimizations feature needs to be enabled to ensure smooth continuation of services during software upgrade process.

Configure rolling upgrade optimizations

This configuration enables rolling upgrade optimizations, such as, retry mechanisms, transient transaction handling etc, during the pod rolling upgrades.

Follow these steps to configure the rolling upgrade optimizations:

Procedure

Step 1 Create an instance of the converged core profile.

Example:

```
config
  profile converged-core cc_profile_name
```

Step 2 Enable the rolling upgrade feature.

Example:

```
config
  profile converged-core cc_profile_name
  supported-features [ rolling-upgrade-all ]
  end
```

SMI rolling upgrade using ISSU

SMI supports two upgrade strategies:

- **Concurrent upgrade:** upgrades all nodes at once.
- **Rolling upgrade:** upgrades nodes one at a time sequentially. It is the default upgrade strategy.

Note It is mandatory to disable **k8s-node-restart-recovery** feature before starting SMI upgrade. It should be enabled after the SMI upgrade is complete.

The rolling update strategy is leveraged to provide ISSU, which ensures minimal service disruption.

Multi-server clusters require robust methods to maintain service availability and minimize session loss during upgrades or failures. High availability and ISSU enhancements support these requirements in cloud-native control-plane deployments.

SMF supports ISSU to keep the cluster operational throughout the SMI upgrade process in the large-all deployment.

Components involved in ISSU

The key components involved in the process are:

- **Cluster operator:** Initiates and manages upgrade workflows and monitors cluster health.
- **Application pods:** Host active services and undergo rolling, controlled, or parallel upgrades.
- **SMI subsystem:** Enables in-service firmware upgrades on each server node.

- **VIP management and keepalived:** Handle endpoint hosting, migration, and node prioritization during upgrades.
- **Pod Disruption Budget (PDB):** Limits how many pods or services may be unavailable during upgrade or failure events.

Table 8: Feature History Table

Feature Name	Release Information	Feature Description
SMF Performance Optimization	Release 2026.04.0	With this release, • the Kubernetes descheduler is temporarily disabled during ISSU upgrade to prevent unnecessary pod disruptions while nodes are restarting. • SMI can send node-level CPU and memory details to the application Ops-Center. This information is required to define the appropriate default values for Intel/AMD nodes with 128 and 160 CPU cores.

Node drain timeout

To ensure service pods PDB is effective during rolling upgrades, SMI cluster sync process is enhanced to support configurable node drain timeout values.

Here are the rolling upgrade configuration options introduced under cluster configuration in the SMI Cluster Deployer

cluster *cluster-name* **configuration rolling-upgrade-settings post-uncordon-wait** *timeout-value-minutes*

cluster *cluster-name* **configuration rolling-upgrade-settings drain-grace-period** *timeout-value-seconds*

cluster *cluster-name* **configuration rolling-upgrade-settings drain-timeout** *timeout-value-seconds*

Table 9: Rolling upgrade configuration options

Setting	Purpose	Type	Default Value
post-uncordon-wait	Maximum time to wait after node uncordon for the stateful pods previously scheduled on that node to return to Running and Ready before proceeding to the next server	timeout in minutes	10 minutes
drain-grace-period	Grace period used during node drain in rolling upgrade	timeout in seconds	25 seconds

Setting	Purpose	Type	Default Value
drain-timeout	Maximum time allowed for the node drain operation to complete during rolling upgrade	timeout in seconds	60 seconds

Here is a sample configuration:

```
config
clusters tmo2-smf
configuration rolling-upgrade-settings drain-timeout 660
configuration cilium enable-multicast false
commit
```

Note It is mandatory to configure node drain timeout to ensure minimal disruption.

K8s node restart recovery

The K8s node restart recovery automatically restores service distribution after a node becomes available again. It aims to provide operators with the necessary configuration and observability details to manage cluster rebalancing effectively.

When a Kubernetes node restarts, service pods are moved to other nodes to maintain availability. By default, these pods do not automatically move back to the original node once it is back online, which can leave the cluster imbalanced.

The K8s Node Restart Recovery automatically detects when a node has recovered and rebalances the services back to that node. It ensures the cluster returns to an optimal state without requiring manual intervention.

A grace period is provided to allow active sessions to finish safely before pods are moved.

K8s Node Restart Recovery is a feature that automatically restores balanced service-pod placement after a Kubernetes node restarts.

In a multi-server deployment, service pods are distributed across available nodes. When a node restarts, pods are evicted and rescheduled onto healthy nodes. When the node returns to service, it often remains empty while other nodes remain overloaded, as Kubernetes does not automatically move pods back to the recovered node. This feature solves this imbalance by using a controlled, paced rebalancing workflow to move pods back to the recovered node, ensuring even pod distribution for critical services like `smf-service` and `sgw-service`.

Information About K8s Node Restart Recovery

The recovery feature is managed by the CEE (Cluster Execution Environment) Operations Center and the CN (Converged Network) Operations Center.

When a node returns to a "Ready" state, the CEE Operations Center detects the event and triggers a `node_restart` alert. The CN Operations Center, configured with the `k8s-node-restart-recovery` feature, consumes this alert. Upon receiving the alert, the internal NodeMgr component initiates a controlled rebalancing workflow. This workflow moves pods one at a time, allowing each to become ready before the next move, which minimizes service impact while correcting the cluster-wide distribution. The process is intelligent and pauses during In-Service Software Upgrades (ISSU) to prevent unnecessary disruption.

Descheduler Requirement

The normal Kubernetes scheduler places new or replacement pods, but it does not rebalance existing pods across a recovered node. NodeMgr therefore drives a controlled rebalancing workflow that moves only the pods needed to restore an even distribution.

The workflow is paced and respects workload disruption protections. It moves protected pods one at a time, allowing each pod to become ready before the next move. This limits service impact while correcting the cluster-wide imbalance.

Ops-center configuration

The recovery feature has two configuration responsibilities. The CEE Operations Center detects when a node returns to service and raises the node-restart alert. The CN Operations Center enables the recovery feature and binds NodeMgr to the alert that it should consume.

Backend handling

The event manager receives the alert through the event service and broadcasts it to the NodeMgr replicas. Each replica validates the feature state and alert data. Only the active leader schedules recovery; a follower does not schedule a second action. Repeated delivery is handled as a duplicate event and must not create duplicate recovery work.

How the descheduler is driven

Recovery requests are processed serially so that two rebalancing operations cannot compete. Before starting, NodeMgr checks that the feature is enabled, the replica is still the leader, the target node is usable, and the cluster is safe to process. If any cluster node is not Ready, a new rebalancing job is deferred.

For each controlled step, NodeMgr targets the recovered node, observes the job result, waits for protected workloads to become ready, and checks whether pod distribution is balanced. It repeats only while progress is needed. If a safety check fails, leadership changes, the feature is disabled, or no progress is possible, it stops and waits for a later valid trigger.

Node Restart during ISSU

During an SMI rolling upgrade (ISSU), cluster nodes restart one at a time. Each restart can normally produce a recovery alert, but running the rebalancing workflow while the upgrade is still in progress would cause unnecessary pod movement and additional disruption.

The Cluster Deployer publishes lifecycle signals when maintenance starts and ends. NodeMgr uses these signals to pause new node-restart recovery for the maintenance period. The state is stored durably and shared so both NodeMgr replicas make the same decision, including after a replica restart. This state is a maintenance record, not a replacement for leader election.

While maintenance is active, restart alerts are not used to start rebalancing. This prevents one recovery operation from being started for every node intentionally restarted as part of ISSU.

CEE Configuration

Use this command for CEE configuration:

```
clusters <cluster-name>
 notifications kubernetes-events enabled
 notifications kubernetes-events target-ops-center <smf name>
 notifications kubernetes-events notify-on-failure true
 exit
```

ISSU sequence

- Before the rolling upgrade starts, the Cluster Deployer publishes the maintenance-start lifecycle event.
- NodeMgr stores the maintenance state. Both replicas use this state to block new recovery work.
- The SMI upgrade restarts nodes one at a time. These restarts can produce node_restart alerts, but the maintenance gate prevents those alerts from starting recovery jobs.
- After the upgrade and cluster synchronization finish, the Cluster Deployer publishes the maintenance-end lifecycle event.
- The deschedular is rub onc , as part of the END event.



Note The SMF/CCG component must be upgraded to the 2026.04.0 release so that its embedded NodeMgr can process *CNDPMaintenanceStarted* and *CNDPMaintenanceEnded* Events.

After the SMF/CCG component is running the 2026.04.0 release, NodeMgr can block new node-restart recovery actions when maintenance starts and allow new recovery actions after maintenance ends.

Metrics

NodeMgr provides the following metrics for SMI maintenance Event handling and node-restart recovery control.

Table 10: Metrics

Metric name	Metric type	Labels	Possible values / meaning
nodemgr_cluster_sync_event_stats	Counter	`reason`	`started` — NodeMgr accepted and persisted `CNDPMaintenanceStarted`. `ended` — NodeMgr accepted and persisted `CNDPMaintenanceEnded`.
nodemgr_cluster_sync_active_operations	Gauge		Current number of active maintenance operations.

Verification

You can verify the maintenance start and end signals, the configured feature and alert binding, the accepted event fields, the leader's recovery decision, the configured recovery controls, the protected workload behavior, and the final pod distribution through the platform's standard events, logs, Jobs, Leases, and workload status.

Benefits of K8s Node Restart Recovery

- **Automatic Rebalancing**: Restores even pod distribution without manual intervention after a node restart.
- **Service Impact Mitigation**: Uses a paced workflow that ensures pods are ready before continuing to the next move.
- **Maintenance Awareness**: Automatically pauses rebalancing activities during rolling upgrades (ISSU) to avoid redundant disruptions.

- **Controlled Operations**: Respects workload disruption budgets and prevents flood-based recovery work through bounded queue protection.

Configure K8 node restart recovery

This configuration allows you to configure the K8 node restart recovery.

The K8 node restart recovery is enabled through standard operator configuration settings. It requires the rules to be added at two ops-center layers, an alert rule in cee-ops-center and a matching activation hook in cn-ops-center.

Procedure

Step 1 Define the Prometheus alert.

```
config
   alerts rules group alert_group_name
   interval-seconds seconds
   rule node_restart
   expression "(changes(kube_node_status_condition{condition=\"Ready\",status=\"true\"}[10m]) >
0) and (kube_node_status_condition{condition=\"Ready\",status=\"true\"} == 1)"
   duration time
   severity major
   type  "Equipment Alarm"
   annotation  summary
      value  "K8s Node={{$labels.node}} - returned to Ready after restart"
```

Example:

```
[cee] cee# show running-config
alerts rules group NodeRestart
 interval-seconds 60
 rule node_restart
  expression "(changes(kube_node_status_condition{condition=\"Ready\",status=\"true\"}[10m]) > 0)
and (kube_node_status_condition{condition=\"Ready\",status=\"true\"} == 1)"
  duration   1m
  severity   minor
  type       "Equipment Alarm"
  annotation summary
   value "K8s Node={{$labels.node}} - returned to Ready after restart"
  exit
 exit
exit
```

Step 2 Enable the recovery feature on the converged-core profile and bind it to the alert name.

```
config
   profile converged-core cc_profile_name
      [ no ] activated-features k8s-node-restart-recovery use-alert node_restart
```

Example:

```
[smf] smf# show running-config profile converged-core
profile converged-core cnvg_core_prfl
```

```
 activated-features k8s-node-restart-recovery use-alert node_restart
exit
```

Keepalived weight timeout and pre-empt delay

This section describes the Keepalived weight timeout and pre-empt delay for the uprade process.

Keepalived weight timeout configuration

Based on the number of proto nodes, this configuration allows to define the weight for the large-all deployment.

For a large-all deployment, the Keepalive weight should be configured as 3.

```
config
clusters tmo2-smf
virtual-ips n4sxi11
check-ports-weight -3
```



Note This configuration is required for all the proto pod VIPs (s11, s5, and N4).

Pre-empt delay configuration

For protocol pods with host networking, preempt-delay should be configured.

```
config
clusters tmo2-smf
virtual-ips gti11s11
check-ports-weight -3
  hosts master-1
   priority      99
   preempt-delay 0
  exit
  hosts master-2
   priority      98
   preempt-delay 2
  exit
  hosts master-3
   priority      100
   preempt-delay 30
  exit
 exit
```



Note

- This configuration is required for all the proto pod VIPs (s11, s5, and N4).
- From this release onwards, the CLI command **clusters tmo2-smf virtual-ips** *interfaces* is deprecated for SBI interfaces for large-all deployment model.

GTPC Protocol Endpoint Merge with UDP Proxy Bypass

Feature Description

Bypass proxy is introduced to enable this GTP packets directly land on gtpc-ep pod. This will avoid the processing at udp-proxy and one hop will be reduced in packet forwarding.

All the features supported by existing gtpc-ep and udp-proxy are integrated in new merged path

following features are integrated from udp-proxy:

- Transaction SLA
- DSCP marking for GTP packets
- Adding BGP routes for roamer subscriber on the fly
- Supporting Dispatcher feature and incoming retransmission
- SGW Cache integration for DDN
- MBR cache integration

Following features are integrated from gtpc-ep:

- Retransmissions based on n3t3 config for outbound requests
- Monitor protocol and Monitor Subscriber
- Echo message handling

GTPC Endpoint with GR-Split

For handling scaled GTP traffic and for the optimal use of CPU, multiple active instances of GTPC-EP are started, and traffic split is done based on GR Instances.

ULB traffic routing architecture

The ULB integration utilizes a dedicated IPC Bridge within the service pods to facilitate direct packet steering and optimize GTP and PFCP ingress traffic flow. This design enables direct routing of GTP and PFCP ingress traffic between the ULB and the SMF service pods, eliminating intermediate protocol processing and enabling direct communication with the appropriate SMF service pod..

The design includes the following components:

- **Packet steering:** The ULB performs protocol-specific lookups to identify the target SMF service pod, enabling direct forwarding of GTP and PFCP ingress messages to the appropriate service pod.
- **Failure handling:** In the event of a pod failure, the ULB eBPF logic automatically detects service unavailability and redistributes traffic to the remaining healthy service instances. During node failures, the Maglev table at the ULB layer triggers re-population of indices to ensure that traffic is forwarded to the surviving nodes.

TEID redesign structure

To support ULB routing, the TEID structure is redesigned to embed routing identifiers that enable the ULB to identify the destination service pod without forwarding traffic through the legacy GTP pod. The encoding structure for GTP traffic is as follows:

Protocol	TEID encoding structure
SMF TEID	24-bit Random ID + 1-bit Nodemgr ID + 1-bit Rack ID
cnSGW TEID	22-bit Random ID + 1-bit Nodemgr ID + 1-bit GR Instance ID

PFCP SEID structure

To support ULB routing for PFCP traffic, the PFCP Session Endpoint Identifier (SEID) embeds routing identifiers that enable the ULB to identify the destination service pod. The ULB extracts the Node Identifier and Session Identifier from the SEID to steer PFCP messages to the appropriate SMF service instance.

Protocol	SEID encoding structure
PFCP SEID	4-bit Node ID + 22-bit Session ID (embedded within the 64-bit SEID)

UDP Proxy Functionality Merge into Protocol Micro-services

Feature Description

The UDP Proxy micro-services provide UDP transport termination for protocols (PFCP, GTPC, and RADIUS) that require UDP. The UDP proxy provides user space packet forwarding and IPC communication towards protocol micro-services. It uses host networking for source IP address observability and operates in Active/Standby mode.

Multiple protocol micro-services depend on UDP proxy for UDP transport, therefore UDP proxy is a scale bottleneck and single point of failure. Merging UDP Proxy functionality into respective protocol micro-services will help mitigate the scale bottleneck and improve CPU usage by virtue of reducing one hop across micro-services in the signaling path.

How it Works

PFCP Protocol Endpoint with UDP Proxy Bypass

Protocol endpoint bypasses UDP proxy and sends N4/Sxa messages towards UPF directly. Incoming N4/Sxa messages from UPF also bypass UDP poxy and land on Protocol pod. (Subject to UPF support for Source IP Address IE in heartbeat request message). Protocol pod continues to use non-host networking mode of operation.

Kubernetes service starts to listen on the configured VIP IP address and standard port, ensuring incoming N4/Sxa UDP packets are sent to Protocol pods. A separate Kubernetes service created for N4 & Sx with separate target ports to identify the interface associated with the incoming message/packet. Kubernetes client IP address affinity is availed to ensure retransmitted packets from UPF are sent to the same Protocol pod instance to hit the retransmission cache successfully.

Current Mode (No Bypass)

In this mode of operation the message exchange for N4, Sxa, GTP-U happen through the UDP proxy. The UDP proxy is responsible for connecting to or receiving connections from UPF.

All the node related messages, or session that is related on PFCP are initiated either by the service or from the UPF and their responses pass through the UDP proxy.

Outbound Bypass Proxy Mode

This mode of operation is enabled by default for all messages that are initiated by S-GW or SMF service and sent by the system toward UPF using PFCP through Kubernetes Pod environment variable "OUTBOUND_PROXY_BYPASS". The messages that are sent by SMF (Protocol pod) directly to UPF are session that is related and as follows:

1. PFCP Session Establishment Request
2. PFCP Session Modification Request
3. PFCP Session Deletion Request

In this mode, the GTP-U messages from UPF or initiated by Service toward UPF continue to be exchanged through the UDP proxy. In this mode, only the session related messages (that is, the ones initiated by SMF Service) flow directly from Protocol towards the UPF.

Protocol pod receives the UPF IP address from the service, which is used to set up connection with UPF and subsequently use the same for session related message exchange. The node related messages continue to take the UDP proxy to protocol or Node Manager path.

Complete Bypass Mode (Inbound and Outbound)

In this mode, both inbound and outbound messages are sent and received by Protocol pod bypassing UDP Proxy. The protocol pod will listen on N4 and GTPu or Sxa ports based on the configured VIPs. Protocol pod ceases to be on a Kubernetes service network and remains in Host based networking mode. Protocol pods gets the IP of the node or VM that it is on, this condition is triggered based on an environment variable present or available for both Protocol and UDP proxy pods (UDP_PROXY_BYPASS). By default, this variable is false and UDP proxy and Protocol continue as they do today with UDP-Proxy exchanging messages with UPF.

UDP_PROXY_BYPASS is set to true only if both the following conditions are met:

1. VIP is configured under endpoint PFCP interface N4 or interface Sxa.
2. There is no VIP configured under endpoint protocol interface N4 or interface Sxa.

With change in value of UDP_PROXY_BYPASS variable, both UDP proxy and Protocol pods are restarted to enable this new mode of working or to fallback to earlier mode of message exchange trhough UDP proxy.

Triggering Bypass Mode using CLI

To trigger the bypass mode or protocol-proxy merged working, the VIP-IPs must be configured under endpoint PFCP as shown here:

```
no instance instance-id 1 endpoint protocol interface n4
no instance instance-id 1 endpoint protocol interface gtpu
instance instance-id 1 endpoint pfcp interface n4 vip-ip X.X.X.X
instance instance-id 1 endpoint pfcp interface gtpu vip-ip X.X.X.X
```



Important With the preceding configuration the value of environment variable UDP_PROXY_BYPASS will change. This triggers a restart of both pods UDP proxy and Protocol.

Every feature that is present under endpoint→ protocol must be correspondingly configured under endpoint → PFCP and which include features like DSCP, SLA, and Dispatcher related configurations. The configurations for all features take effect only if internal VIP-IP is configured under endpoint → PFCP and interface N4 or interface Sxa. There should be interface N4 and VIP-IP or interface Sxa and VIP-IP present under endpoint → protocol.

Rendering CLI Values

Based on N4 and Sxa VIP configuration, the rendering logic calculates which values to publish under endpoint protocol. The configuration is rendered in pods having the key as “endpointIp”. The configuration path in each individual pod is located at /config/AppName/vip-ip/endpointIp.yaml. The affected pods are:

1. Protocol
2. Node Mgr
3. SMF-Service
4. SGW Service.

Having endpoint → pfcp configurations render under endpoint → protocol helps in avoiding changes to background configuration read logic.

Node Management

In this case Protocol starts a PFCP endpoint for peers to connect with it. At the same time, it will also establish connection with UPF as and when the app service initiates a PFCP message towards the UPF. Following messages are included:

1. PFCP Association Setup Request/Response
2. PFCP Association Update Request/Response
3. PFCP Session Report Request/Response
4. PFCP Node Report Request/Response
5. Heartbeat Request/Response

Session Management

Session Management messages initiated by the service and sent directly to UPF through the Protocol pod. The protocol pod initiates connection with UPF to send these messages, this is the reason protocol pod must be in “Host networking” to take the IP address of the node on which it is on.

Standardized Port Numbers

While triggering the “Merged” mode, the protocol pod transitions into Host based networking. Protocol pod takes the IP address of the Host or the Node much like the existing UDP proxy pod. It is essential that UDP proxy, GTPC-EP, and Protocol do not share the same ports. The thumb rule for port calculation is:

```
Port_Value = Base_Port _Value + (Gr_Instance_Id_index *50) + (Logical_Instance_id mod 50)
```

Gr_Instance_id: The GR Instance ID supplied in the configurations using CLI.

Logical_Instance_id: Identifier for the logical SMF instance.

Prometheus Port:

With complete UDP proxy bypass the Prometheus port of 8080 is not used, instead the start port for Prometheus 8004 for instance 1. The "instance-Id" added with 8003 must be the port number.

Proxy Keep-Alive Port:

The proxy keepalive port starts from 27500+ "Instance-Id".

1. GR Instance 1 & Logical Instance Id 0 :- 27500 + (0 * 50) + (0 % 50) = 27500
2. GR Instance 2 & Logical Instance Id 0 :- 27500 + (1 * 50) + (0 % 50) = 27550

Admin Port for Keepaive and Liveness Probe:

Admin Port will be 7879 + (Gr_Instance_Id_index *50) + (Logical_Instance_id mod 50)

Infra Diagnostics Port:

Infra Diag Port will be 7779 + (Gr_Instance_Id_index *50) + (Logical_Instance_id mod 50)

PProf port:

PProf Profiling port will be 7679 + (Gr_Instance_Id_index *50) + (Logical_Instance_id mod 50)

Runtime pod scales for server additions

Table 11: 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 SMF 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

SMF 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 SMF) 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
```

All-in-One deployment models

Table 12: Feature history

Feature name	Release information	Description
Low footprint deployments for SMF using AIO models	2025.04.0	AIO models are designed to support low footprint deployments for SMF. Deploying the AIO models allows the network operators to deploy high capacity deployments in the resource-optimized way.

All-in-One (AIO) models are low footprint converged core deployment models that allow the network operator to support increased UEs with increased capacity hardware.

Thesee are the two AIO deployment models:

- **AIO:** AIO deployment model provides full functionality with optimized resource allocation. This model is optimized for high throughput and session density on single-node deployments.
- **AIO-small:** AIO-small deployment model is resource-optimized for smaller hardware footprints and maintaining functionality.

This table shows the details of the AIO deployment models for SMF.

Table 13: Specifications for AIO deployment models

Specifications	AIO	AIO-small
Capacity	Up to 1 million SMF sessions	Up to 400,000 SMF sessions
Recommended for	Single node deployment with 128+ CPU cores.	Single node or partial node deployment with 32+ CPU cores
Traffic mix	90% 4G sessions and 10% 5G sessions.	90% 4G sessions and 10% 5G sessions.
Suitability	Suitable for production environments with high throughput requirements.	Suitable for testing, development, or smaller production deployments.

Note

- High Availability is supported on AIO GR model through GR triggers.
- High Availability for AIO model is not supported without AIO GR.
- The AIO models support traffic and pod monitoring functionalities.

Configure AIO deployment models

Before you begin

These are the prerequisites to deploying the AIO models:

- Cluster Manager Configuration
  - Cluster manager must deploy the single-node cluster.
  - All the required network configurations must be in place.
  - The node labels must be properly configured for pod placement.
- Network Configuration
  - All the required Virtual IPs (VIPs) must be configured.
  - The network connectivity must be established between the components.
  - Proper DNS resolution must be configured.
  - The AIO GR networking model should be the same as the multi-node GR cluster networking model.

For more information, see the Geo Replication section.

- Operations Center Configuration
  - Both CEE and SMF operations center must be configured with "single-node true" parameter.
  - Proper monitoring and alerting must be set up.

These steps allow the network operator to deploy AIO models for SMF.

Procedure

Use the CLI **deployment model { aio | aio-small }** in the global configuration mode.

Example:

```
[smf] smf#config
[smf] smf(config)# deployment model aio
[smf] smf(config)# end
```