Cloud-native packet core network functions

Requirements, design and architecture

Application note

Abstract

Service providers are using cloud-native technology to automate and simplify network operations, and thereby reduce OPEX. These cloud-native paradigms, which have been used mainly for very scalable and distributed web applications and services, can also be applied to packet core functions and gateways to achieve scalability, automation, high availability, deployment flexibility and service velocity.

This application note describes some of the general principles of cloud-native application architecture and explains how these are applied to the Nokia Cloud Packet Core—enabling a service provider to build a single core that supports 5G, Evolved Packet Core (EPC) and 2G/3G packet switched domains..

Contents

Introduction

To reduce OPEX, service providers are automating and simplifying their network operations, thereby achieving faster time-to-market for new services and the ability to deploy services across diverse cloud environments. Cloud-native technology allows rapid development, automated deployment, life-cycle management (LCM), scaling and healing of applications.

Cloud-native paradigms have been used mainly for very scalable and distributed web applications and services. These same paradigms with some specific considerations can be applied to packet core functions and gateways to achieve scalability, automation, high availability, deployment flexibility and service velocity.

The Nokia Cloud Packet Core solution is architected to fully leverage cloud-native techniques and methodologies. Disaggregated software and a state-efficient design that enables separation of state from processing are fundamental to the architecture of virtualized network functions (VNFs) that comprise the Nokia Cloud Packet Core.

As a leading member of the Cloud Native Computing Foundation (CNCF), Nokia actively contributes to several projects in the CNCF. In addition, the Nokia Cloud Packet Core uses a multitude of open-source software services and frameworks developed by the CNCF.

To meet varying business and operational objectives, the Nokia Cloud Packet Core software is built as infrastructure agnostic. It uses common application software whether it is deployed as a set of physical network functions (PNFs) or as VNFs in either virtual machines (VMs) or containers. The Nokia Cloud Packet Core and its cloud-native capabilities are also independent of the underlying cloud infrastructure used for orchestration, LCM and infrastructure resource management of VNFs.

The Nokia Cloud Packet Core supports 5G Core, Evolved Packet Core (EPC) and 2G/3G packet switched domains, allowing a service provider to build a single core with cloud-native capabilities.

This application note describes some of the general principles of cloud-native application architecture and explains how these are applied to the Nokia Cloud Packet Core solution.

Attributes of cloud-native architecture

This section discusses the key architectural pillars of cloud-native applications and the applicability of these pillars to packet core network functions.

Software disaggregation with micro-services

Micro-services architecture

A cloud-native application is implemented as a set of loosely coupled software components, also known as micro-services, which coordinate to deliver the business logic while running as independent processes. These micro-services can be managed, scaled, restarted and updated independent of each other. These capabilities provide efficient resource utilization as well as rapid delivery of software changes. The interaction between these micro-services is via well-defined versioned application programming interfaces (APIs).

State-efficient network functions

In a stateless architecture, the transient but enduring state of entities is separated from the application engine that processes those entities. For packet core control plane elements, these entities are the sessions that represent the individual subscribers that connect to a mobile network.

The application engine (typically a set of state machines and associated logic) processes the subscriber sessions according to procedures defined in 3GPP specifications. The engines implementing the business logic keep a local cache of session state during execution. On completion of a procedure, the engine stores the state in a resilient, high-performance, key-value data-store.

Preserving the local cache allows efficient transaction processing at high rates while the database store optimizes redundancy and scaling operations. For example, if a processing engine fails or needs to be restarted, it can pull the relevant state from the database when needed, to process a subscriber transaction. Furthermore, having the state in the database enables the engine to purge its cache, when needed, to more efficiently utilize resources.

Containerization

Containers provide a packaging format that includes the application as well as all the operating system (OS) dependencies and runtime information required. Unlike VMs, which include the complete guest OS, containers use the host OS and leverage Linux namespaces to provide partitioning of processes, the networking stack and the file systems of the underlying host OS. As a result, container images require a smaller memory footprint than VMs and have very fast spin-up times of typically a few 100 milliseconds to a few seconds. This allows fast scaling and healing of containerized applications.

A disaggregated cloud-native application is made up of multiple containers, each of which runs one or more micro-services. Because containers depend only on the stable underlying Linux kernel API, containers are very portable. Therefore, it is easy to seamlessly run a Containerized Network Function (CNF) in different environments: development, staging or production.

The portability of containers applies equally to private, public and hybrid clouds, enabling redistribution of workloads as needed.

Orchestration and automation

The cloud-native applications that are executed as a set of micro-services can have multiple instances, which can be spawned on demand to scale the application. The micro-services can be independently scaled, healed and updated.

Deploying such cloud-native applications manually is operationally very onerous. Orchestration and automation to deploy, scale, heal and update containerized applications is the key to streamline operations and reduce a service provider's overall OPEX.

Kubernetes (K8s) has emerged as the accepted open-source container orchestration system for automating the deployment, scaling operations and management of containerized applications. K8s manages the life cycle of containerized applications in a cluster of nodes, which is a collection of worker machines such as VMs or physical servers.

The simplicity resulting from K8s comes from its declarative model, where the user needs to specify only the desired state of the deployment through manifests and policies in the form of a YAML file and Helm charts. K8s reconciles the system to provide the desired state of the deployment.

DevOps

The DevOps methodology consists of in-service software and configuration updates plus continuous integration/continuous delivery (CI/CD).

In-service software and configuration updates

A cloud-native application should support independent updates of each micro-service. If the micro-service being updated has multiple instances for scaling, a rolling update can be performed seamlessly with minimal impact to user traffic. Versioning of data exchanged between different micro-services enables seamless incremental update of the micro-services within the CNF.

CI/CD

As new software versions are delivered with enhanced features and bug fixes, a smooth workflow needs to be created that integrates the software into a running network. Automated pipelines can utilize the Kubernetes engine and software upgrade procedures for fluid deployment and validation of incremental software changes. These can be installed in a lab environment and then, pending validation, automatically transitioned into an isolated staging environment before broadly deploying in a production field.

This process of CI/CD automates the entire workflow from vendor delivery to production field in a controlled and supervised, yet highly automated, way.

Tools and metrics

A cloud-native deployment can be enhanced by integration with open-source tools. Metrics are used to assess the performance and health of an application.

Open-source tools

Cloud-native applications should be easily integrated with common, open-source tools for event logging, metrics collection and visualization. Event logs capture security, application and debug events and are continuously generated by CNF. The logs can be stored locally in the application or can be streamed to a collector.

Following the cloud-native paradigm, logs should be treated as event streams and not stored local to the application. This means a service provider can choose to build a common logging strategy for all CNFs.

Fluentd, Elasticsearch and Kibana are a common triad of services for logging. Logs are collected by Fluentd, converted to a common Javascript Object Notation (JSON) format, and stored in Elasticsearch. The service provider can use Elasticsearch and Kibana to visualize historical logging information.

Metrics

Metrics are various counters that are used to assess performance and health of an application. Metrics should be streamed to a collector for faster and more accurate processing.

Prometheus is an example of a popular open-source metric collector. Prometheus relies on polling the application through a REST API and can be integrated with K8s to facilitate discovery of application services.

Prometheus stores metrics as a time series within its own database, allowing a service provider to analyze historical data. The provider can also use Grafana to visualize metrics through Prometheus.

Nokia cloud-native packet core network functions

Key EPC and 5G Core network functions are delivered by the Nokia products described in this section. All of these products have inherent cloud-native design principles and can be deployed as disaggregated CNFs.

Cloud Mobile Gateway

The Nokia Cloud Mobile Gateway (CMG) delivers:

- Trusted wireless access gateway (TWAG)
- Gateway GPRS Support Node (GGSN)
- Serving gateway-control and serving gateway-user (SGW-C, SGW-U)
- Packet data network gateway-control (PGW-c) and packet data network gateway-user (PGW-u)
- Evolved packet data gateway (ePDG)
- Session management function (SMF)
- User plane function (UPF)
- Non-3GPP interworking function (N3IWF) complying with the 3GPP service-based architecture (SBA).

The CMG control-plane (CMG-c) and user-plane (CMG-u) can be decoupled as defined in the 3GPP control and user plane separation (CUPS) architecture. The decoupled control and user plane functions can be individually deployed as CNFs, and each CNF can have independent placement, scaling, healing and updates.

Cloud Mobility Manager

The Nokia Cloud Mobility Manager (CMM) supports the following combined network functions:

- General Packet Radio Service (GPRS)
- Mobile management entity (MME)
- Serving GPRS support node (SGSN)
- Access and mobility management function (AMF).

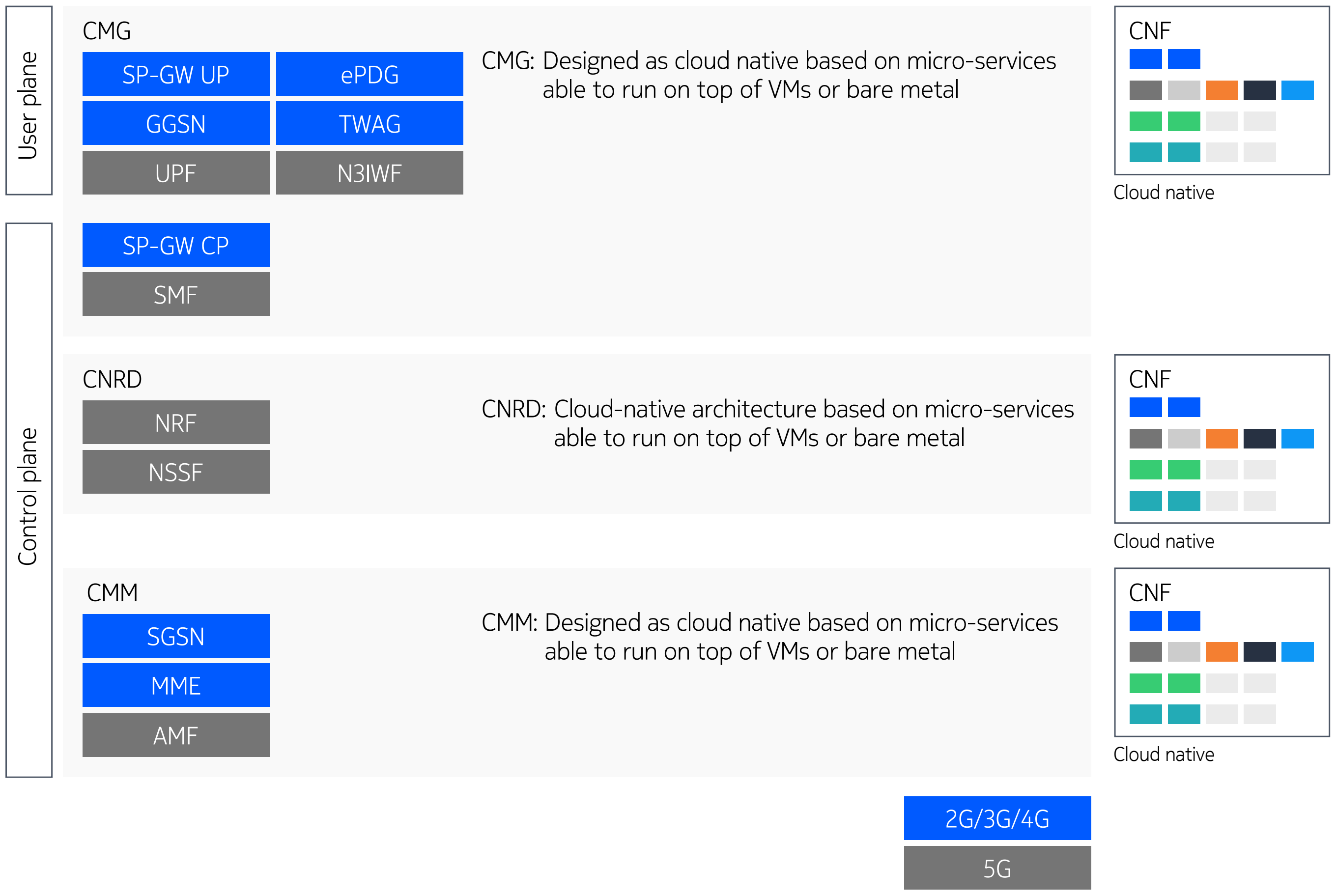
Cloud Network Resource Director

The Nokia Cloud Network Resource Director (CNRD) supports the network function repository function (NRF) and network slice selection function (NSSF) as defined in the 3GPP SBA.

Subscriber data, policy and charging are also available as Nokia products but are not discussed in this document.

Figure 1 shows the CNFs of the CMG, CMM and CNRD.

Figure 1. Nokia cloud-native packet core network functions


User plane
Control plane
CMG
SP-GW UP
ePDG
GGSN
TWAG
UPF
N3IWF
SP-GW CP
SMF
CMG: Designed as cloud native based on micro-services able to run on top of VMs or bare metal
CNF
Cloud native
CNRD
NRF
NSSF
CNRD: Cloud-native architecture based on micro-services able to run on top of VMs or bare metal
CNF
Cloud native
CMM
SGSN
MME
AMF
CMM: Designed as cloud native based on micro-services able to run on top of VMs or bare metal
CNF
Cloud native
2G/3G/4G
5G

Design and architecture of Nokia CNFs

While the Nokia CMG, CMM and CNRD follow cloud-native design principles, this application note focuses specifically on the design and architecture of the CMG-c and CMG-u as representative CNFs and on considerations for integration with K8s.

CMG-c CNF

CMG-c CNF is implemented as a set of disaggregated micro-services deployed on K8s as a set of pods. Some of the pod types are unique to the CMG-c CNF but several are open source and used with other CNFs.

Unique pod types

The following are the key pod types that are unique to CMG-c:

- **SMF/PGW-c:** The main call processing pod for Wi-Fi®, GSM, UMTS, EPC and 5G session management functions; performs session, policy and charging management for all access types and enables smooth handoff between technologies.

- **Operations, administration and maintenance (OAM):** Provides management access and routing, and load-balancing across SMF/PGW-c pods.
- **Database:** SMF/PGW-c pods store the state of their session transactions here so that future transactions can be performed by alternate pods in case of SMF/PGW-c failure or scale operations. The database pod type uses a NoSQL (Structured Query Language) database.
- **Load balancer (LB):** A Nokia-developed load balancer (LB) used for Transmission Control Protocol/User Datagram Protocol (TCP/UDP) such as GPRS Tunneling Protocol (GTP-C), Packet Forwarding Control Protocol (PFCP) and Diameter. GTP-C and Diameter apply when EPC interworking is used for migration to a 5G Core. Cluster external access is provided through the LB.

  The LB is exposed externally to the K8s cluster via multiple options:

  - Multus/single root-input/output virtualization (SR-IOV) or open v-switch-data plane development kit (OVS-DPDK) for direct access to the LB pod.
  - Kube-router, which is an open-source Layer 4 (L4) proxy in tunnel mode that uses IP virtual server (IPVS) to load-balance traffic to the LB pods.

  Inter-pod communication internal to the cluster is over the K8s-native container network.

Shared pod types

The following pod types are used with the CMG-c CNF but can be shared with other CNFs:

- **Istio gateway:** An open-source Istio pod (based on Envoy) that performs Hypertext Transport Protocol (HTTP) load-balancing within the K8s cluster based on rules configured by a CMG administrator. Istio allows flexible load-balancing and Canary-style CMG upgrades. Sidecar containers are injected into specific CMG pod types.
- **NGINX:** An HTTP proxy that exposes the CMG instance to external cluster clients.

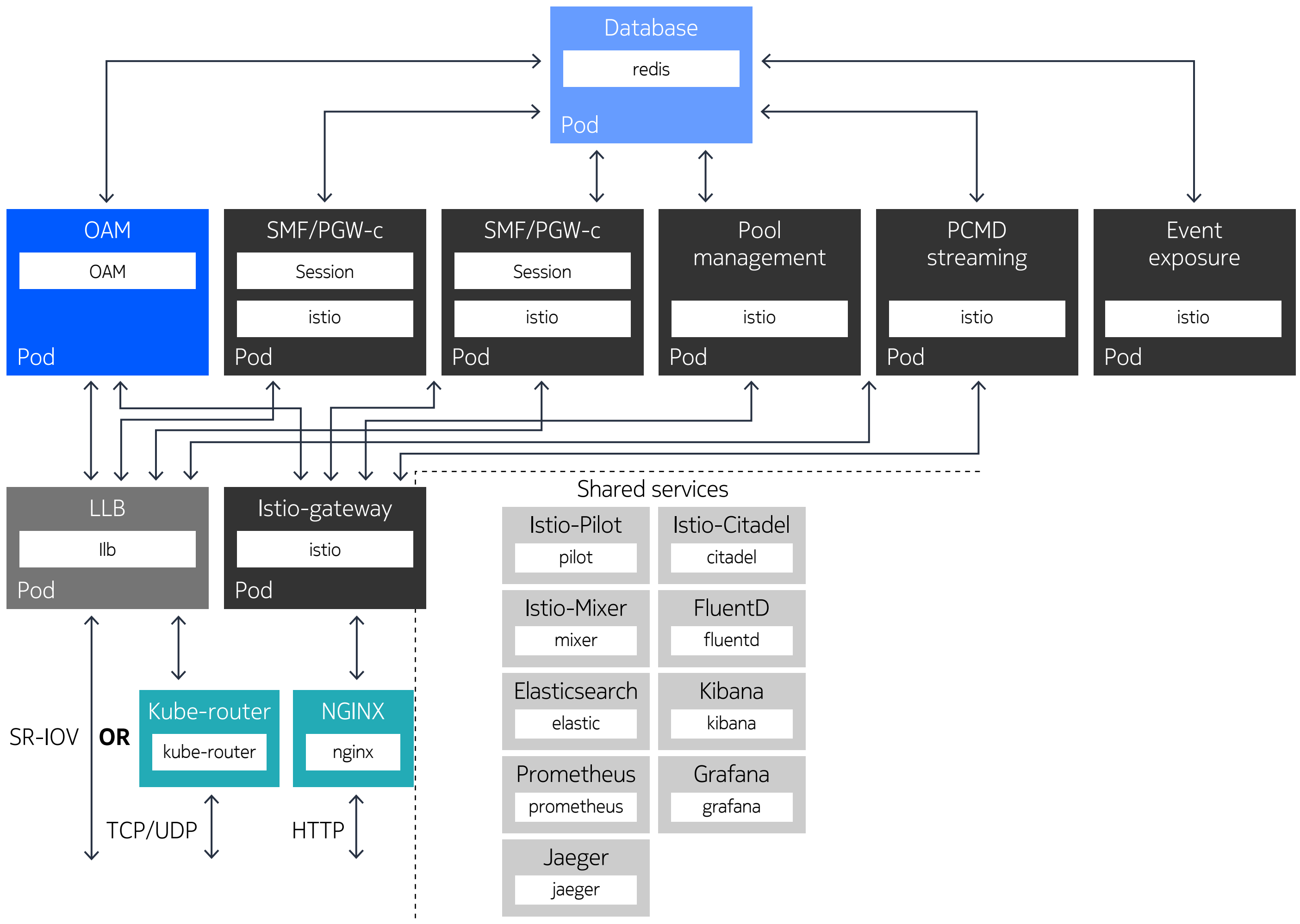
Open-source services

The CMG-c is integrated with several open-source services that run inside the cluster:

- **Prometheus/Grafana:** Gathers metrics from CMG-c specific and shared services.
- **Fluentd/Elasticsearch/Kibana:** Used for logging CMG-c and shared services.
- **Jaeger:** Used for tracing between services.
- **Istio Pilot/Mixer/Citadel:** Controllers for the Istio-gateway and sidecar containers.

Figure 2 shows the CMG-c CNF.

Figure 2. CMG-c CNF



The CMG-c CNF is planned to be disaggregated further into additional services running on different pods (e.g., address pool management service, per-call management data (PCMD) streaming service and event-exposure service).

CMG-c architecture highlights

The CMG-c has a number of architectural and functional highlights as detailed in Table 1.

Table 1. CMG-c architectural and functional highlights

CMG-c architecture highlights	
Highlight	**Description**
Deployed as a set of containerized disaggregated micro-services	Supported as a container-in-VM or bare-metal deployments
Full K8s integration	• Internal/external networking with key K8s concepts: Services, LBs, Istio for service mesh • Deployment and LCM automated via Helm3 and K8s
Comprehensive integrated load-balancer function	• Nokia-developed TCP/UDP for legacy protocols • Augmented with third-party HTTP/2 load-balancer for 5G Core
Integration with common open-source management components	Utilized for service-mesh, event logging, metrics gathering and tracing
CMG-c functional highlights	
Highlight	**Description**
Supported for EPC and 5G Core	Combined SMF/PGW-c functions, enabling EPC-to-5G Core interworking
State-efficient design	User equipment (UE) state in separate data-store (Redis) to allow efficient and dynamic scale-out and redundancy
Multiple KPI-gathering options	Customized markup language (XML) files, Prometheus, remote procedure calls (gRPC)-based model-driven telemetry

CMG-u CNF

The CMG-u CNF that comprises the SGW-u, PGW-u and UPF is implemented as a set of disaggregated micro-services deployed in K8s as a set of pods. Some of the pod types are unique to the CMG-u CNF but several are open source and used in common with other CNFs.

Unique pod types

The following pod types are unique to CMG-u:

- **UPF/PGW-u:** Performs local control plane and user plane functions on the CMG-u. Local control plane functions include per-session PFCP (Sx/N4) processing to create forwarding and traffic management state, and usage tracking and reporting as directed by CMG-c.

  User plane processing on the UPF/PGW-u pod includes the standard 3GPP forwarding as well as providing integrated Gi-LAN/N6 functions such as deep packet inspection (DPI), network address translation (NAT) and TCP-optimization (TCP-O).

  The Gi-LAN functions can be run as separate micro-services in their own pods or as standalone CNFs. However, when they are integrated with the user plane processing, the throughput is maximized and latency is minimized by eliminating additional I/O, flow-lookups, and inter-pod east-west traffic. The CMG-u pod provides independent in-service updates of DPI signatures. Service providers should carefully weigh the performance impact of disaggregating data path functions.

  The UPF/PGW-u pods can scale-in/scale-out on demand. Auto-scaling operations on the CMG-u do not require any interaction with the CMG-c.

  A key attribute of Nokia's CUPS implementation is complete independence between the CMG-c and CMG-u with respect to local scale-in/scale-out and failover of respective pods. This is a key operational advantage that makes life simpler for service providers.

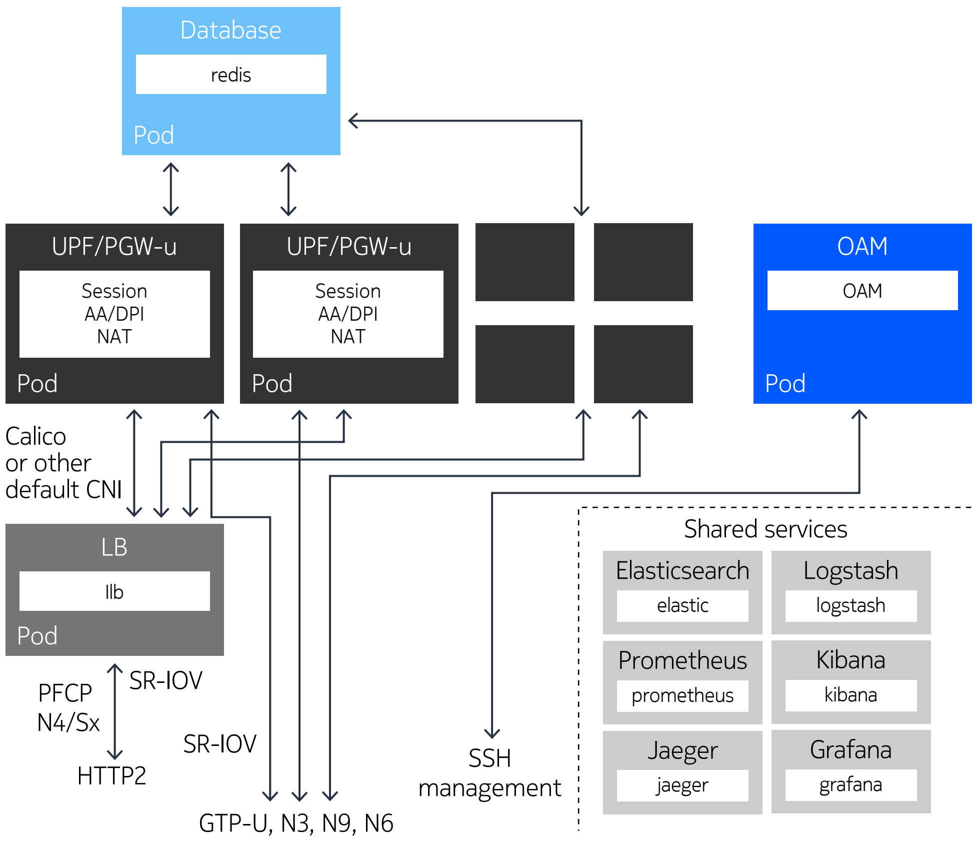
Furthermore, user plane traffic lands directly on UPF/PGW-u pods, which have external access through SR-IOV on the worker node.

- **Database:** A separate Redis database for user-plane forwarding tables eliminates the need for any Sx/N4 interaction with the CMG-c for local redundancy.
- **LB:** Performs TCP/UDP load-balancing towards the UPF/PGW-u pods and provides cluster external access for control plane (PFCP, HTTP2). The LB is exposed externally via the same options as the LB on CMG-c: direct access to the pod via Multus/SR-IOV or via Kube-router utilizing native Kubernetes networking.

  User plane traffic lands directly on UPF/PGW-u pods to maximize throughput, thereby avoiding an additional LB hop. Therefore, UPF/PGW-u has direct external access through Multus/SR-IOV on the node. Typically, an LB becomes a bottleneck for the user plane pods because the port density and capacity of network interface cards on the LB limits the number of pods the LB can serve. Inter-pod communication internal to the cluster is over the K8s-native container network.
- **OAM:** Provides management access and routing, and assists the load-balancing layer in distributing incoming control plane messages (e.g., PFCP interaction with SMF or HTTP2/SBA-based interaction with the NRF) among the set of UPF/PGW-u pods.

Figure 3 shows the high-level micro-services architecture blueprint for CMG-u.

Figure 3. CMG-u CNF

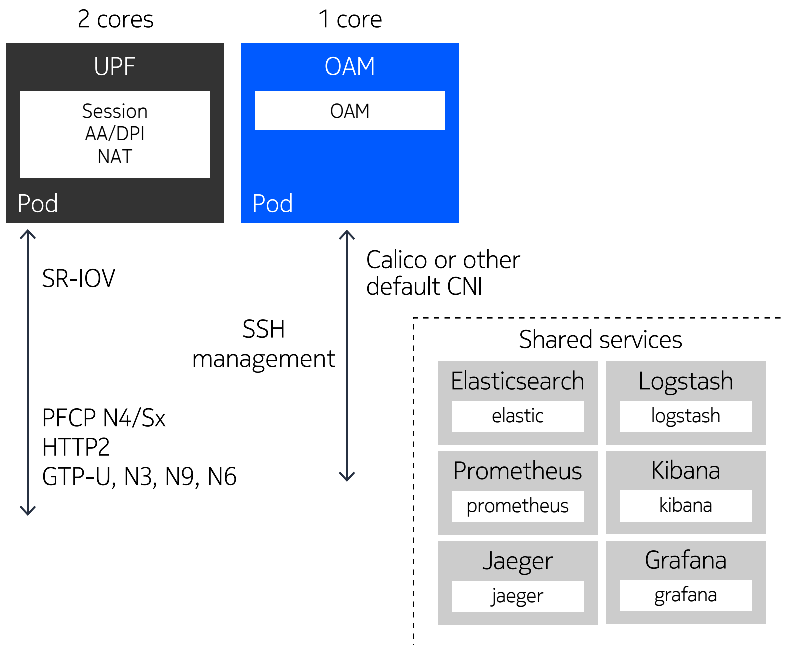


Deployment flexibility

The CMG-u can scale to 1Tb/s of capacity on current off-the-shelf hardware. At the other end of the spectrum, for highly distributed multi-access edge computing (MEC) deployments, a small footprint CMG-u of just three cores can perform an SGW/SAEGW (System Architecture Evolution gateway) or an Intermediate-UPF (I-UPF) function of local breakout of traffic to distributed applications (see Figure 4).

This tiny footprint is capable of over 5 Gb/s of throughput and is a fully functional UPF with all the value-added functions, of which NAT is of particular interest. Furthermore, it has all the capabilities of an edge router, making it the perfect front-end for all applications at the MEC site.

Figure 4. Micro CMG-u for distributed MEC deployments



The CMG-u has a number of functional highlights as detailed in Table 2.

Table 2. CMG-u functional highlights

CMG-u functional highlights	
Highlight	**Description**
Supported for EPC and 5G Core	Combined UPF/PGW-u functions, enabling EPC-to-5G Core interworking
Optimized footprint that maximizes throughput and minimizes latency	• Appropriate level of disaggregation without impacting throughput and latency • Gateway data-path with integrated Gi-LAN/N6 functions • LB-less user plane • Direct-to-pod accelerated I/O with Multus/SR-IOV or OVS-DPDK
Seamless local scaling for failover	Forwarding state in separate Redis database; no Sx/N4 interaction with SMF/PGW-c on failover or sale-out
Scalable CMG-u version	Single instance scalable on demand (up to 1Tb/s)
Micro CMG-u for distributed MEC deployments	Minimum-sized instance requiring just three cores for a fully functional user plane

Features and considerations of containers

The features and considerations of containers include:

- Overcoming container networking challenges
- Availability
- Serviceability
- Security
- Container management.

Overcoming container networking challenges

The K8s network model is implemented using container network interface (CNI) plug-ins. Nokia CNFs work seamlessly with commonly used plug-ins.

When Kubernetes instantiates a pod, it assigns an ephemeral IP address out of a configured pool. As a result, if a pod restarts, it may get a different IP address.

The K8s service abstraction provides a stable fully qualified domain name (FQDN) or IP address to represent a set of backend application pods. The service abstraction can be used to establish a native load-balancing and selection scheme to the application pods. Clients that interact with a K8s service may have their requests served by multiple pods without needing to identify the pods or their IP addresses.

Nokia CNFs support standard K8s networking for container-to-container, pod-to-pod, pod-to-service and external-to-service communication. However, default K8s networking presents several operational challenges that Nokia's innovative architecture has solved.

Single IP address for a pod

By default, K8s provides a single IP address for a pod. An important operational requirement for packet core functions is to support multiple IP interfaces in different routing contexts for isolation.

For example, separate IP addresses and routing contexts can be used for the management interface, signaling interfaces to peers in the home network, roaming interfaces and interfaces to external servers that include the policy control and rules function (PCRF), online charging system (OCS) and the authentication, authorization and accounting (AAA) server.

Nokia CNFs support the Multus CNI plug-in, a delegating CNI for K8s that can call multiple CNI plug-ins. This enables multiple network interfaces on a pod beyond the default network interface used for pod-to-pod communication. The network interfaces can be associated with different virtual routing and forwarding (VRFs) to support routing separation.

The CMG is based on Nokia's industry-leading routing software stack. This feature-rich routing stack is used for exchanging reachability to peers on the RAN side and to announce reachability of UE pools to routing peers on the network side.

Layer-4 proxies

Native implementation of K8s service relies on L4 proxies that run in the kernel or in the user space. These proxies have the negative effect of hiding the client's original source and destination IP of traffic to and from the application pods. In the case of HTTP, an external/upstream proxy can insert the client source/destination in an HTTP header. (This is the approach Nokia takes for HTTP.)

However, packet core network functions often depend on knowing the client source IP, especially in EPC cases where a peer's identity is only known by its IP address. The client IP is used to apply specific policies as well as for identifying traffic load from a peer and for troubleshooting. 5G Core network functions have identities (network function instance IDs), so identification by source IP is less important. But for EPC peers (those running GTP-C), original source and destination IP addresses need to be visible.

There are two approaches that the Nokia CMG can use for maintaining source/destination IP addresses:

- **Host-networking:** Involves plumbing the application pods directly to the node's interfaces. For example, a pod can be configured to request SR-IOV or OVS-DPDK connectivity to the node interface. SR-IOV and OVS-DPDK can also be used to increase packet forwarding performance.

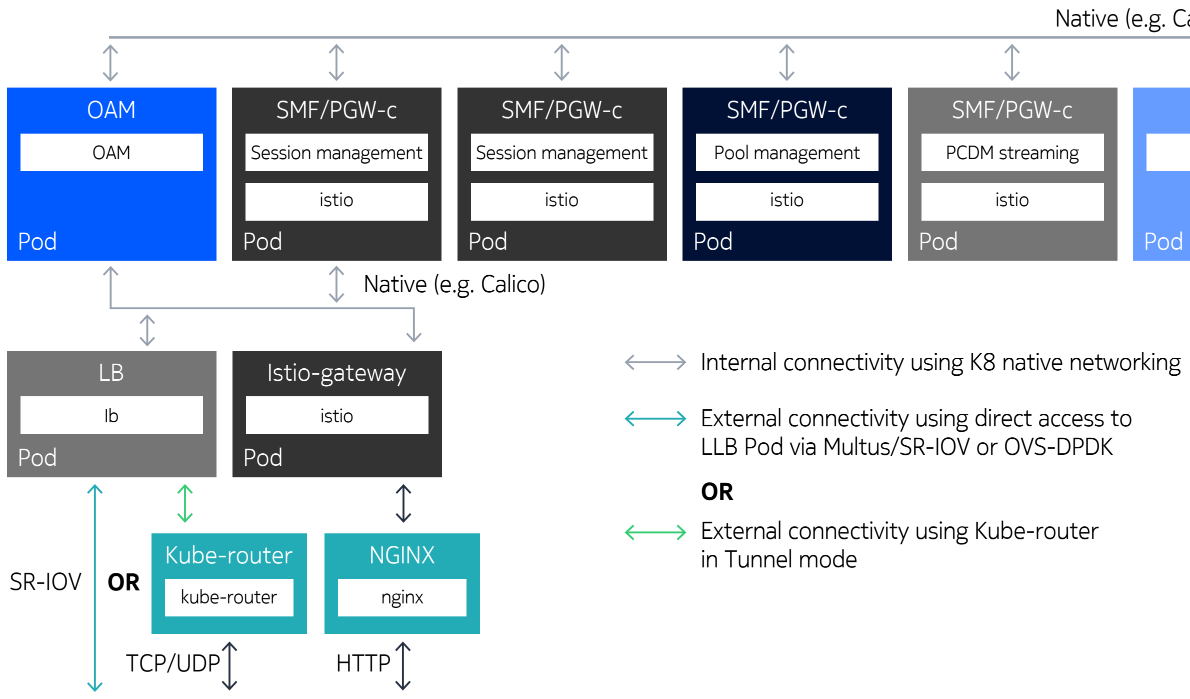
  The LB pods in the CNF provide cluster external access. Using SR-IOV or OVS-DPDK for external access to the LB avoids the IP identification issue by bypassing K8s networking and the underlying load-balancing and selection that the K8s service construct provides. The load-balancing is solved at the application level.

  Internal to the cluster, the CNF's inter-pod traffic uses application-provided tunneling over the underlying native K8s network setup by a CNI plug-in (e.g., Calico, shown in Figure 5).

- **Tunneling approach:** The CNF can use a separate proxy to tunnel the original packet through to a pod so as not to lose the original source/destination IP address. Public cloud providers offer such tunneling proxies typically to allow Direct Server Return. However, as an alternative for private clusters, an IP Virtual Server (IPVS) can be used as a tunneling load balancer.

  "Kube-router" is an open-source solution using IPVS in tunneling mode for private clusters, and also incorporates border gateway protocol (BGP) for equal cost multi-path (ECMP)-style load-balancing from an external router. With this approach, Kube-router at the edge of the cluster exposes the service endpoint as a K8s service to external peers. As a proxy, Kube-router then load-balances and tunnels traffic to the application's LB pods.

Figure 5. Internal and external network connectivity in CMG-c CNF



Limited support for IPv6 and dual-stack

Support for IPv6-only and dual-stack is still immature in K8s.

Nokia CNFs work with both IPv6-only and dual-stack for pod-to-pod and external networking. IPv6-only and dual-stack UEs are supported. An integrated NAT-64 function is also available.

Table 3 shows how Nokia CNFs overcome container networking challenges.

Table 3. Overcoming container networking challenges with Nokia CNFs

Container networking requirement	Nokia CNF functionality
Compatible with multiple CNIs	Nokia CNFs are verified to seamlessly work with several CNIs: Calico, Flannel, SR-IOV, host-device
Support multiple network interfaces per pod	Nokia CNFs support multiple interfaces per pod in different routing-contexts for security and isolation
Preserve original peer IP address to application pods through K8s L4 proxies	Full peer visibility for tracking and debugging
Provide IPv6-only and dual-stack network connectivity and UE	Nokia CNFs are ready for IPv6-only or a dual-stack K8s cluster

Availability

In contrast to VMs, container images are very small and can typically start in few hundreds of milliseconds. K8s can be configured to provide very fast pod and node failure detection. Coupled with the state-efficient design of Nokia CNFs, this allows for fast, predictable and cost-effective redundancy models. It is possible to achieve sub-second failover times with limited or no spare resources set aside within the CNF.

Nokia control plane CNFs support both all active (N+) redundancy with no resources in standby or N:K redundancy with limited resources in standby. A failed pod can restart and populate its cache of the state for impacted sessions from the state database.

For data-plane pods that perform packet forwarding, intra-CNF redundancy supports all-active (N+), N:K and 1:1 redundancy models. For latency-sensitive voice over LTE (VoLTE) or voice over New Radio (VoNR) calls to persist during failover, 1:1 redundancy can be enabled. For consumer internet traffic, all-active or N:K redundancy suffices. Nokia CNFs support a choice of redundancy models based on access point name (APN) or data network name (DNN).

Serviceability

Serviceability consists of deployment automation, auto-scaling and in-service software updates.

Deployment automation

Using K8s constructs, Nokia provides automation of deployment, configuration and LCM. Helm charts are used for automated deployment of CNF application pods and components such as the database and CNIs. In addition, for reusability of Nokia CNFs in different environments (e.g., development, staging, production environments; private or public clouds), the templating of K8s manifests supported by Helm is leveraged. The Helm charts provided contain YAML template files and values for sets of parameters in the templated YAML.

The Helm charts are used for Day One configuration automation. K8s configuration maps (ConfigMaps) and secrets are used to inject configuration into the OAM (configuration management) container for the Nokia CNF. Helm charts provided for configuration management contain ConfigMaps and secrets for baseline configuration.

The configuration itself can include parameters to automate deployment for multiple CNF instances. Per-CNF instance parameters (via values.YAML) can be provided during installation and upgrades; this streamlines deployment and updates for large rollout of multiple CNF instances; for example, the deployment of hundreds of micro CMG-u instances in distributed MEC locations.

As an alternative to ConfigMaps, Nokia also supports model-driven (YANG-based) transactional configuration via Netconf or gRPC network management interface (gNMI).

Nokia-provided Helm charts enable a mechanism for CI/CD. Helm charts can be integrated into a Jenkin's pipeline and even connected to a webhook acting on the existence of new image versions pushed to a repository, to trigger a pipeline to install or update a CNF instance's software.

Beyond basic automation for application provisioning and configuration management, using Nokia-provided custom K8s Operators enable service providers to ease their Day Two management of CNF workloads. Our custom K8s operations allow automation for:

- Monitoring alarms, key performance indicators (KPIs) and key change indicators (KCIs)
- Scale-in/scale-out based on platform- or application-specific metrics
- In-service software upgrades with or without Canary testing
- Simulating failure in all or part of the cluster to test resilience.

Nokia CNFs leverage Helm and K8s Operator framework to:

- Provide automation for deployment, configuration and LCM
- Simplify customization for Nokia CNF deployments in different environments
- Ease deployment of CNFs, e.g., CMG-u instances for large distributed MEC deployments

Auto-scaling

Nokia CNFs support automatic scale-out and scale-in via the K8s horizontal pod auto-scalar (HPA) as a default mechanism. This allows scaling based on CPU and memory utilization metrics. However, auto-scaling based on custom metrics (e.g., number of protocol data unit sessions) is also supported. This is based on Nokia-provided custom K8s operators for Nokia CNFs.

Prometheus is supported as a metrics collector and metrics API server.

In-service software updates

Nokia CNFs provide in-service software updates by using the underlying K8s rolling-update capability to incrementally update pods that constitute the CNF. The rolling updates are used for both image and configuration changes.

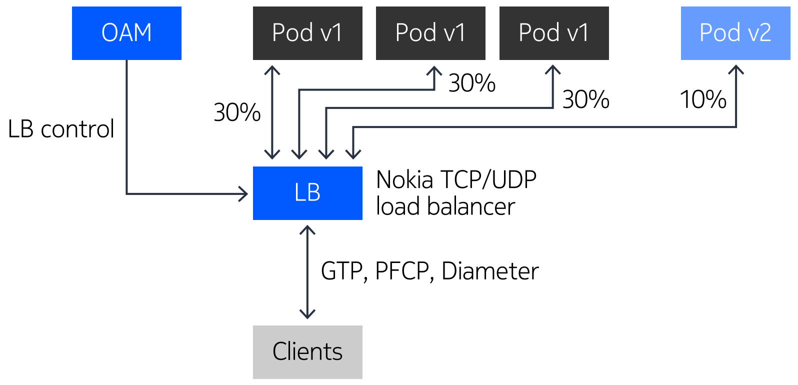
The update procedure can be flexibly managed based on settings allowed by K8s. These settings are used to control a number of pod-related aspects, including the number of new pods to spin up at once, the number of outstanding pods, health check of new pods and the delay in deleting old pods.

Multiple instances of different pod types that correspond to disaggregated mirco-services of a CNF can be upgraded indepdently. Versioning of data exchanged between different pod types is supported for seamless incremental update of pods within the CNF.

The update can be rolled out based on Canary testing. K8s can be used to direct a small subset of traffic load (e.g., a small number of sessions) to the updated pods.

As shown in Figure 6, the packet core CNFs support the capability to program application load balancers for GTP, PFCP, Steram Control Transmission Protocol (SCTP) and Diameter traffic to direct a configured percentage of total traffic towards the updated pods.

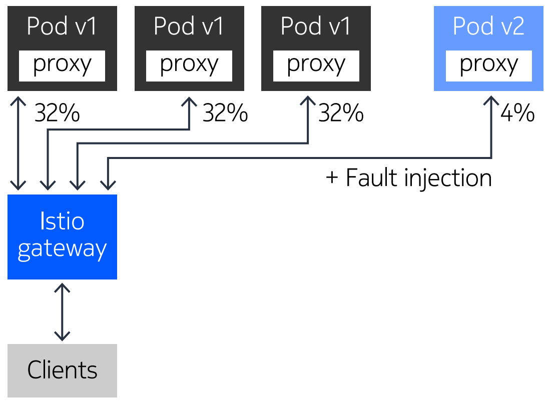
Figure 6. Programmable TCP/UDP LB for Canary testing



For HTTP traffic, Istio can be programmed with routing rules to send a small percentage of total traffic matching the rule to the updated pods (see Figure 7); this provides maximum flexibilty because the amount of traffic sent to udpated pods is indpendent of the number of updated pods.

It allows traffic from specific subscribers to be directed to the updated pods used for Canary testing. Istio itself can be used to inject faults for further testing. When the changes with updated pods are deemed to be functional, the service provider can adjust the LB and remove pods with old verions as needed.

Figure 7. Programmable traffic distribution with the Istio gateway for Canary testing



Security

Security is multidimensional. It includes image construction and delivery, pod/container runtime and many other deployment aspects that must be enforced throughout the life cycle of the CNF. Nokia expects that each service provider will implement a security policy that fits their own requirements and deployment model.

Nokia’s approach is to provide containers that meet the most fundamental requirements while maintaining flexibility to adapt to different deployment models.

A number of security considerations are applicable at the pod/container level:

- The ability to limit the use of the kernel API to a small set of secure system calls (seccomp)
- The use of application profiles to allow the administrator to limit an application's capability to access kernel and network resources (apparmor)
- The ability to run containers without privilege (no root access) to eliminate use of the system administration (sysadmin) capability and restrict the CNFs to a limited set of Linux capabilities (Nokia CNFs run with the lowest possible privilege to meet the required functionality and performance)
- The use of CNIs that implement namespace-based ingress and egress networking policy (e.g., Calico)
- The ability to detect and log container behavior to protect the platform from damaging activity (e.g., using sysdig Falco rules).

In a multi-tenant deployment, containers share the same host kernel, which effectively means the kernel is a single point of failure. This is due to the fact that all runtime vulnerabilities may potentially converge there. For example, if a container causes a kernel panic, everything is affected. The tools and practices in the previous list limit the attack surface on the kernel while promoting the design of safer and simpler applications.

In addition, service providers can make security more robust by adopting the following steps:

- Image verification and validation. Container images should be signed for authenticity and integrity, with the image itself scanned to determine if it includes forbidden software or outdated software (i.e., it is the latest release with security fixes for open source) and implements the correct use of the kernel APIs; these scans and validations are part of the Artifactory or the Docker Repository
- Maintaining the kernel to the latest security standards and updates
- Ensuring the correct usage of K8s features and service mesh to secure resources that are not namespace based.

For performance, Nokia's CMG CNFs need to tune certain kernel parameters to increase network read/write buffers, inter-task communication messages and queue sizes via sysctls. These sysctls parameters are name-spaced, meaning they can be set on the container namespace separately from the default (host) namespace.

The easiest way to set these parameters for a container namespace is to grant the container privileged mode. A pod can use a privileged init-container to set sysctl. An init-container runs just long enough to set the sysctl and then exits. The use of privileged mode is prohibited for shared multi-tenant clusters.

As an option, specific sysctls can be configured in the pod security policy by the service provider and explicitly allowed on a node via K8s configuration. The service provider can define a subset of nodes within the cluster with the sysctl performance settings needed for the CNF. The cluster can be configured with the K8s taints and tolerations feature to schedule high-performance pods on high-performance nodes. Application pods that do not require such performance can be scheduled onto normal nodes.

A pod's toleration can cause it to be scheduled on a node that will allow its required unsafe sysctl to be set.

Security best practices of Nokia CNFs

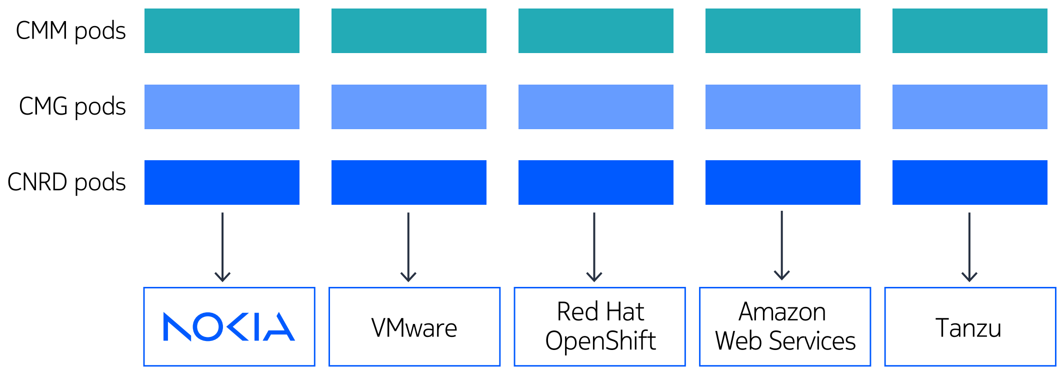
Nokia CNFs multi-dimensional security considerations and best practices include the following:

- They work with security best practices that limit the attack surface on the kernel.
- They adhere to security policies that are essential to run in multi-tenant shared clusters.
- They provide deployment flexibility to run in VMs or bare-metal and on dedicated or multi-tenant shared clusters.

Container management

As shown in Figure 8, Nokia CNFs can run on several K8s-based container-as-a-service (CaaS) platforms, including Nokia container services (NCS), as well as popular third-party CaaS platforms, including Red Hat Openshift, Vmware Tanzu and Amazon web services (AWS) Elastic Kubernetes Service (EKS).

Figure 8. Nokia CNFs integration with multiple CaaS platforms



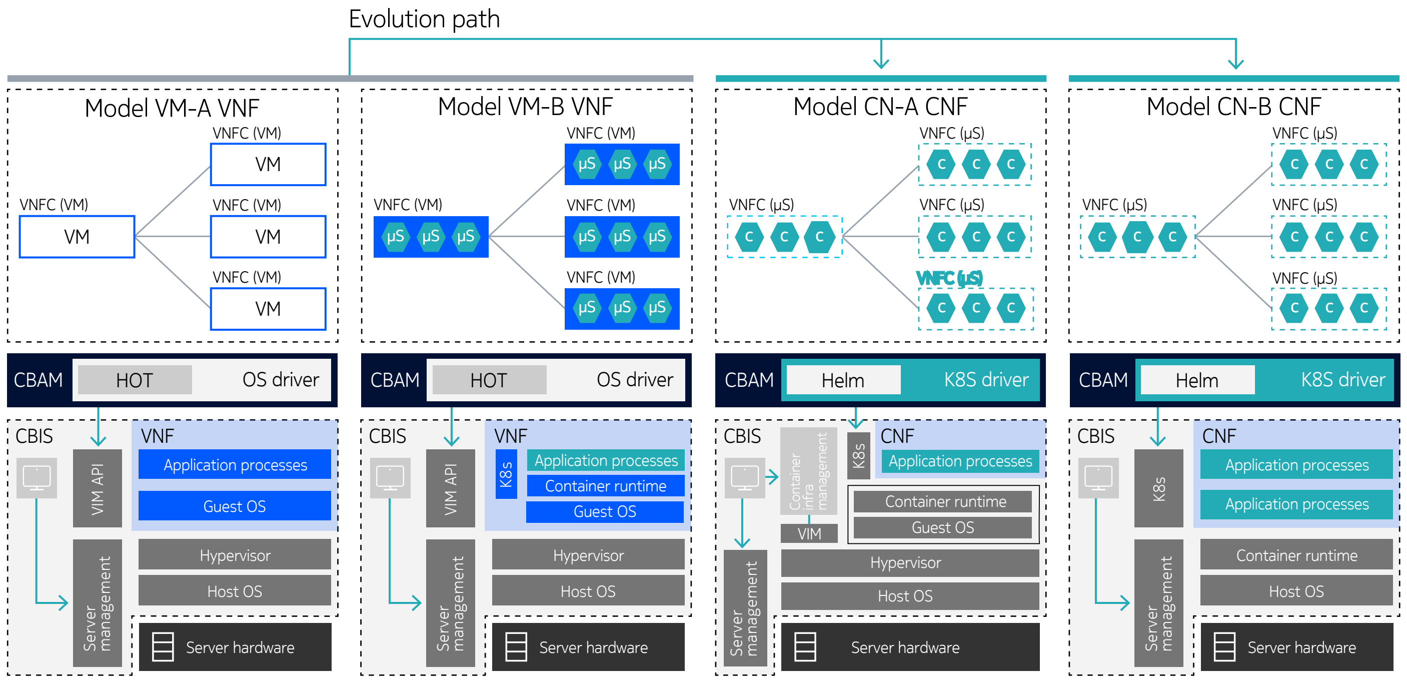
Nokia can also integrate with service provider's own K8s-based container platform. Container platforms need to support both containers-on-bare-metal and in VMs, provide flexible deployment choices with multiple network and storage plug-ins, enable multi-tenancy with configurable security policies, and provide automation hooks for deployment, LCM and configuration. For high-performance packet core CNFs, container platforms need to provide fundamental capabilities that include:

- CPU management policies to enable CPU pinning and isolation
- Support for allocating and consuming HugePages
- Support for the Multus CNI for attaching multiple network interfaces to pods
- The capability to use SR-IOV hardware on container platform nodes by enabling the attachment of the SR-IOV virtual function interfaces to pods
- Fast pod and node eviction for high availability and sub-second failover.

Cloud transformation: Delivery models

Figure 9 shows some typical steps in a migration from VM-based VNFs to CNFs that run on a K8s cluster. Most VNFs have started with one of the VM-based options on the left side of the figure and have been transitioning toward the right side of the figure.

Figure 9. Cloud transformation



The VNF models follow the classic ETSI network function virtualization (NFV) architecture. They are managed via a virtual network function manager (VNFM) such as the Nokia CloudBand Application Manager (CBAM), which interacts with an OpenStack virtual infrastructure manager (VIM) via Heat.

For the purposes of this application note, the Nokia CloudBand Infrastructure Services (CBIS)/OpenStack is shown but the VMware infrastructure is also applicable.

VNFs are delivered as set of quick emulator (QEMU) copy on write (QCOW2) images and a VNFM template package that includes Heat templates.

Model VM-A VNF

In model VM-A VNF, the VNF is managed as a set of application-specific VMs where virtual network function components (VNFCs) are VMs. Each VM runs its own guest OS, which may or may not be a Linux distribution. Application processes within the guest do not run within containers. The LCM of the VNF is performed on VMs.

A VNFC is the fundamental scalable unit of the application. Scale-out can be done by replicating one or more VNFCs along with any application processes within the VNFCs. Heat templates that build the VNFCs and their required networks provide instantiation, healing and scaling of the VNF.

Model VM-B VNF

In model VM-B VNF, the VNF is also managed as a set of application-specific VMs. However, the guest OS is a Linux distribution so that application processes can run within containers. This model requires a container runtime such as Docker to be embedded within a VM. K8s might also be present within a VNFC.

Similar to model VM-A VNF, the VM-based VNFC is still the unit of LCM. As with model VM-A VNF, healing and scaling is done by replacing and replicating VMs.

The VNF vendor provides Heat templates that can be used by VNFM to instantiate, heal or scale the application. There is no separate action required by the service provider to instantiate, heal or scale the containers within. Adding a VM means adding the containers present within the VM.

Model CN-A CNF

In model CN-A CNF, there are two layers of infrastructure: a VM infrastructure and a K8s management system. First, VM management creates a set of the resources that are captured in the form of VMs. The application runs on Linux as a set of containers. The application containers run on a set of K8s worker nodes that are implemented as a set of VMs running a guest OS that is some flavor of Linux.
The application is delivered separately from the K8s infrastructure and is managed as a set of pods.

This model can support multi-tenancy whereby multiple CNFs share a single cluster. The service provider can also choose to deploy a single CNF cluster. The LCM of the K8s cluster and that of the CNF is decoupled.

To scale out the CNF, the service provider must ensure that appropriate capacity exists on the cluster and may need to scale out if required. Similarly, before deciding to reduce the size of a cluster, a service provider must assess the impact on CNFs within the cluster. The LCM of the CNF is performed via Helm.

Model CN-B CNF

In model CN-B CNF, the CNF runs as a set of pods and is managed via Helm as in model CN-A CNF. The cluster is comprised of bare-metal nodes rather than VMs. The LCM of the cluster is decoupled from that of the CNF as in model CN-A CNF. The service provider can choose to deploy CNFs within a multi-tenant cluster or by dedicating a cluster to a CNF.

As shown in Table 4, there are a number of advantages and disadvantages of the two different CNF-based deployment models.

Table 4. Advantages and disadvantages of different deployment models

Model CN-A CNF	
Advantages	
Uses an existing OpenStack infrastructure	If a service provider has a set of OpenStack-based VNFs, this option allows the provider to run the CNF alongside those VNFs. Familiar LCM operations can be applied to the K8s cluster.
Easier to deploy a dedicated cluster for a CNF	By including a VM layer, model CN-A CNF can be used to build clusters that require smaller nodes than a unit of a physical server. This is useful if the service provider wants to deploy CNFs in dedicated clusters. Dedicating a cluster to a CNF can be used to avoid multiple CNFs sharing a single kernel instance or requiring different cluster-level security settings.
Disadvantages	
Less efficient use of resources	The VM layer requires more memory and typically requires hard allocations of resources (such as CPUs) between VMs. There are multiple networking layers to consider. The VM layer may use a Neutron plug-in, which is independent of the CNI plug-ins used within the cluster.
Decoupled LCM	It is more likely that CN-A CNF is used to provide dedicated clusters per CNF. As a result, cluster and CNF LCM must be coordinated.
Model CN-B CNF	
Advantages	
More efficient use of resources	With only one OS to maintain, no hypervisor and no OVS-DPDK means not needing to dedicate vCPUs. Networking is simplified because there are no longer independent node and container network layers.
Multi-tenant clusters	A bare-metal cluster is more readily multi-tenant and allows the full use of host resources to its tenants. A service provider can deploy a large cluster and ease concerns of separate LCM of cluster and CNFs. The cluster size can be set to accommodate the scale-out of multiple CNFs.
Disadvantages	
Multi-tenant clusters with kernel sharing	CNFs that share a node share that node's kernel. The service provider must address security concerns because containers do not offer the security isolation of VMs. K8s provides security functions for multi-tenant clusters such as namespaces and network policies. Furthermore, the service provider can disallow tenants from sharing nodes via the K8s scheduler (i.e., via the taints and tolerations feature).

Conclusion

Nokia's Cloud Packet Core can be deployed using CNFs. Nokia's CNFs support both EPC and 5G Core functions. Combined functions are provided to support EPC- 5G Core interworking. Nokia CNFs are fully integrated with K8s and leverage Helm and K8s service provider frameworks for automated deployment, update and scaling across different cloud deployment environments, such as private or public cloud.

With state-efficient design and the right level of software disaggregation, Nokia CNFs support a micro-services architecture without adverse impact on transaction rates or throughput. The Nokia PGW-u/UPF CNF can be scaled on demand up to 1Tb/s. For distributed MEC deployments, a minimum-sized micro UPF variant is also supported as a CNF, using as few as three physical cores.

Fast (sub-second) recovery from K8s pod and node failures is supported with limited or zero standby resources.

The Nokia CNFs satisfy key operational networking requirements such as multiple network interfaces with routing isolation, preserving visibility of peer IP address for cluster external and internal access, compatibility with commonly used CNIs, and readiness for IPv6-only or dual-stack clusters.

Nokia CNFs comply with security best practices and security policies necessary to run in multi-tenant shared clusters. The CNFs support both container-in-VM and bare-metal deployments. They can run on both the Nokia-provided CaaS platform as well as popular third-party K8s-based container platforms.

To learn more about Nokia's cloud-native packet core, visit our Cloud Packet Core solution web page

Abbreviations

3GPP	3rd Generation Partnership Project
AA	application assurance
AAA	authentication, authorization and accounting
AMF	access and mobility management function
API	application programming interface
BGP	Border Gateway Protocol
CBAM	Nokia CloudBand Application Manager
CBIS	Nokia CloudBand Infrastructure Software
CI/CD	continuous integration/continuous delivery
CMG	Nokia Cloud Mobile Gateway
CMG-c	CMG control-plane
CMG-u	CMG user-plane
CMM	Nokia Cloud Mobility Manager
CNF	cloud-native network function
CNI	container network interface
CNRD	Nokia Cloud Network Resource Director

CPU	central processing unit
CUPS	control and user plane separation
DevOps	Development and Operations
DPI	deep packet inspection
EPC	Evolved Packet Core
ePDG	evolved packet data gateway
ETSI	European Telecommunications Standards Institute
GGSN	Gateway GPRS Support Node
gNMI	generalized Network Management Interface
GPRS	General Packet Radio Service
gRPC	generalized Remote Procedure Call
GSM	Global System for Mobile Communications
GTP	GPRS Tunneling Protocol
HTTP	Hypertext Transport Protocol
IPVS	IP virtual server
JSON	JavaScript Object Notation
LAN	local area network
LB	load balancer
LCM	life cycle management
LLB	line load balancer
MEC	multi-access edge computing
MME	mobile management entity
N3IWF	non-3GPP interworking function
NAT	network address translation
NRF	network function repository function
NSSF	network slice selection function
OAM	operations, administration and maintenance
OPEX	operating expenses
OS	operating system
OVS-DPDK	open v-switch-data plane development kit
PCMD	per-call management data
PDN	Packet data network
PFCP	Packet Forwarding Control Protocol

PGW	packet data network gateway
PGW-c	packet data network gateway-control
PGW-u	Packet data network gateway-user
RAN	radio access network
REST	representational state transfer
SBA	service-based architecture
SGSN	serving GPRS support node
SGW	serving gateway
SGW-C	serving gateway-control
SGW-U	serving gateway-user
SMF	session management function
SP-GW CP	serving gateway PDN gateway control plane
SP-GW UP	serving gateway PDN gateway user plane
SR-IOV	single root-input/output virtualization
SSH	Secure Shell
TCP	Transmission Control Protocol
TWAG	trusted wireless access gateway
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunications System
UPF	user plane function
VM	virtual machine
VNF	virtualized network function
VNFC	virtual network function component
VNFM	virtual network function manager
VoLTE	voice over long term evolution
XML	extended markup language
YAML	yet another markup language
YANG	yet another next generation (data modeling language)