Accelerating Cloud Native in Telco

Challenges of Cloud Native Telco Transformation today and how to overcome them - A CSP perspective

v1.0 - November 3, 2023

Preamble

The document presented here is a product of the initial joint work of several Communication Service Providers (CSPs) who are active in Cloud Native Computing Foundation (CNCF)'s Cloud Native Network Function Working Group (CNF WG), NGMN Alliance, and projects like LF Europe's Sylva and LFN Anuket. It is a draft that has been published with the goal of inviting feedback from other CSPs and motivating discussion and improvements in the broader telecommunication industry. We are hoping that through public discourse we can make the document more complete, relevant, and ready for final release. If you would like to contribute to the discussion and document please feel free to open an issue or create a pull request.

Introduction

The recently published Cloud Native Manifesto from Next Generation Mobile Networks (NGMN) Alliance does an excellent job outlining the vision and target picture for cloud native telecommunication networks. The transformation towards a cloud native production model has already commenced in many Communication Service Providers (CSPs). Practical challenges and pain points on this journey, which hinder progress towards the target expressed in the NGMN Cloud Native Manifesto, have been identified and are being felt. These hindering aspects are especially prominent in the CSPs which are already taking practical transformation steps and are trying to closely follow the vision described.

We, the group of CSPs gathered around Cloud Native Computing Foundation (CNCF)'s Cloud Native Network Function Working Group (CNF WG), live on the frontlines of this transformation and have gathered valuable experience in this regard. We firmly believe that to attain the envisioned outcome, the entire industry needs to work together to align around key strategic and operational principles. Besides building a sound understanding of what it would take for the transformation of CNFs to become cloud native, it's also important to emphasize the ecosystem that would support that evolution.

CNF suppliers have not been able to comply with the cloud native and openness requirements of CSPs and one reason for that closed architecture and solution approach is revenue from professional service that allows them to create business models on CNFs deployment, automation, and operationalizing work. There is a need to evolve into cloud native architecture, largely based on 12-factors for CNFs (see Annex 1 and Reference 5), that can be supported by changing existing commercial models that could greatly benefit CSPs and vendors alike to create a new equilibrium in win-win. Vendors must provide open APIs, clear documentation, and cloud native architectures and implementations that empower CSPs with self-service capabilities in the cloud ecosystem. This will help drive agility and innovation, and reduce Opex costs within CSP. CNF vendors can monetize the value of openness to evolve business models that move away from closed solutions and professional services.

As such, we want to highlight major challenges facing cloud native telco transformations today and formulate principles and requirements that will aid the industry in achieving alignment and overcoming obstacles.

Challenges in Cloud Native Telco Transformation Today

Pre-Validation. Historically, Network Functions have been developed and pre-integrated with well-defined infrastructure, which was known in advance. That pre-validation was done by a Network Function vendor and the system was delivered as a validated/certified bundle together with performance and stability guarantees. In the cloud native world, there are too many permutations which makes it impractical to follow the traditional certification path. However, Cloud Native Network Function (CNF) vendors are still sticking to it by picking a small number of opinionated infrastructure flavors (different from vendor to vendor!) to pre-validate against, making any infrastructure outside this selection too complicated, too costly, and too slow to deliver for CSPs. This creates problems in the adoption of those CNFs as CSPs generally prefer to each have a unified cloud native infrastructure layer, which they are free to choose and often it can differ from the opinionated infrastructure already validated by the CNF vendors.

Adaptations. CNFs are typically delivered as a collection of artifacts such as YAML, Helm charts, and container images. These artifacts are intended for deployment in CSP's cloud native environment. However, every CSP has somewhat different rules, policies, security standards, API versions, and approaches to lifecycle operations (e.g. use of NFVO capabilities, GitOps pipelines, etc.). Due to that, it is often not possible to deploy the CNF directly in any environment in a consistently replicable way, but it requires some adaptations. That shall normally not pose a problem, as most of these adaptations can be performed in deployment configuration often in YAML files, by either CSP's DevOps team or the vendor's delivery personnel. Nonetheless, we often encounter situations where CSPs are not allowed to perform such adaptations (under threat of losing support if doing otherwise) since these artifacts are part of the release and could be adapted only in new release delivery or through the custom change request. As a result, this situation often leads to a frustrating cycle of discussions and significantly hampers the CNF onboarding process.

Validation. This step did not really exist in the previous scenarios due to reliance on pre-validation and pre-integration. Due to the number of permutations found in cloud native ecosystems, pre-validation has limited value. Only validation of CNFs on CSP premises with CSP's flavor of cloud native infrastructure and its specific integrations has high relevance and value for concluding if the CNF can be deployed and promoted to production. Today, we still see that many CNFs are not ready to be validated in the local CSP environment and rather insist on conformance with the pre-validation stack. This practice is unsustainable and requires a fresh and flexible local validation approach. Automation (Continuous Testing) is especially important when validating frequently released cloud native applications and checking conformance with frequently updated cloud platforms.

Automation. In the ongoing pursuit of end-to-end orchestration and deployment, and configuration automation, the Telco industry has devised numerous frameworks, models, and standards. Some have achieved considerable success, while others have seen varying levels of adoption. However, the cloud native ecosystem, with a focus on GitOps practices, is propelling CNFs toward more advanced and automated models.

Many CNFs are still reliant on manual artifact deployment and are rooted in traditional telco methods, such as NETCONF and YANG for configuration management. These practices pose significant challenges for CSPs aiming for a fully automated CNF lifecycle. Moreover, the ETSI standard follows an imperative top-down approach, often characterized as "fire and forget". This approach doesn't readily support reconciliation and depends on orchestration entities operating externally to Kubernetes "out-of-band." Even when the CNFs are following the Kubernetes native approach, we face challenges with the quality of artifacts like Helm Charts which are not generalized nor easily customizable, as well as with divergent configuration schemas. This all creates further complexities in the transition to the declarative and GitOps-driven automation models prevalent in the cloud native ecosystem.

Dependencies. Cloud Native applications shall be completely separated from the underlying infrastructure, especially hardware. Nevertheless, in practice today there are often very hard dependencies present, be it on specific technologies, or on specific vendor products. The CNFs often require a specific hardware type or brand (e.g. CPU, NIC) and do not allow for flexibility supported by local validation. CNFs are not able to run on any CNCF-certified Kubernetes distributions. Even if the dependency is fulfilled there is a lack of attention to those dependencies. For example, a CNF can break because the firmware of the network card was updated, which shows that pre-validation of a particular combination of dependencies was not performed and proactive CNF update measures have not been taken. This creates a lot of operational burdens and negatively impacts KPIs for cloud native CNF deployments.

Lifecycle. Kubernetes occupies a central place in cloud native infrastructure by following the paradigm of ephemeral resources and relying on "rolling upgrades" to deploy changes. This paradigm is applied in both the lifecycle management of applications and the cloud infrastructure. As a consequence, for example, Kubernetes cluster nodes can ordinarily have rather short uptime of several days to several weeks. This is in contrast with the traditional carrier-grade-driven focus on the uptime of individual system elements. Although large parts of CNFs do not have problems with that cloud native lifecycle approach, we are experiencing that many CNFs have some elements which are rather sensitive to it. CNFs that are not resilient to ephemeral Kubernetes nodes (e.g. crashing when cluster scaling or upgrade occurs) lead to service interruptions during the lifecycle operations, which is not acceptable, such as on SCTP pods due to s1 interface interruptions which leads to unsupported ISSU. It is often the case that Pod Disruption Budgets are not properly set, the consequence of which is either service interruption or lifecycle operations being blocked.

Tracing. The more cloud native transformation progresses, the more challenging it is to perform the e2e protocol tracing using traditional mechanisms based on tapping points on the network fabric level. The reason for that is the dynamic nature of cloud native workloads including CNFs. When several CNFs run within one large data center the pods could be distributed to any of the servers in any of the racks. This means that particular communication can go via multiple network elements and as the traditional tapping setup is not configurable or capable enough,it is practically impossible to create reasonable port mirroring to capture the traces. In many cases, the CNFs or their microservices run on the same node and their communication does not go via data center network fabric at all. Furthermore, encryption and mTLS became a de-facto standard for CNFs, so even if tapped, network traffic can not be really analyzed and so the purpose of tracing can not be fulfilled. Cloud native tracing mechanisms (e.g. eBPF) are unfortunately not helping here as most of the telco-relevant traffic goes via secondary interfaces (Multus) which are not covered by vanilla Kubernetes.

Architecture. We are witnessing that there are still CNFs that are in their architecture exhibiting properties of Virtualized Network Functions (VNFs). For example, we see the "pinning" of Pods to particular NUMA nodes, or worse to specific cluster nodes. We also still see 1+1 redundancy models for Pods within the cluster instead of N+1. Although it is technically possible to run such Network Functions on cloud native infrastructure, this increases the burden of operating them and risks having a negative impact on service quality, as small disruptions which are normal in cloud native infrastructures result in problems within the CNFs. Furthermore, the scalability of today's CNFs is still sub-optimal. In many cases, it still relies on vertical scaling and manual interventions. Sudden increases in demand cause performance degradation and even downtime if the system is not dimensioned in advance for that peak load.

Security. In the experience so far we have noticed that CNFs, in their default setup, have quite a relaxed posture when it comes to dealing with cluster security-relevant aspects like Roles, ClusterRoles, privileged access, cluster node level access, and similar functionalities. We frequently observe that the principle of least privilege is not consistently followed and that Roles frequently require rights for everything ("*") and ClusterRoles are used without real need. CNFs sometimes use problematic practices (such as hostPath mounting to write their logs, hostPorts for communication, privilege escalation, running containers as root, managing the configuration of the node networking stack, and performing dangerous sysctl calls), none of which are allowed in a properly hardened environment. It looks like such CNFs assume that the infrastructure can be consumed from a cluster admin perspective without any restrictions, which in realistic circumstances is never the case. Such expectation could be reasonable in a combo/silo package where CNF and infrastructure come together from one hand as a managed package. However, in other cases, CNFs are usually "guests" on the infrastructure and as such must have appropriate security imposed restrictions and limitations.

Resilience. In contrast to traditional expectations within the telecommunications domain, one important property of the cloud infrastructure is that it is imperfect. Cloud infrastructure does not give strict performance and stability guarantees. However, it offers mechanisms to applications through which they can achieve a high degree of resilience to this imperfection. Yet, repeatedly we experience the cases in which the imperfection of cloud infrastructure has a severe impact on the CNFs to the point that complete re-deployment is the only viable solution. Instances of such impact include CNFs completely crashing due to the ephemeral storage on the Kubernetes cluster reaching full capacity with logs of that CNF or because write operations to persistent volumes could not be performed for a short period of time. This state is unsustainable as such events and situations are going to happen in the cloud environment all the time. Therefore, applications including CNFs, which aim to run in the cloud, have to account for such events in their design and utilize cloud native mechanisms to maintain robustness consistently or facilitate automated recovery.

Principles and Requirements to Enable Progress in Cloud Native Telco Transformation

These principles and requirements address the systemic challenges listed above. They are intended to serve as the guardrails which every industry/ecosystem player, who aims to engage in cloud native telco transformation together with CSPs, shall respect. In this way, setting the minimal set of conditions, we want to eliminate main blockers and fragmentation, which is a chronic condition in the telco ecosystem, and thus accelerate the path towards benefits for all. We aim to work with existing industry initiatives such as Anuket RA2 / GSMA NG.139 Kubernetes Reference Architecture to ensure these principles are developed in collaboration with existing communities, and included in existing well-established publications.

1. **Pre-validation.** The pre-validation of CNF needs to be performed against a reference that is common for all players, which is the upstream Kubernetes and components from the CNCF ecosystem.
    1. Each CNF shall be validated, including the implementation of any adaptations that may be required, within 3 months of the Kubernetes

release.

   2. Every CNF shall certify adherence to cloud native principles and best practices using CNF Test Suite (https://github.com/cncf/cnf-testsuite) as a vendor-neutral validation tool.
   3. Pre-validating CNF against additional commercial distributions such as OpenShift, Tanzu, Rancher, and Hyperscaler solutions is a plus, but not mandatory.

2. **Adaptations.** CNF artifacts (e.g. YAML manifests, Helm charts, NFVO descriptors) shall become open for CSP's DevOps or vendor's delivery team to adapt them to align the deployments to the local specifics of CSP (e.g. Policy, RBAC, Compatibility) without special Change Request or complex processes involving R&D.
   1. Validation of such adapted CNF deployment shall be performed on CSP premises.
   2. Given the successful validation vendor support for such deployment shall be granted.
   3. CNFs shall be modular, microservice-based, open applications and not big "black" boxes.
   4. CNF vendors shall provide all artifacts (Helm charts, CRDs, Operators) passing strict linter in open documentation and provide APIs, instead of encapsulating them in proprietary tools.
3. **Validation.** CNFs shall be delivered with a series of automated tests that can be used to validate the CNF on the spot in CSP's context.
   1. This validation shall count as only relevant one, preceding any prevalidation or lack of it.
   2. The validation shall assure that all artifacts are passing strict linters to prove that portability is assured.
   3. It shall serve as a condition for support and SLA.
   4. The validation shall be a continuous process and shall be instantly done on any change be it on CNF or on the infrastructure side.
4. **Automation.** CNF deployment and configuration shall be fully automated ("everything as a code") and done exclusively with declarative cloud native mechanisms like GitOps.
   1. Mainstream open source deployment tools from CNCF ecosystems, like FluxCD or ArgoCD, shall be supported per default.
   2. All configurations shall be done via Configmaps and/or similar cloud native constructs (eg. Kubernetes Resource Models)
   3. CNF is allowed to use traditional telco mechanisms internally as a transition step, however, that should be fully encapsulated and abstracted away.
   4. Microservices should be loosely coupled with NO tight dependency on each other, to ensure scalability and ease of deployment, e.g. without the need to wait for NETCONF day-1 configuration till further microservices get deployed.
   5. Artifacts are delivered via OCI-compliant repositories.
   6. The CNF LCM should be described declaratively and support con-

   tinuous intent-based deployments for example IP address assignment during deployment.

   7. New released software version (CNF/microservices) includes machine-readable code to run health checks.
   8. Release notes and impact reports should be included as machine-readable code in every published release.
   9. CSP-internal automation pipeline shall be allowed to hook into the vendor software delivery solution (e.g. to subscribe to CNF releases).
   10. Artifacts delivered with CNFs (e.g. Helm charts) shall be customizable for efficient multi-purpose deployments.
   11. CNF configuration schemas have to follow the standards that shall be aligned among CSPs and vendors.

5. **Dependencies.** The CNFs shall be completely independent from underlying infrastructure platforms.
   1. Alternatively it shall equally support all the mainstream available hardware.
   2. Local validation, not product policy, shall give an answer if CNF can run as expected on particular hardware or not.
   3. In case of hard technical dependencies, the vendor of such CNF shall timely pre-validate its CNF against all new releases of hardware-related software (e.g. drivers, firmware) and proactively adapt the CNF to avoid the negative impact of dependency in production.
   4. Application resource requirements must be configured declaratively. The application must not be statically configured to utilize specific resources including devices, nodes, or machines
   5. When performance is a requirement, the application should utilize open standards for adapters where possible at all levels they are available. For example support for multiple CNIs vs a single CNI. Support for SR-IOV and…
   6. To be able to deploy on any CNCF Certified Kubernetes distributions
   7. CNFs make use of standard APIs for infrastructure and platform capabilities. For example Service Mesh Interface, Ingress, CNI, etc.
6. **Lifecycle.** CNFs have to be constructed in a way to fully tolerates graceful cluster rolling upgrade procedures without blocking them and without service interruptions.
   1. CNF shall implement N+1 redundancy mechanisms.
   2. CNF shall rely on mechanisms around PodDisruptionBudgets to secure the conditions for itself to run uninterrupted during lifecycle procedures.
   3. The application must function properly if it is rescheduled to other nodes without interruption of its services (e.g. move user sessions without interruptions)
   4. Upgrade procedures on CNF shall also follow rolling-upgrade principles and shall be done in-service
7. **Tracing.** The CNFs shall be instrumented to emit the protocol tracing data directly from their microservices to the configurable targets

   (e.g. application-level tracing https://opentracing.io/)
   1. These traces shall be sufficient for typical e2e analysis that is done with standard telco tools such as NetScout, Gigamon, Polystar etc.

8. **Architecture.** The CNFs need to be architected in line with 12-factors for CNF compliance with cloud native (Annex 1 and Reference 3) and in a way not depend on any particular cluster node or reasonably small group of cluster nodes.
   1. Microservices should be small and independently deployable units of functionality.
   2. Any common functionality (e.g. observability, access management) should be provided as a centralized service that can be reused. CNFs should share databases, load balancers, business logic, and common services and become fully disaggregated.
   3. A microservice should have a single, well-defined responsibility, and should only communicate with other microservices to accomplish tasks.
   4. The data-processing logic should be in the microservices themselves, rather than in a centralized hub, to ensure scalability and reliability.
   5. Microservices should be designed to dynamically scale up or down in response to changes in demand, to ensure that the application remains responsive and available using “Horizontal Pod Autoscaling” (HPA).
   6. Each microservice should log information and expose metrics about its performance and usage, which can be used to identify and diagnose issues.
   7. CNFs should expose their state (e.g. health) in a cloud native way
   8. CNFs can share databases, load balancers, business logic, and common services and become fully disaggregated.
   9. CNF has to tolerate automatic scaling at the node and container level by the Kubernetes orchestrator.
   10. CNF has to support self-healing.
9. **Security.** In order to run in a generic cloud native environment, CNFs have to strip down their expectations and require exactly the minimum rights that are needed for functioning.
   1. Any practice that poses the risk such as usage of hostPaths, privilege escalations, root containers, etc. needs either to be eliminated or replaced with an alternative cloud native approach.
   2. The application must adhere to cluster policies enforced by the cluster manager including overriding its default policies
   3. *The application should follow the Principle of Least Privilege*
   4. RBAC definitions must declare the minimal set of rights needed for the application to function The application should not request open-ended / all rights in its RBAC definitions
   5. Applications should not require privileges to run including privileged pods and cluster roles
   6. Applications that require privileges must declare which components

    require privileges in both machine-readable and human-readable open formats
    7. The application must be isolated with Namespaces and not use the default namespace

10. **Resilience.** High-quality CNFs should be resilient to underlying infrastructure issues including complete failure, meaning that instead of completely failing, they note that something is wrong, log the errors, etc, and then return to full working order upon restoration of the underlying infrastructure service/resource.
    1. CNFs use cloud native principles (e.g. Kubernetes capabilities) when implementing resilience architecture without fully focusing on 3GPP (e.g. CNFs do not rely solely on robust infrastructure)
    2. CNFs are resilient to network corruption / poor quality network connection (e.g. packet drops, high latency, etc.).
    3. CNFs are resilient to complete loss of network connection.
    4. CNFs are resilient to poor-quality storage connection (e.g. high latency, read/write performance degradation, etc.)
    5. CNFs are resilient to complete loss of storage connection.
    6. CNFs are resilient to storage disks being full
    7. CNFs are resilient to CPU stresses
    8. CNFs are resilient to Memory stresses
    9. CNFs are resilient to the complete loss of underlying infrastructure resources (e.g. a node failure)

ANNEX 1

12 Factors for CNFs compliance to Cloud Native Principles

CNFs need to adopt these principles as well. CNFs need to be:

- Compatible: A cloud native approach allows applications and workloads to run anywhere. Ideally, CNFs should work with any certified Kubernetes product, even if we need to use container network interface (CNI) plug-ins or other extensions. CNFs should work on any CNI-compatible network that meets their functionality requirements. Network interfaces and CNI plug-ins create hardware dependencies and tie them to specific infrastructure. A wise, but far-reaching, approach would be to develop all networks and I/O acceleration purely in software. Broader adoption of this software separation might take years to mature. In the short term, we can improve the automation to maximize the agility of the applications.
- Stateless: Cloud native applications need resiliency to quickly fail and recover elsewhere in the cloud. The legacy approach uses local storage, which makes network functions heavy and slow. To improve workload mobility, we need to store the state in custom resource definitions (CRDs) or a separate database.
- Secure: At its most basic, this principle says “CNFs must run unprivileged”; in other words, the CNFs must be deployable on modern cloud

platforms without the need for root administrative privileges. In the cloud native approach, multiple CNFs must be able to run concurrently on the same hardware. During the transition from PNFs to VNFs, applications required root-level access to Linux, as if they were running on dedicated servers. The same trend continued in the transition to CNFs. We need to remove this dependency on root access by redeveloping the application code to use more modern, cloud native security protocols.

- Scalable: Cloud native telco applications need to support horizontal scaling (across multiple machines) and vertical scaling (between different machine sizes). We want to optimize cost and performance by starting with a very small application and growing it as needed. This model yields efficiency and agility.
- Configurable: Open configuration offers telcos the control and freedom of DevOps tools to create and manage services. This should happen via custom resource definitions (CRDs) and operators, or other declarative interfaces.

- Observable: Cloud native applications need an Open Metrics interface that Prometheus and other monitoring tools can use. Kubernetes needs to access performance metrics that support container-level resiliency features. Analytics applications can process these metrics and suggest configuration changes to improve utilization and performance.
- Portable: Applications that can run on multiple clouds are more agile. CNFs must be able to declare their platform requirements without implying a specific implementation. The cloud fulfills those requirements, making network functions oblivious to the underlying cloud offering. This maximizes portability between execution environments.
- Installable and Upgradeable: The use of CRDs, operators, and declarative configurations gives flexibility and ease in the deployment and upgrade of CNFs. Automation tooling can track and validate the installation and upgrade processes, with rollbacks supported if needed.
- Parity across environments: Cloud native applications need to minimize divergence between development and production, enabling continuous deployment for maximum agility. Telcos can deliver features and upgrades faster by implementing DevOps practices over a CI/CD pipeline.
- Open: Cloud native applications need to be orchestrated, run as a service, and expose themselves via RESTful interfaces. Nephio will achieve this goal by enabling third-party automation tools to reconfigure the application and achieve any type of orchestration use case.
- Traceable: Cloud native applications need to support real-time troubleshooting through Open APIs with telemetry-compatible tracing.
- Loggable: Cloud native applications need to support uniform logging for consistency and access to network-wide logs.

ACKNOWLEDGEMENTS

Special acknowledgments go to the following Communication Service Providers, who have contributed to this whitepaper:

- Bell Canada
  - Daniel Bernier, Technical Director
  - Roger Lupien, Sr. Mgr - Enterprise Architecture, Cloud Transformation
- Charter Communications
  - Mohammad Zebetian, Head of Cloud, Network, Edge, and Infrastructure Architecture
- Deutsche Telekom Technik
  - Vuk Gojnic, Cloud Native Telco Platform Lead
  - Nathan Rader, VP Service and Capability Exposure
- DNA Oyj
  - Johanna Heinonen, Development Manager
- Orange
  - Philippe Ensarguet, VP of Software Engineering
  - Guillaume Nevicato, Network Anticipation & Research Manager
- Swisscom
  - Ashan Senevirathne, Product Owner Mobile Cloud Native Orchestration
  - Josua Hiller, Product Manager Mobile Data Services
- TELUS
  - Andrei Ivanov, Principal Technology Architect
  - Sana Tariq, Ph.D - Principal Technology Architect | Cloud and Service Orchestration
- Vodafone
  - Tom Kivlin, Principal Cloud Architect
  - Riccardo Gasparetto Stori, Principal Cloud Architect

Editing and facilitation acknowledgments:

- Taylor Carpenter, Vulk Coop Partner & CNF WG Co-Chair
- Lucina Stricko, Vulk Coop Partner & CNF Certification Maintainer

REFERENCES

1. Cloud Native Networking principles whitepaper - https://networking.cloud-native-principles.org/cloud-native-networking-preamble
2. NGMN
   - NGMN publishes Cloud Native Manifesto - https://www.ngmn.org/highlight/ngmn-publishes-cloud-native-manifesto.html
   - Cloud Native Manifesto "An Operator View" (PDF)
3. The Twelve-Factor App - https://12factor.net/
4. X-Factor CNFs - https://x.cnf.dev/
5. On the road to public cloud 5G networks – Nephio

6. Anuket RA2 - Kubernetes-based Reference Architecture