AWS Whitepaper

ETSI NFVO Compliant Orchestration in the Kubernetes/Cloud Native World

ETSI NFVO Compliant Orchestration in the Kubernetes/Cloud Native World: AWS Whitepaper

Table of Contents

ETSI NFVO Compliant Orchestration in the Kubernetes/ Cloud Native World

Publication date: **October 25, 2022** (*Document revisions*)

Abstract and introduction

This whitepaper examines the network functions virtualization (NFV) orchestration requirements and specification as put forth by the European Telecommunication Standard Institute (ETSI), in the context of the Kubernetes way of orchestrating services and applications. This whitepaper explores the interplay between the ETSI components and Kubernetes/Amazon Elastic Kubernetes Service (Amazon EKS) framework by examining requirements that can map from one framework to another, and outlines some of the gaps in mapping others. We also examine evolving business and operational needs, and importance of a true cloud-native orchestration framework to address those needs. We further provide a high-level architecture of a cloud-native orchestrator that can be implemented cost-effectively and reliably by some of the AWS cloud services.

This whitepaper is aimed at Communication Services Providers (CSPs) and Independent Software Vendors (ISVs).

Are you Well-Architected?

The AWS Well-Architected Framework helps you understand the pros and cons of the decisions you make when building systems in the cloud. The six pillars of the Framework allow you to learn architectural best practices for designing and operating reliable, secure, efficient, cost-effective, and sustainable systems. Using the AWS Well-Architected Tool, available at no charge in the AWS Management Console, you can review your workloads against these best practices by answering a set of questions for each pillar.

For more expert guidance and best practices for your cloud architecture—reference architecture deployments, diagrams, and whitepapers—refer to the AWS Architecture Center.

Introduction

In the past, before CSPs push toward virtualization, Independent Software Vendors (ISVs) often created their own proprietary ways of managing hardware and software, sometimes integrated, sometimes separate. While it was difficult to manage a multi-vendor network, methods for each

particular physically integrated functions (Physical network functions, or PNFs) worked reasonably well, and CSPs managed to create 2G and 3G networks that are still working in parts of the world.

Running network in such ways was expensive, and operators were often left with huge amounts of hardware that came with dedicated software functions, and became obsolete and required "forklift" upgrades.

When virtualization started being popular in Enterprise space, operators wanted to take advantage of hardware and software separation, and a network function virtualization (NFV) workgroup in the European Telecommunications Standards Institute (ETSI) was created in 2013. This workgroup looked at the concepts and issues of management of network functions and infrastructure in a virtualized world of virtual machines (VMs) and came up with a prescriptive reference called NFV Management and Orchestration (MANO). While the framework was often not implemented as-is, it did a great job in abstracting requirements and providing a unified framework to look at operations and management. As part of this, the group proposed using Topology and Orchestration Specification for Cloud Applications (TOSCA) as a declarative method of indicating network service and function requirements.

In the late 2000s and early 2010s, containers started becoming popular for software development and deployment. Containers are units of application that package code and all dependencies and can be run individually and reliably from one environment to another. Since an application consisted of many of these "container modules", also called Pods, a way to manage them was needed. Many different container orchestration systems were developed, but the one that became most popular was an open-source project called Kubernetes. Kubernetes ensured declarative interfaces at each level and defined a set of building blocks/intents ("primitives") in terms of API objects. These objects are representation of various resources such as Pods, Secrets, Deployments, Services. Kubernetes ensured that its design was loosely coupled, which made it easy to extend it to support the varying needs of different workloads, while still following intent-based framework.

TOSCA and Kubernetes both use YAML representation. The main difference is that ETSI MANO focused on specifying operations at each interface, whereas Kubernetes focused on the specification of the end state of the operation via APIs. In some ways they are similar. ETSI MANO with TOSCA templates started as a declarative framework in terms of Network Service Descriptor (NSD) and provided an abstraction for Virtual Network Functions (VNFs) independent of specific network functions. However, ETSI MANO with TOSCA then mapped this declarative framework to the imperative style of specification at lower levels. When ETSI NFV was developed, there were multiple competing virtualization frameworks such as OpenStack and VMware. Therefore, ETSI NFV had to create a common abstraction of them to specify architecture and requirements. While

the advantages of these abstractions were multiple, creating another layer of abstractions made specifications difficult to apply. Further, mapping declarative specification to imperative operations was also difficult.

Kubernetes kept the philosophy of defining an application independent of its function, but in contrast to ETSI MANO, it kept everything intent/API driven and defined a declarative way to define objects such as deployments. Once an application was defined in terms of its objects, Kubernetes ensured that the application was properly run, and ensured that the various parts of application could communicate easily. Determining and applying application specific configuration is beyond the scope of both ETSI MANO and Kubernetes, though they both provide some supporting constructs such as an element management system (EM), an operator's framework, or config files.

Network functions have additional requirements than commonly found cloud-applications. ETSI MANO framework was specifically designed to address these specialized requirements in the context of virtualized network functions. As we move to containerization, it becomes important to map the management and orchestration of CNFs to the ETSI MANO framework so that none of the requirements are missed and the containerized orchestration continues to work well without mandating changes in the business and operation support systems. With Kubernetes becoming the dominant common platform for container orchestration, an orchestration specification directly in terms of native Kubernetes terms would provide simplification and enable taking advantage of Kubernetes features and extensions to address MANO requirements. In the absence of such directly mapped specifications, multiple ways of interpretating the MANO requirements in the cloud-native/containerized contexts are possible. There have been multiple projects to address some of the challenges of defining common function orchestration system in containerized and cloud-native contexts. Linux open-source projects, such as ONAP, Nephio, and Anuket point to the need of such work.

In this whitepaper, we examine each component and requirement of NFV MANO architecture, and explore how they are normally addressed in a cloud native world. To keep the discussion closer to implementation, sometimes this whitepaper explores these requirements in the context of AWS services (such as Amazon EKS, Amazon Elastic Container Registry (Amazon ECR, and Amazon Elastic Compute Cloud (Amazon EC2) Auto Scaling groups) to avoid adding another layer of abstraction and ensure easy readability.

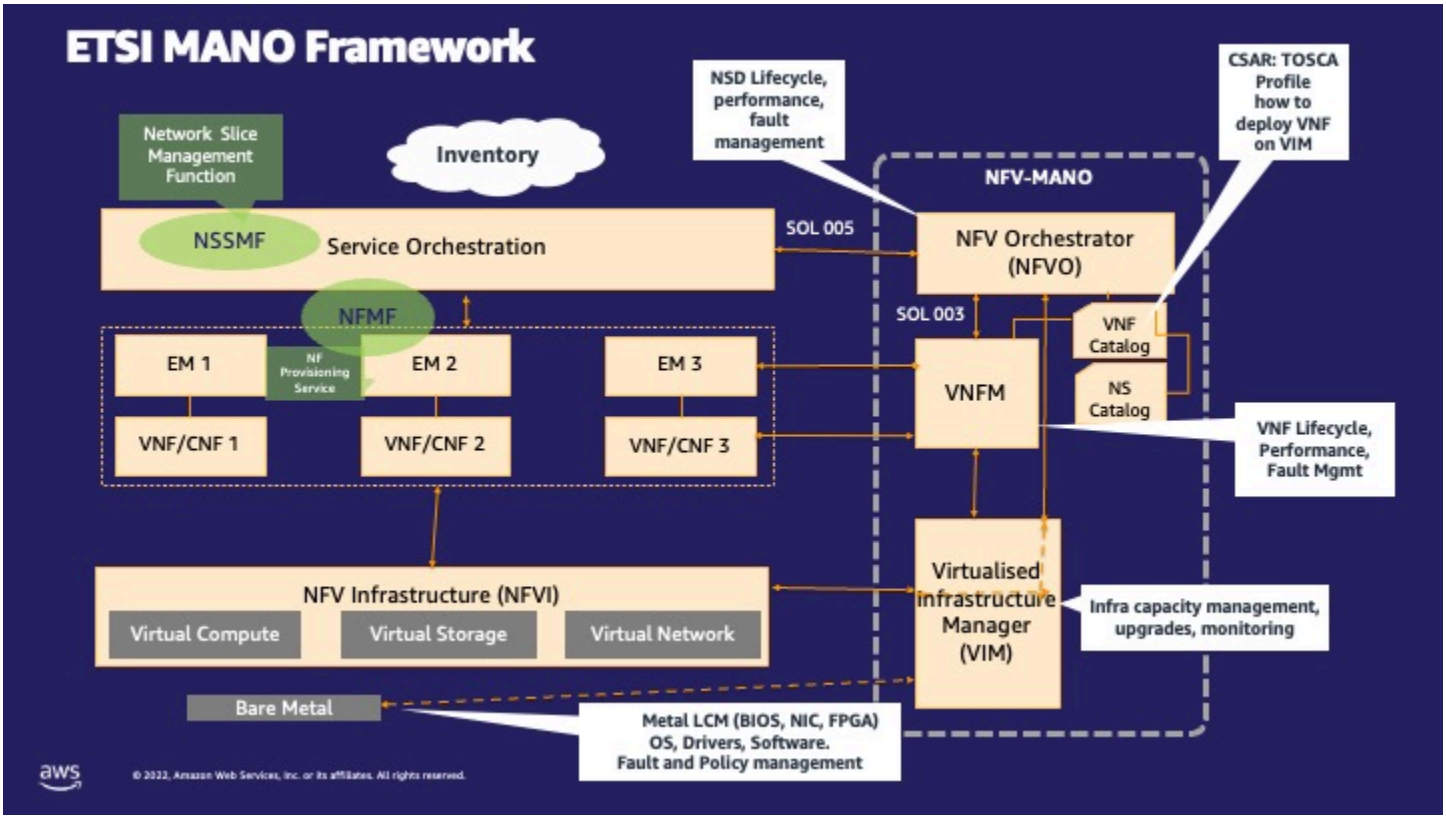
Mapping ETSI MANO to Kubernetes

In this section, we depict the high-level mapping of Kubernetes objects and functions to ETSI MANO architecture, with the following sections going in detail over each of the components:

- Virtual Infrastructure Manager (VIM)
- Virtual Network Function Manager (VNFM)
- NFV Orchestrator (NFVO) and its mapping to Kubernetes in detail

High-level mapping architecture

The traditional ETSI MANO framework was developed in the context of virtual machines (VM). The following figure indicates ETSI MANO architecture along with major functions performed by each of its components. Network slice management and its associated functions such as Network Slice Subnet Management Function (NSSMF) and Network Function Management Function (NFMF) are part of 3GPP specifications and are beyond the realm of MANO framework; however, they are shown in the figure to provide complete view of network and service management.



The traditional ETSI MANO framework as defined in the context of virtual machines along with 3GPP management functions.

Mapping VIM and VNFM to Kubernetes

In this section, we focus on the VIM and VNFM functional blocks of the ETSI MANO architecture and its functions, and then map to equivalent cloud-native constructs.

Following are the requirements that the MANO architecture puts on VIM:

- Manage lifecycle of virtual resources in an NFVI domain. Create, maintain, and tear down VMs from physical resources in an NFVI domain:
- Keep inventory of VMs associated with physical resources.
- Performance and fault management of hardware, software and virtual resources.
- Exposes physical and virtual resources to other management systems through northbound APIs.

In Kubernetes context, VIM is responsible for placing Pods and containers on nodes that can be bare-metal machine or a virtualization platform themselves. In this regard, the second and third requirements are out of scope of Kubernetes-based VIM in a cloud operational environment, specifically when we consider features such as automatic scaling node groups provided by managed Kubernetes solutions such as EKS. The functionality of the first and last requirements are also provided by services such as managed node groups and automatic scaling node groups in EKS.

VNFM performs the following tasks:

- **Manage life cycle of VNFs** — VNFM creates, maintains, and ends VNF instances, which are installed on the Virtual Machines (VMs) that the VIM creates and manages).
- **Fault, configuration, accounting, performance, and security (FCAPS)** — Management of VNFs. VNFM doesn't deal with application specific FCAPS, but generic FCAPS dealing with virtualization.
- **Scale up/scale down VNFs which results in scaling up and scaling down of CPU usage** — While the original VNFs were mostly scaled vertically, in the cloud model, there are two kinds of scaling: horizontal and vertical. Horizontal scaling is often the preferred mode of operation.

When applications are containerized (instead of virtualized), they are referred to as a containerized network function (CNF). In Kubernetes, containerized applications are run as Pods. Kubernetes

manages the lifecycle of Pods, and scales them up and down using changes in deployments as specified constraints in configurations such as minimum, maximum, and desired number of replicas.

Operation and management of infrastructure and platform is intrinsically provided by common cloud tools such as AWS CloudWatch, K8 config-maps. Due to scale, organization and security issues, one Kubernetes cluster is generally not sufficient for a telecom network operator, and a multi-cluster infra management solution is needed. There are many open-source and vendor provided solutions to manage multiple Kubernetes clusters. In AWS, this multi-cluster control plane management is provided by the EKS main control plane, which ensure that all clusters are healthy and scale as needed. The EKS dashboard provides a single pane of glass to manage multiple clusters, though each cluster does have its own API endpoint to which kubectl and helm commands are directed.

From the description and analysis for VIM and VNFM, it might seem like there is a good correlation between ETSI requirements and Kubernetes. However, ETSI MANO differs from the Kubernetes model in that the VNFM maintains a detailed view of deployed virtualization aspect of its associated VNFs, and exposes it northbound to NFVO. In Kubernetes, that information is not exposed. Kubernetes doesn't expose its internal workings and placement to the upper layers. The only way to control the operations is by clearly defined intents (which can include labels, tags, selectors, and so on), which forces the application to interact only through intents and object definitions.

When mapping ETSI MANO to Kubernetes and the cloud, it is important to re-interpret the ETSI MANO architecture to avoid sharing details of the lower layers, such as EC2 instances, to upper layers. With powerful functionalities such as managed node groups and automatic scaling constructs in AWS, the NFVO should not be affected by lower-level changes.

NFVO: functions and mapping

NFVO is the brain and heart of ETSI orchestration. Before exploring its mapping to Kubernetes, the following points are worth mentioning about VNFs:

- A VNF can't bring itself to service.
- Every VNF needs to be managed and orchestrated.
- A network service (NS) is composed of one of more VNFs. NS is onboarded on NFVO.
- An NS lifecycle is managed by NFVO.

- VNF lifecycle is managed by VNFM, which is responsible for bringing VNF into existence following the templates in VNF descriptor.
- VNF scaling can be invoked by NFVO, by element manager (EM), or the VNF can ask to scale itself based on VNF load.

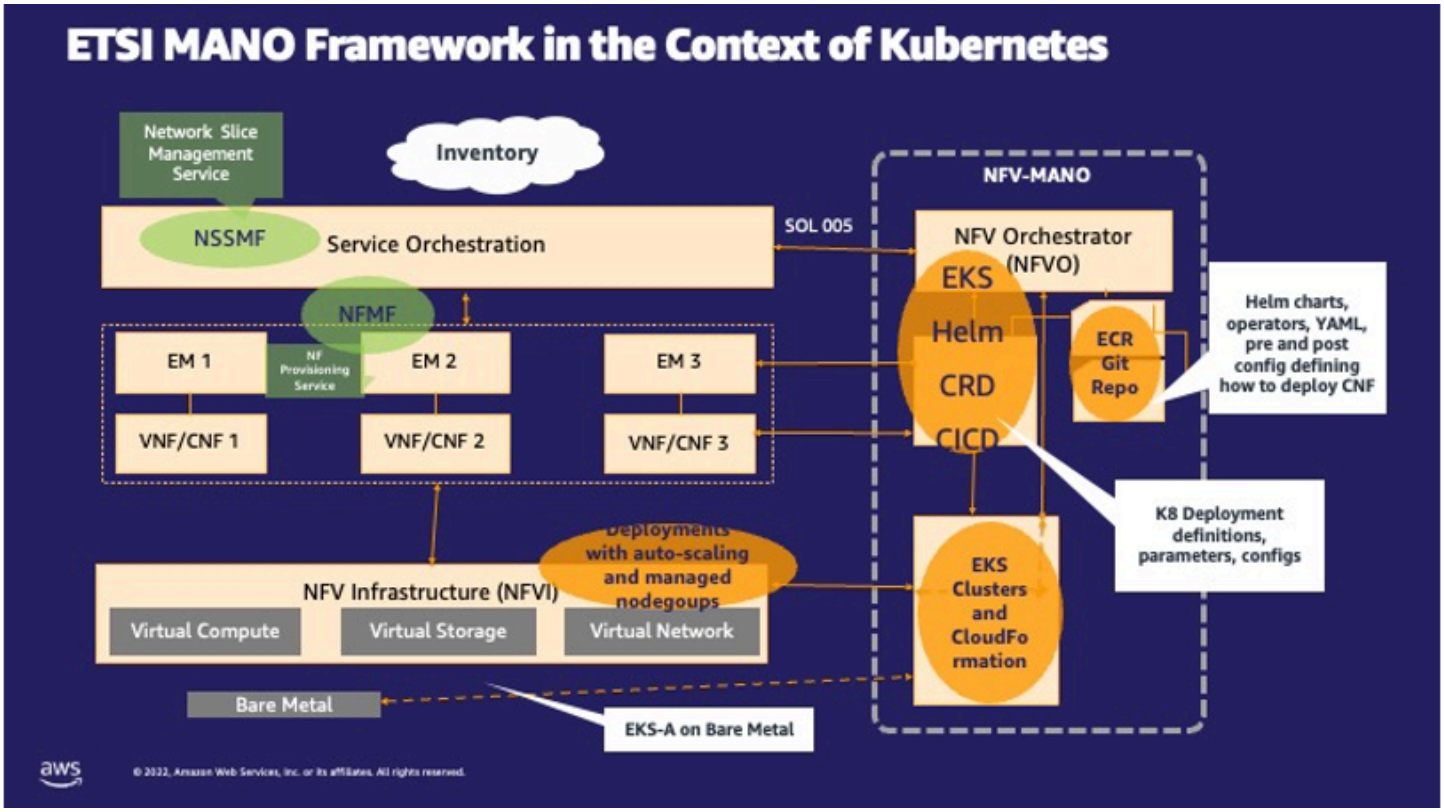
Due to these characteristics of VNFs, the following functions are required of NFVO:

- Manage/coordinate the resources from different VIMs, when there are multiple VIMs (that may be in the same or different physical resources), NFVO calls northbound APIs of VIM instead of engaging with the NFVI resources directly.
- Manage/coordinate the creation of a service that involves multiple VNFs that might be managed by different VNFMs. In NFVO parlance, this is sometimes called service orchestration. Often, this is a source of confusion because in the cloud model, a service might represent an application.
- Manage topology of the network services instances (also called VNF forwarding graphs).

In particular, NFVO is responsible to track scale status, virtualized resources used, and connectivity to VIMs to manage the resources of VNFs. In ETSI MANO, some of the functionalities done by VNFM are controlled by NFVO in imperative way by mechanisms such as request and grant, where VNFM asks explicit permissions from NFVO for some of its operations. This required NFVO to micro-manage some of VNFM operations, which it should have ignored.

To be fair, ETSI IFA029 specification did translate VIM and VNFM architecture and requirements into a container framework by creating new terms Container Infrastructure Service (CIS), Container Infrastructure Service Management (CISM) and Container Infrastructure Service Instance (CISI). However, IFA029 didn't get sufficient exposure, partly due to the difficulty in interpreting the new abstraction for containerized platforms, instead of directly describing functionalities in well-known Kubernetes terms. This required additional effort in reading and translating IFA029 to Kubernetes. Because the Kubernetes framework has become a standard for telecom network equipment providers (NEPs) and ISVs to use for their functions, in this whitepaper, we chose to directly translate requirements to Kubernetes. When we need specific product details, we have used Amazon EKS and associated services.

In light of the previous discussions, in the following diagram, we propose a way to map the same ETSI defined functions to corresponding Kubernetes and associated objects. Notice that the mapping is not 1-1, and Kubernetes and associated services such as ECR and Helm cover more than one element of the ETSI MANO architecture.



ETSI MANO Framework and Kubernetes and associated constructs

This creates some difficulties in applying understandings gained from ETSI MANO NFV to Kubernetes operational environments. Some of these difficulties are discussed next.

Framework mapping challenges

As outlined earlier, the main difficulty in mapping ETSI MANO model to Kubernetes framework is that the first takes the declarative approach to imperative operations, whereas the later takes the declarative approach to an intent-based framework. ETSI MANO specifies direct procedures between NFVO and VNFM and VIM, whereas Kubernetes performs all communication among its components as well as on north and southbound interfaces by artifacts, APIs and manifests with specification of the desired final state of operation. This paper next discusses in detail the impact of this difference in approaches.

The ETSI MANO architecture uses lifecycle operation granting, where ETSI VNFM asks NFVO before initiating changes within the defined parameters. Kubernetes doesn't ask the orchestrator whether to scale in, scale out, where to place, and so on. In the Kubernetes style of operation, NFVO only needs to define the desired end state in declarative ways by defining artifacts, such as deployment and service YAML files, and then Kubernetes schedules ensures operation within the specified

range. When needed, Kubernetes can also use the cluster automatic scaler to request additional resources from an underlying cloud infrastructure to meet the additional needs of a Kubernetes workload within its realm of operations.

Lifecycle operation granting duplicates the work and breaks the hierarchical model. In the cloud-native world, NFVO indication of the desired state of operation should be implemented using intent-driven Kubernetes constructs and changes to Deployment and so on, instead of implementing the procedural model between the NFVO and VNFM.

The Kubernetes scheduler, along with constructs such as AWS Auto Scaling groups in EKS are well equipped to efficiently manage demand on resources, and NFVO should leave it to them to manage in the Kubernetes framework. With Kubernetes, the big, closed control loop has been replaced with multiple small, declarative control loops that gives it the power and flexibility to become a standard in the cloud-native world.

Another complexity is the configuration of VNFs once they are instantiated. Configuration is coupled in ETSI MANO where the VNFM, while instantiating VNF, also takes the responsibility for interacting with the element manager to configure the VNF. Kubernetes doesn't have built-in mechanisms for application configuration, although with capabilities such as lifecycle hooks, init containers, ConfigMap, and Operators, you can build an efficient devops way of configuring CNFs during or after CNF instantiation.

Service function chaining is a higher level of abstraction built on top of individual network and service functions that applies ordering constraints to the packet or flow of packets based on the result of some classification upstream. The networking world relies extensively on service function chaining to fulfill important tasks such as insertion of firewall or deep packet inspection in the path of some flows. The difficulties in defining a good service chaining solution stem from apparently simple but harder to implement requirements, such as being lightweight, allowing for easy debugging, and not adding unnecessary encapsulation overhead while steering traffic to the next service in the chain without packet modification. Some of the services are stateful, which implies that the packet should not only follow the service but also the exact instantiated service function that had served the earlier packet in the flow. These requirements have been hard to implement in the VNF world and also in the Kubernetes world. More research and work are needed to solve service chaining issues in a true intent-driven way.

The last but not least important aspect is service assurance, meaning how the system deals with failures and performance impact. Due to the Kubernetes intent-based framework and its ability to take care of Pod failures, many of the failures and performance issues can be taken care of at the Kubernetes level, and operation support system (OSS) and business support system (BSS) don't

need to have control to recover from those failures. However, you do need a mechanism to track service KPIs from the OSS/BSS perspective. There isn't really a Kubernetes-native standard, but there are some popular standards that have emerged (such as Prometheus and OpenTelemetry) that do collect metrics, and on which service KPI solutions can be built. Service assurance itself is a big area of study but we will not go into further details in this paper.

Next, we'll examine the impact of orchestration on achieving business and technical requirements of a CSP.

Service provider requirements

With the mobile traffic increasing almost 10% quarter by quarter (Ericsson Mobility Report) and technology evolving at a fast pace, service providers are under pressure from multiple dimensions. Some of the challenges are as follows:

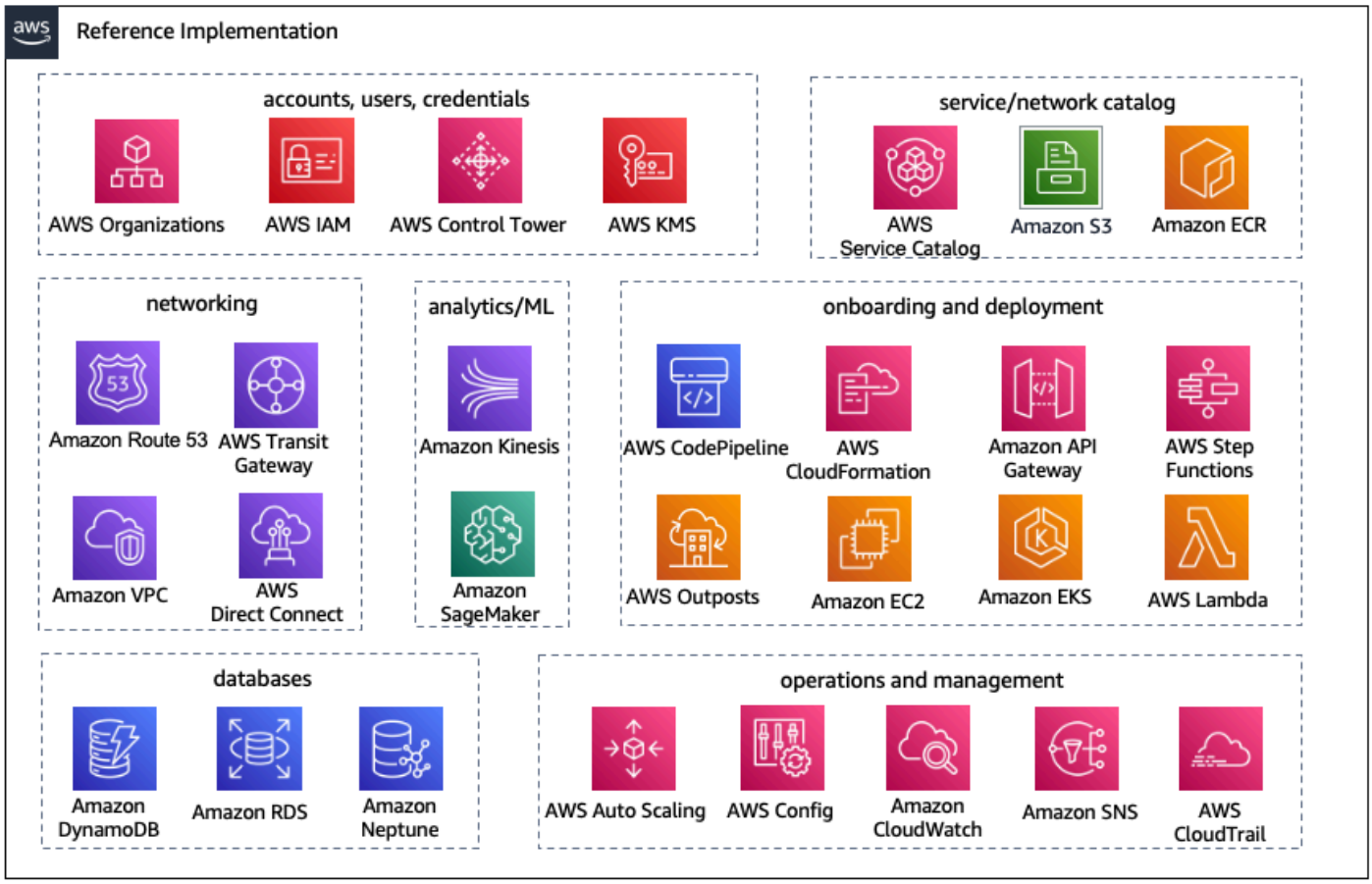
- **Cost savings (TCO) and productivity** — The increase in traffic corresponds to higher bandwidth and processing capacity requirements. With proliferation of unlimited plans, this increase might not necessarily translate to new revenues. Therefore, it is imperative to lower the cost of operations and to enable new enterprise use cases by avoiding complex interactions between functions. Avoiding duplication of functionalities and tying it with custom logics increases the cost of deployment and operations. It's important to think of truly cloud-native, Kubernetes-savvy orchestrators with abstractions that avoid duplication and overlap of functionalities. Intent-driven and not procedure-driven coordination between different functions also increases staff productivity.
- **Agility** — Since the traffic demand fluctuates, CSPs are thinking of ways to elastically scale to provide the required capacity. This requires a versatile network orchestrator that can stand up, take down, scale-out, or scale-in both network functions and infrastructure. Infrastructure cost should be optimized, with baseline infrastructure being negotiated as a long-term contract, while the fluctuating demand is addressed by a pay-as-you go model. Cloud cost models are beneficial in this role if the network orchestrator supports fluctuating and agile operation.
- **Assurance and Resilience** — With more and more safety- and business-critical services relying on connected infrastructure, it is important that the connectivity service meets SLAs. When failures happen, an intelligent orchestration system should provide automated self-healing while keeping customer experience within desired SLA bounds.

To achieve the preceding business goals, it's important that customers design their orchestration and automation architecture with the goals in mind. To achieve these goals, orchestrator implementation itself should take advantage of the latest cloud-native and serverless implementation best practices. These features enable the orchestration solution to be resilient and agile, and allow the solution to take advantage of the latest infrastructure and CNF functionalities. It also makes the orchestrator itself to evolve rapidly as the requirements changes.

In the following section, we outline one such possible implementation in the context of day-to-day operations of the network that achieves the desired agility, assurance, and resilience goals while also being cost-effective

Solution architecture

Due to the different approaches in cloud-native operations versus virtualized application operations, the day -1 to day 2 operations also look different in the two. In this section, we will go over typical day -1 to day 2 operations, and explore how a cloud native orchestration can be built using those requirements. The following diagram represents a grouping of relevant AWS services for implementation of a cloud-native network orchestrator. To maintain ease of reading, we have ignored some of the operational requirements such as reliability, security and recovery from the following diagram, however in a real implementation, it is important to consider those requirements as well.



AWS constructs for a cloud-native CNF and infrastructure orchestrator

Motivation and tasks of each service in the preceding figure will become clear as the operational requirements are examined in detail in the following. Some guidelines to develop automation steps that are scalable while taking advantage of AWS and EKS native constructs for maximum flexibility is also presented.

Day -1: Planning

Common tasks during this phase are:

- Creation of account structure
- Planning IP/subnets
- Working with ISVs and CSP to populate Customer Information Questionnaire (CIQ) and artifacts
- Ordering AWS Outposts if needed
- Defining naming conventions, metadata and tags
- Creating AWS Identity and Access Management (AWS IAM) accounts and roles in the account structure
- Creating service and CNF catalog
- Giving appropriate permissions and set policies
- Creating infrastructure — Deploy AWS constructs such as Amazon Virtual Private Cloud (Amazon VPC), AWS Transit Gateway, AWS Direct Connect, and subnets.

Most of these activities are covered by proper landing zone design, discussions with ISVs and network teams to create well-structured accounts and permissions, naming conventions and network design. AWS services such as AWS Organizations and AWS Control Tower can be quite useful to develop proper account structure and management, such as new account creation. With (AWS IAM), you can specify who or what can access services and resources in AWS, centrally manage fine-grained permissions, and analyze access and refine permissions across AWS. AWS Key Management Service (AWS KMS) helps create, manage, and control cryptographic keys across applications and more than 100 AWS services, and helps with secure access and management.

Amazon Virtual Private Cloud (Amazon VPC) gives you full control over virtual networking environment, including resource placement, connectivity, and security. One of more VPCs might be required depending on VPC design and scale. AWS Direct Connect links the CSP internal network to a Direct Connect location over a standard ethernet fiber-optic cable. AWS Transit Gateway connects Amazon VPCs and on-premises networks through a central routing hub. This simplifies the network and puts an end to complex peering relationships as each new connection is only made once. Amazon Route 53 is a highly-available and scalable domain name system (DNS) web service. Route 53 connects user requests to internet applications running on AWS or on-premises.

Amazon Elastic Container Registry (Amazon ECR) is an AWS-managed, Open Container Initiative (OCI)-compliant container image registry service that is secure, scalable, and reliable. Amazon ECR

supports private repositories with resource-based permissions using AWS IAM. This ensures that only specified users or Amazon EC2 instances can access container repositories and images, thereby allowing separation across vendors. Customers can use the familiar Docker CLI, or their preferred client, to push, pull, and manage Docker images, OCI images, and OCI-compatible artifacts.

Significant engineering effort and consideration should be given at this stage, as this lays the foundation of future automation and operations. Although proper planning needs human decisions, implementation of these design choices can often be automated.

Day 0: Topology development

Some of the tasks in this phase of deployment are as follows:

- Activate hardware, such as AWS Outposts instance, if it is deployed.
- Develop a service/CNF catalog — This catalog contains services that upper layers can call.
- Deploy platforms such as EKS clusters, Container Network Interfaces (CNIs), Container Storage Interfaces (CSIs), vRouters, and observability infrastructure such as probes and clients.
- Boot up infrastructure such as node groups.

Some of the well-developed robotic automation tools or customized process automation tools can be developed to activate AWS Outposts. Services such as Service Catalog can be useful in creating the catalog and customizing it for the particular CSP and ISVs. Care should be taken to properly abstract configuration parameters to avoid bloating the catalog size.

AWS CloudFormation or the AWS Cloud Development Kit (AWS CDK) constructs are flexible and functionally rich tools to deploy infrastructure and many of the platform components such as EKS clusters, CNIs, CSIs, and so on. The invocation of AWS CloudFormation/AWS CDK templates can be further customized using AWS CodePipeline with tools such as AWS CodeCommit and AWS CodeDeploy. Some of the functions can also be automated using a purpose-built automation workflow using AWS Lambda and AWS Step Functions.

Increased agility and resilience can be achieved using Amazon EKS, an AWS managed Kubernetes service that makes it easy to run Kubernetes on AWS. The Kubernetes control plane managed by EKS runs inside an EKS-managed VPC, running components such as the Kubernetes API server nodes and etcd cluster. Kubernetes API server nodes run the API server, scheduler, kube-controller-manager, and so on. in an EC2 Auto Scaling group, allowing it to scale based on demand. API server nodes run in a minimum configuration of two in distinct Availability Zones (AZs), while the etcd

server nodes run in an auto-scaling group that spans three Availability Zones. This architecture ensures that an event in a single Availability Zone doesn't affect the EKS cluster's availability. The control plane backup (such as etcd backup) is periodically performed by AWS. Having a unified, managed Kubernetes control plane helps with operational agility and automation.

For on-premises infrastructure not managed by AWS, Amazon EKS Anywhere is a new deployment option that allows customers to create and operate Kubernetes clusters. Amazon EKS Anywhere helps simplify the creation and operation of on-premises Kubernetes clusters with default component configurations while providing tools for automating cluster management. While not as feature-rich as EKS, it gives customers option to view all of their Kubernetes clusters in one dashboard.

Day 1: Instantiation

This phase of deployment deals with the following tasks:

- Instantiate CNFs
- Update route tables

Most of these functions can be automated if properly designed. CloudFormation and AWS CDKs are good constructs for this part. An automation pipeline can be build using AWS-provided continuous integration/continuous development (CI/CD) tools such as AWS CodeCommit and AWS CodeDeploy, or custom workflows can be created using AWS Lambda and AWS Step Functions that invoke appropriate AWS CloudFormation and CDK templates. Again, care should be taken to develop pipelines with appropriate parameterization so as to not bloat the number of pipelines or custom workflows. Some beneficial approaches in this part of the deployment are constructs such as Amazon EKS blueprints.

You must also create proper databases to handle the vast and different types of data that is generated by network functions, and to ensure proper mapping between services, functions and their instantiation. Graph database such as Amazon Neptune can be useful in this regard. Amazon Neptune is a fast, reliable, fully managed graph database service that makes it easy to build and run applications. Amazon DynamoDB is a fully managed, serverless, key-value NoSQL database designed to run high-performance applications at any scale. DynamoDB offers built-in security, continuous backups, automated multi-Region replication, in-memory caching, and data import and export tools. Amazon Relational Database Service (Amazon RDS) is a managed relational

database service for MySQL, PostgreSQL, MariaDB, Oracle BYOL, or SQL Server. Some AWS partner solutions, such as Portworx PX-Enterprise, can be useful in architecting CNFs for high availability.

Day 2: Operation and management

This is arguably the hardest part of the automation lifecycle, and deals with day-to-day operation of the network. This phase deals with the following tasks:

- Update and scale CNFs
- Update and scale network services
- Update EKS version
- Update configuration
- Allow creation of new services
- Monitor and manage
- End service/CNFs when not needed

New infrastructure, network and functions can also be deployed in this part to address increased network demand. Hence, it is important not to view this phase in isolation of earlier phases but think of it this phase as invoking all the earlier phases as and when needed.

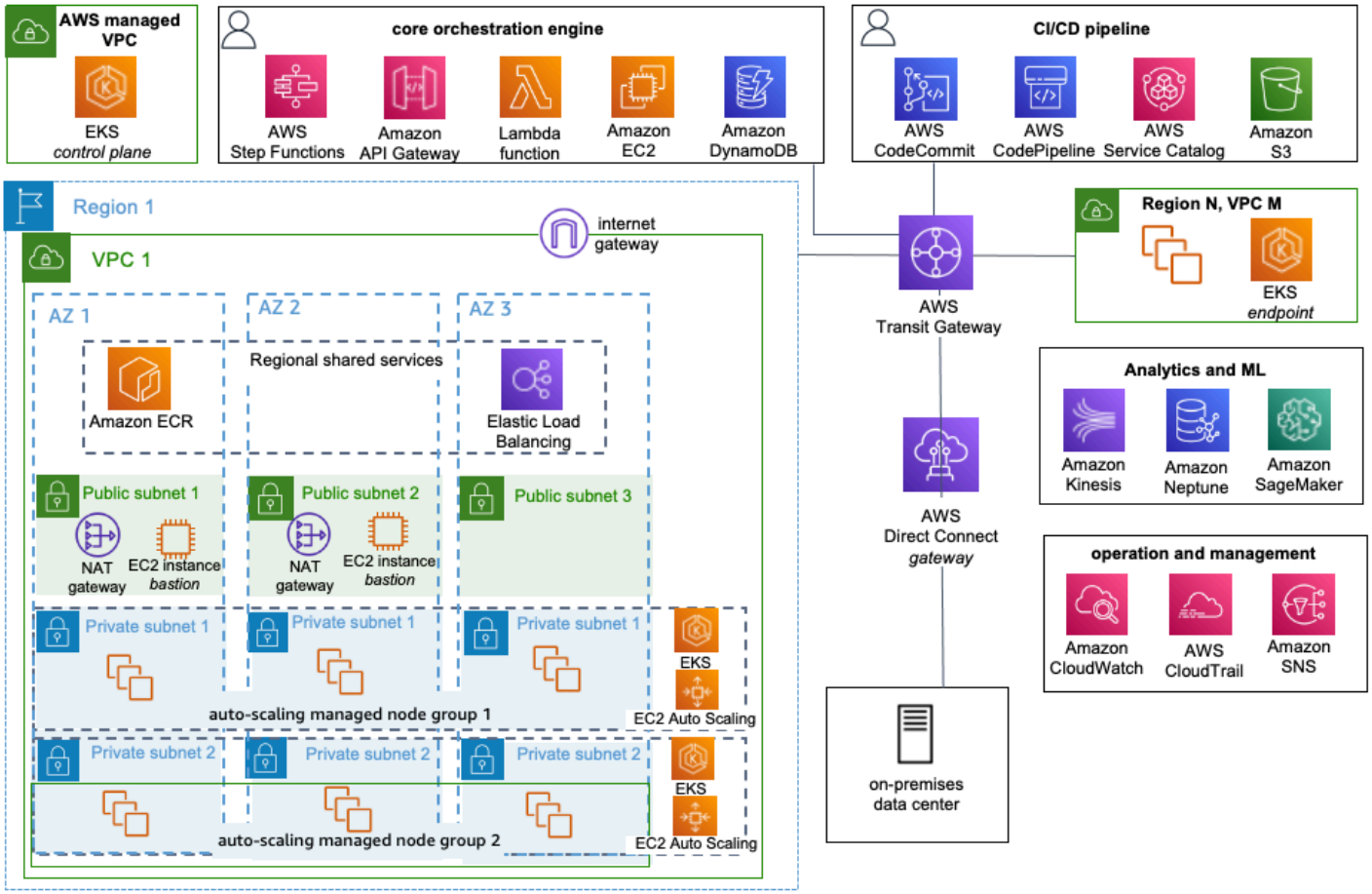
This phase is also the one that is most difficult to handle with traditional CI/CD. However, the new GitOps-based approaches can be particularly beneficial in this context. GitOps enables configuration as code and, if properly implemented, can take care of drift management from the desired configuration. This model is often utilized as an efficient strategy for provisioning cloud provider-specific managed resources, such as Amazon Simple Storage Service (Amazon S3) bucket and Amazon RDS instance, on which application workloads depends. Furthermore, AWS constructs such as AWS Auto Scaling can provide a cost-effective way to manage utilization and allow for traffic adaptation. Combining this approach with an application configuration provides a useful method to manage the operational configuration.

Monitoring, observability, and logging alarm for the Day 2 operations can be achieved using services such as Amazon CloudWatch, AWS CloudTrail, and AWS-provided managed services such as Amazon Managed Service for Prometheus and AWS Distro for OpenTelemetry. AWS CloudWatch collects monitoring and operational data in the form of logs, metrics, and events so that the operation teams can get a unified view of operational health and gain complete visibility of AWS resources, applications, and services running on AWS and on-premises. You can use CloudWatch

to detect anomalous behavior in CSP environments, set alarms, visualize logs and metrics side by side, take automated actions, troubleshoot issues, and discover insights to keep network running smoothly.

Because network functions continuously emit performance data and Key Performance Indicators (KPIs), you'll need a way to process this streaming data. Amazon Kinesis makes it easy to collect, process, and analyze near real-time, streaming data to get timely insights and react quickly to new information. Amazon SageMaker helps prepare, build, train, and deploy high-quality machine learning (ML) models quickly by bringing together a broad set of capabilities purpose-built for ML. This makes it easy to get insights on the deployment and operations.

With the previous described AWS constructs, one possible implementation on AWS is be as follows:



Example implementation architecture of a cloud-native CNF and infrastructure orchestrator

This diagram represents VPC constructs, EKS clusters, load-balancers and repositories, network connections, and so on in the context of Region, Availability Zones, and on-premises data centers. For ease of representation, we haven't depicted some of the functionalities such as account and

user administration, the creation of a landing zone, security, and DNS that were part of the earlier architecture, because many of those features will run in their own VPCs within control of cross-account permissions.

Conclusion

In this whitepaper, we have discussed the essential role of agile and resilient cloud-native network orchestration to achieve the business and operational goals of service providers. We examined and mapped the NFV orchestration requirements and specifications as put by ETSI in the context of Kubernetes and cloud-native implementations. We classified the requirements into three categories: those that can be mapped, those that are not relevant, and those that need further research and work. We also discussed orchestration needs in various phases of implementation, and provided a way to implement agile and resilient automation and orchestration solutions using AWS container and serverless constructs. These constructs are flexible and can be easily adopted to meet CSP and ISV automation requirements.

For further information on AWS telco offerings, and how some of these constructs have been used with the service providers, visit https://aws.amazon.com/telecom/ .

Contributors

Contributors to this document include:

- Dr. Manjari Asawa, Senior Solution Architect, in the AWS Worldwide Telecom Business Unit

Document revisions

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Initial publication	Whitepaper published.	October 25, 2022

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.