CLOUDIFY

Open vCPE Design Patterns with TOSCA and Cloudify

hello@cloudify.co | www.cloudify.co

Executive Summary

vCPE is probably the most widely sought after use case for enterprise and telecom networking. It is a vital system which connects remote offices to each other, and to the internet in an automated and systematic manner, which reduces deployment times significantly, and by doing so, also reduces cost.

This paper will discuss a new concept in Open vCPE - an Integration-based Orchestration Approach. It will introduce new design patterns to provide an easy mechanism and simple tooling for dynamic and agile open service creation and management.

Open, in the context of this paper, means that a user can choose any VNF onboarded to a catalog and use it. For example, a user can choose any vFW from the those in the catalog. Figure 1 shows the vFW as part of a service chain starting from the CPE through a PE router to a hosted vFW in the service provider's cloud. Additional VNFs can be dynamically chained and the service updated.

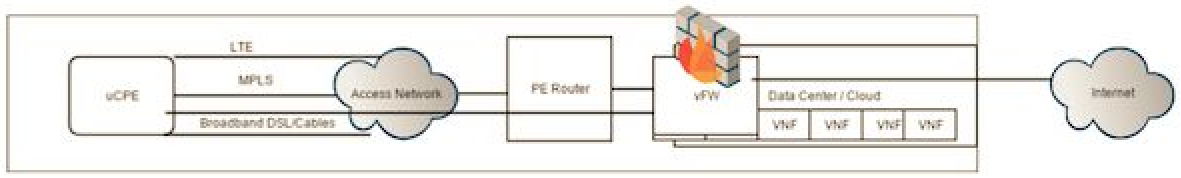
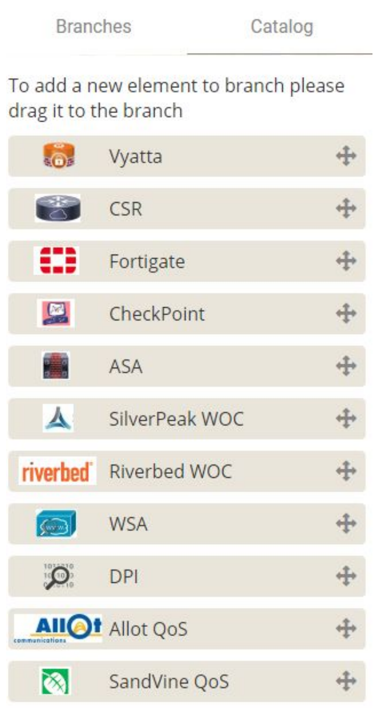


Figure 1

The whole process starts with the creation of a VNF catalog. Figure 2 shows an example of such a catalog populated with multiple VNF types. To simplify the process of onboarding a VNF, we created a wizard that generates TOSCA blueprints which can be onboarded to multiple ONAP and MANO stacks.



Branches

Catalog

To add a new element to branch please drag it to the branch

Vyatta

CSR

Fortigate

CheckPoint

ASA

SilverPeak WOC

Riverbed WOC

WSA

DPI

Allot QoS

SandVine QoS

Figure 2

This wizard generates lifecycle interfaces to support different phases of the onboarding process, e.g. Install, Configure, Start, Post-Start, Stop, Delete. Each lifecycle interface can be associated with an operation conducted at this phase.

Each VNF is encapsulated in its own TOSCA blueprint and could be thought of as a Microservice. Chaining multiple VNFs into a service could be achieved in several ways - one of them is the *proxy pattern* that is able to chain blueprints together by sending each one of them input parameters and getting back output parameters that can be fed into other blueprints, as depicted in Figure 3.

Moreover, a TOSCA blueprint can be generated for each PNF, so you virtually treat PNFs as Microservices as well, which makes the manipulation at the definition level the same for VNFs and PNFs. Of course, this implementation is different and makes use of a different plugin.

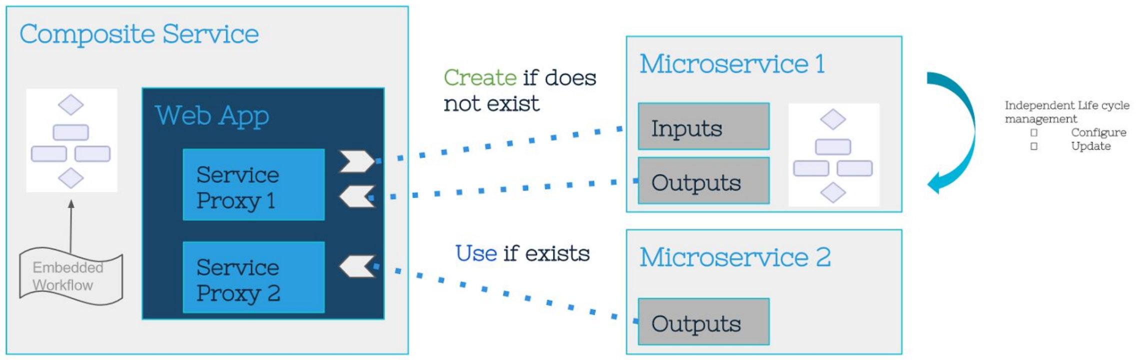


Figure 3

The next step is to create the service. This paper describes two types of services:

- **Basic service** - This is a layered design pattern approach where the user first defines the basic infrastructure components needed to run any service by all tenants. The next layer would be provisioning service components for each tenant, e.g. each tenant has its own VLAN IDs across the end-to-end chain. The next layer is per CPE or branch of the tenant, e.g. the desired configuration to provision per remote branch CPE.

- **Advanced service chain** - Here we abstract the underlying implementation and create new TOSCA types to describe Connection Points that hook into a VNF on one hand and to the underlying infrastructure (e.g. OVS switch ID, VLAN ID) on the other hand. An example of such a Forwarding Graph (FG) is shown in Figures 4 and 5. Figure 4 shows the conceptual chaining of a UTM vFW and a Docker container while Figure 5 shows the TOSCA node template definition of the Forwarding Graph.

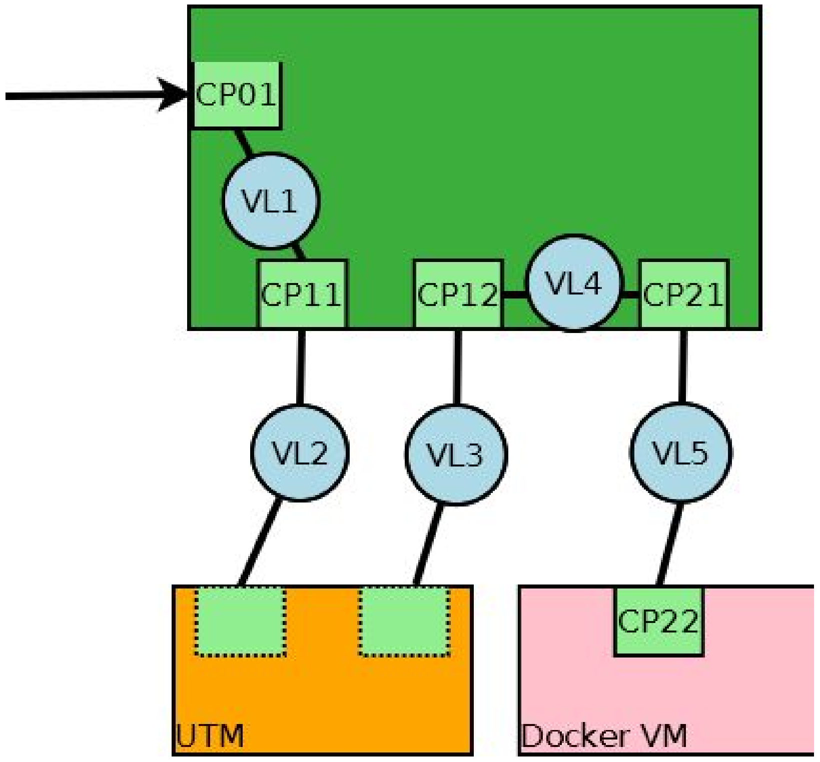
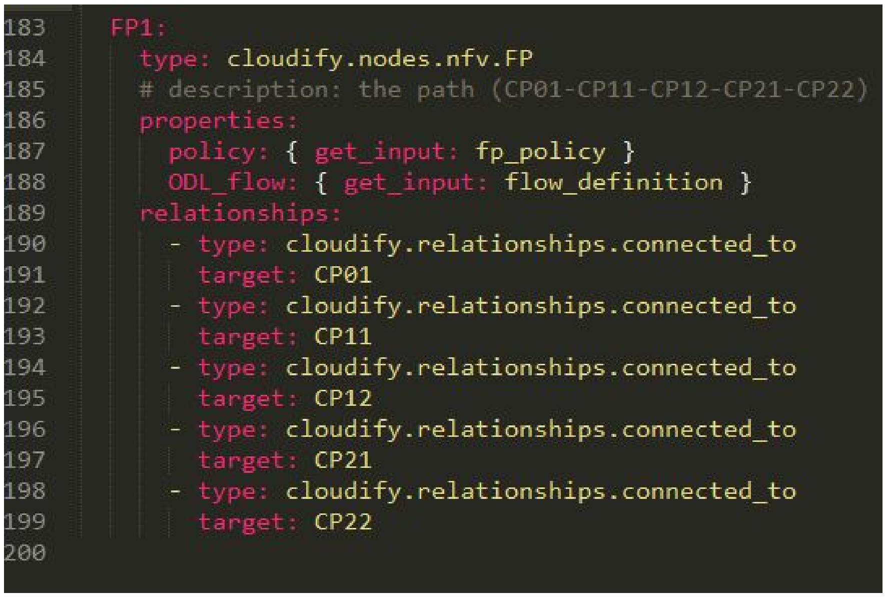


Figure 4



```
  FP1:
    type: cloudify.nodes.nfv.FP
    # description: the path (CP01-CP11-CP12-CP21-CP22)
    properties:
      policy: { get_input: fp_policy }
      ODL_flow: { get_input: flow_definition }
    relationships:
      - type: cloudify.relationships.connected_to
        target: CP01
      - type: cloudify.relationships.connected_to
        target: CP11
      - type: cloudify.relationships.connected_to
        target: CP12
      - type: cloudify.relationships.connected_to
        target: CP21
      - type: cloudify.relationships.connected_to
        target: CP22
```

Figure 5

The created service topology graph is maintained for future operations. For example, Day 2 operation scenarios such as service chain updates. This is achieved utilizing Cloudify's advanced capability for comparing two different topologies represented in TOSCA YAML files and applying the changes. Such changes could be adding another VNF to a service chain or scaling out different components along the service chain path. Figure 6 shows dynamic addition of a third Microservice to the service chain utilizing the Proxy design pattern.

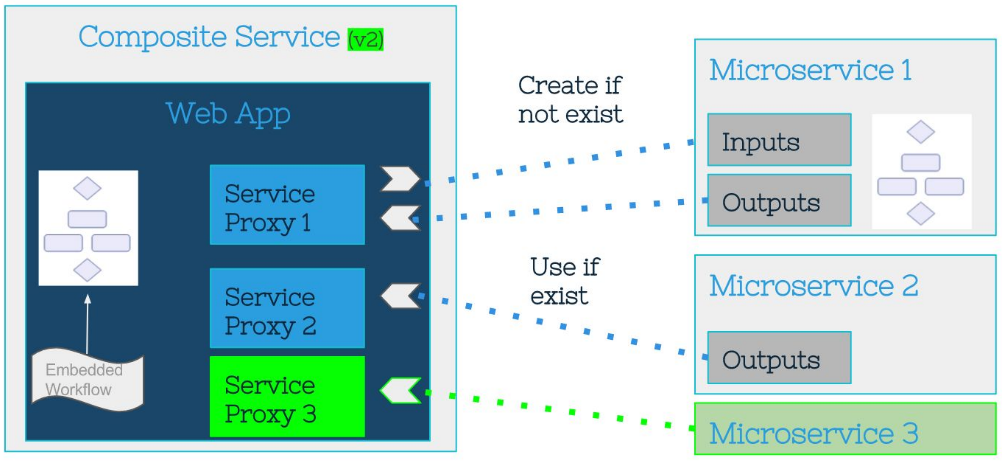


Figure 6

Introduction

This paper discusses common and useful design patterns for the creation of NFV-based services, otherwise known as Network Services in NFV terminology. It expands on the concepts explored in "Building Large Scale Services with Micro Services, TOSCA and Cloudify" paper by Nati Shalom, and includes an example of an open vCPE providing a secured internet connection as defined in the blog post "An Open Source, Dynamic, and Integration-Based vCPE/SD-WAN Orchestration Service Built with Cloudify" by Shay Naeh.

Open means that you are not tied to any one vendor solution - no more lock-in. Should you prefer to change any component along the path from CPE to virtualized VNFs, you can do so. For example, if you want to switch to a different Firewall vendor, just choose it from the catalog and dynamically update your service chain - in real-time.

This paper brings together a few concepts, which have been applied at actual customer engagements. We begin with the VNF onboarding Microservice design pattern, then move on to describe a basic service chaining pattern, followed by an advanced pattern supporting forwarding graphs and connection points as defined in the TOSCA NFV profile.

Design patterns are flexible and allow encapsulation of VNFs as well as PNFs. This enables Service Providers (SP) to get started with any hardware they have without the need to change their infrastructure end-to-end.

The orchestration-first approach is an evolutionary, open, and integrated model that allows the SP to implement and grow in phases. Other approaches require fundamental replacement of many infrastructure components, if not all of them, and is very expensive.

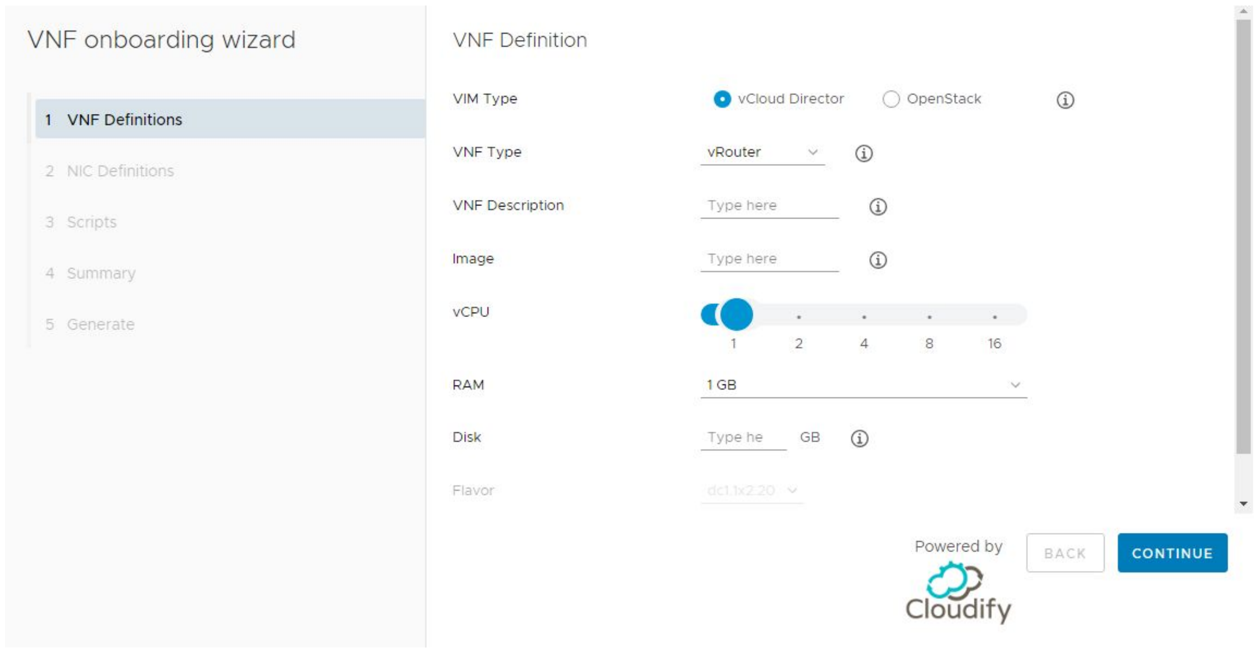
The paper will also discuss real use cases for dynamic service chaining across multiple tiers. There are different service-chaining implementations, some use basic rudimentary VLAN IDs, while others involve VLAN IDs, VRFs or SDN controllers.

Using abstraction patterns, such as new TOSCA types that are associated and matched with lifecycle operations (or plugins), provides the necessary abstraction and implementation details. They can support various configurations mixed together in one chain - PNFs, VNFs, Containers, etc. These VNFs become cloud-native VNFs running as Microservices with built-in support for healing, scaling, dynamic service-chain updates, and more, which makes service creation dynamic and agile.

VNF encapsulation pattern aka VNF Onboarding

As noted above, open means having the ability to utilize any VNF that you feel best fits your stack without lock-in to a specific vendor. Moreover, it means being able to onboard the VNF to any ONAP / MANO stack. VNF vendors need to invest time, cost, and have experience to make their VNF cloud-native and to onboard it to these different stacks.

To expedite the process, Cloudify, together with VMWare and Intel, created a wizard that generates a TOSCA blueprint enabling VNF onboarding on multiple stacks. The generated TOSCA blueprint includes lifecycle interfaces associated with operations conducted at different phases in the onboarding process such as Configure, Start, Stop, etc.



VIM Type: (● vCloud Director) ○ OpenStack
VNF Type: vRouter
VNF Description: Type here
Image: Type here
vCPU: 1 2 4 8 16 (1 selected)
RAM: 1 GB
Disk: Type he GB
Flavor: dc1.1x2.20

Figure 7

Taking a quick look at Figure 7, we can see that the wizard has three main steps.

In step 1, we collect information on the VNF such as VNF type, VNF image, number of vCPUs, RAM and disk space, cloud flavor type, etc.

In step 2, we assign the VNF network interfaces to the correct networks, e.g. interface one to a management network, interface two to inbound, and interface three to an outbound network segment.

In step 3, we incorporate the lifecycle scripts to be executed at the right lifecycle phases of the VNF. The vendor supplies a URL to a script to be executed at each lifecycle event.

The final step will generate a blueprint ready to be executed on any compatible MANO stack.

Now, let’s talk about Microservices. You can look at each VNF as a Microservice encapsulated in a TOSCA node, which supports interfaces and lifecycle management (LCM) events. You can utilize TOSCA objects and create specific TOSCA types for a VNF. These new TOSCA types can inherit from a base root TOSCA type (class).

A VNF that is expressed in this way can participate in a network of relationships between Microservices, ordered or unordered. In cases like this, the strength and ability of TOSCA is used to describe topologies of Microservices, aka topologies of VNFs. More on that in the service-chaining pattern section.

Below is an example for a simple vFW VNF instance.

```
Firewall_Micro-Service:
   type: openstack.Instance
   properties:
     image_id: { get_input: image_id }
     flavor_id: { get_input: flavor_id }
     os_users:
       root:
         password:
   interfaces:
     Standard:
       create:
       configure:
```

PNF encapsulation pattern

A great advantage of this is the ability also treat a PNF as a Microservice as well, although with different operations, and many times you don’t create it like a VM because it exists, but you can definitely configure it with lifecycle operations. At the TOSCA abstraction level, you can treat PNFs the same way you treat VNFs which is simple to handle in terms of the TOSCA data model as well as the operations executed on that PNF.

So a TOSCA node representing a PNF looks similar to a VNF TOSCA type just without the *create* lifecycle.

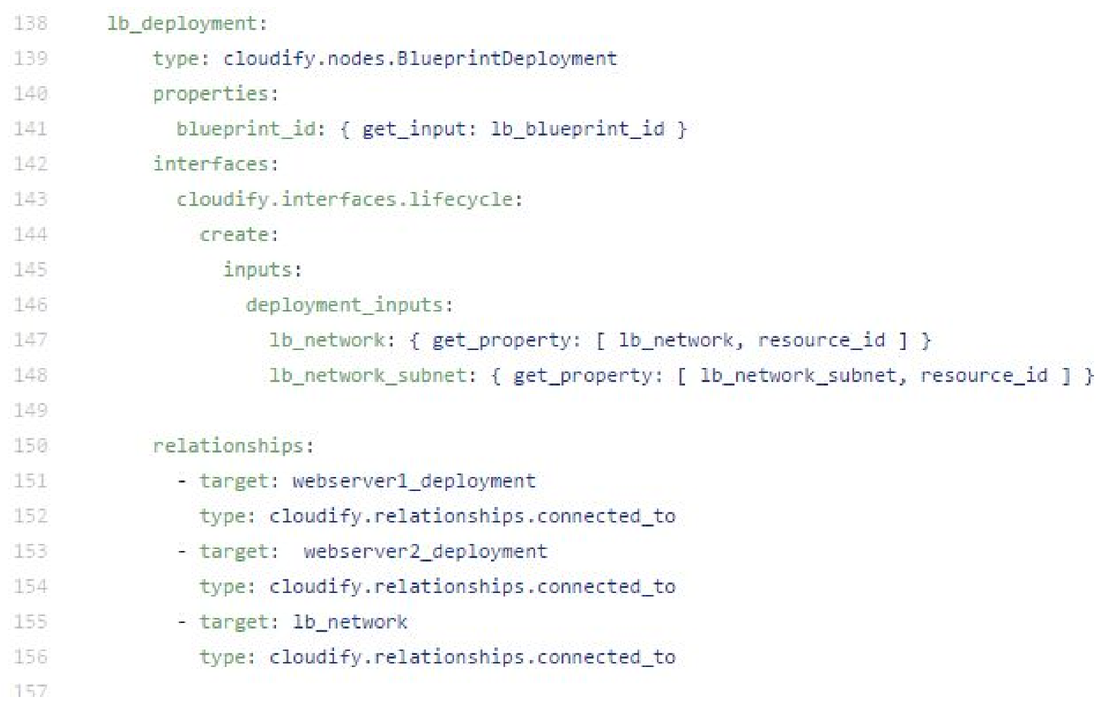
Container encapsulation pattern

Containers can be naturally of expressed as Microservices. In many cases, you will have a hybrid solution consisting of VNFs, PNFs, and containers. Utilizing a design pattern to encapsulate a Microservice as part of a Kubernetes cluster running in a pod, is a common example. Later on, we will see an example how to tie all the pieces together, creating a service chain coming from a remote CPE to a VM running a vFW and then chained into a container application running in a K8S pod.

The Proxy design pattern

As shown in Figures 3 and 4, the proxy pattern can concatenate, or chain, several blueprints together. Each blueprint can represent an encapsulated VNF, PNF, or Microservice in a TOSCA blueprint. The proxy pattern has a master coordinator blueprint which can deploy the other blueprints. Each blueprint can get input parameters and return output parameters. In this manner you can achieve simplicity by taking complex architectures and flatten them to a simple architecture.

Figure 8 shows an example of a TOSCA type at the master coordinator used to deploy another blueprint. You can see this in the *create* lifecycle where parameters are passed, for the load-balancer network and subnet, to the new blueprint.



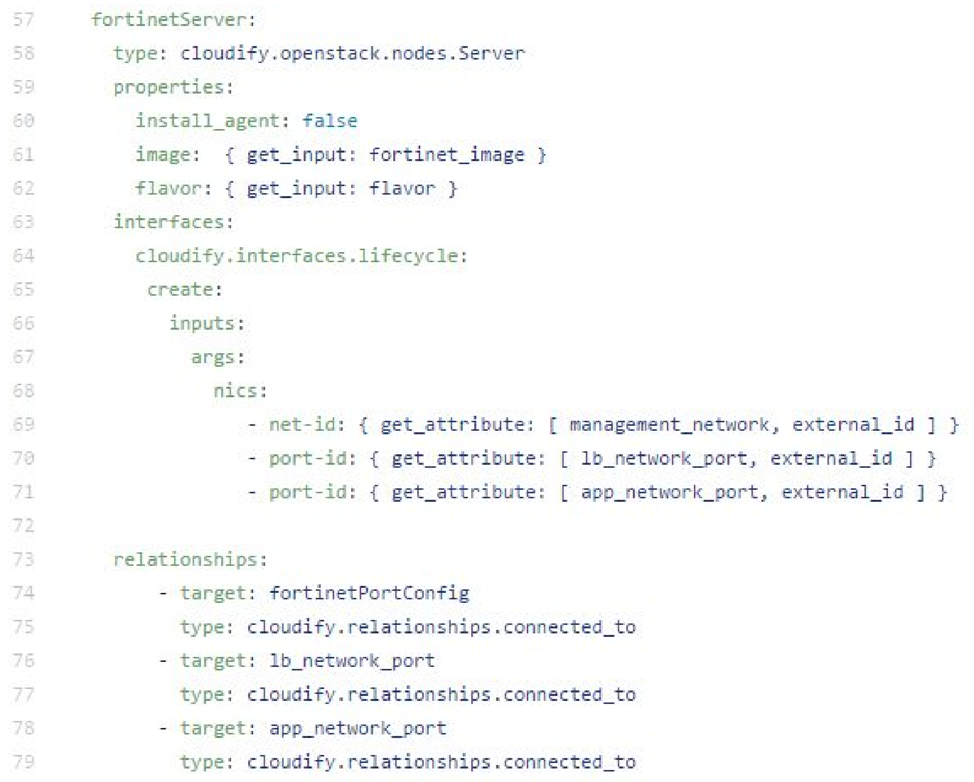
```
lb_deployment:
    type: cloudify.nodes.BlueprintDeployment
    properties:
      blueprint_id: { get_input: lb_blueprint_id }
    interfaces:
      cloudify.interfaces.lifecycle:
        create:
          inputs:
            deployment_inputs:
              lb_network: { get_property: [ lb_network, resource_id ] }
              lb_network_subnet: { get_property: [ lb_network_subnet, resource_id ] }

    relationships:
      - target: webserver1_deployment
        type: cloudify.relationships.connected_to
      - target:  webserver2_deployment
        type: cloudify.relationships.connected_to
      - target: lb_network
        type: cloudify.relationships.connected_to

```

Figure 8

Figure 9 shows a vFW VNF encapsulated in a TOSCA blueprint being deployed by the master blueprint.

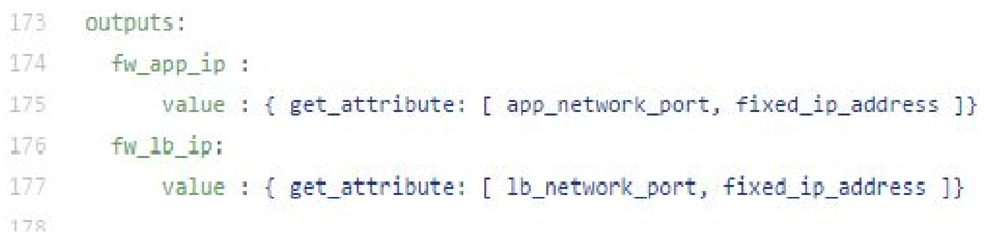


```
fortinetServer:
  type: cloudify.openstack.nodes.Server
  properties:
    install_agent: false
    image:  { get_input: fortinet_image }
    flavor: { get_input: flavor }
  interfaces:
    cloudify.interfaces.lifecycle:
     create:
       inputs:
         args:
           nics:
              - net-id: { get_attribute: [ management_network, external_id ] }
              - port-id: { get_attribute: [ lb_network_port, external_id ] }
              - port-id: { get_attribute: [ app_network_port, external_id ] }

  relationships:
      - target: fortinetPortConfig
        type: cloudify.relationships.connected_to
      - target: lb_network_port
        type: cloudify.relationships.connected_to
      - target: app_network_port
        type: cloudify.relationships.connected_to
```

Figure 9

Figure 10 shows how a deployed blueprint returns parameters in the *outputs* section of the TOSCA deployment blueprint.



```
outputs:
  fw_app_ip :
      value : { get_attribute: [ app_network_port, fixed_ip_address ]}
  fw_lb_ip:
      value : { get_attribute: [ lb_network_port, fixed_ip_address ]}
```

Figure 10

Basic service composition utilizing a layered design pattern

Now let's see how we create a service composition using TOSCA and a layered design pattern.

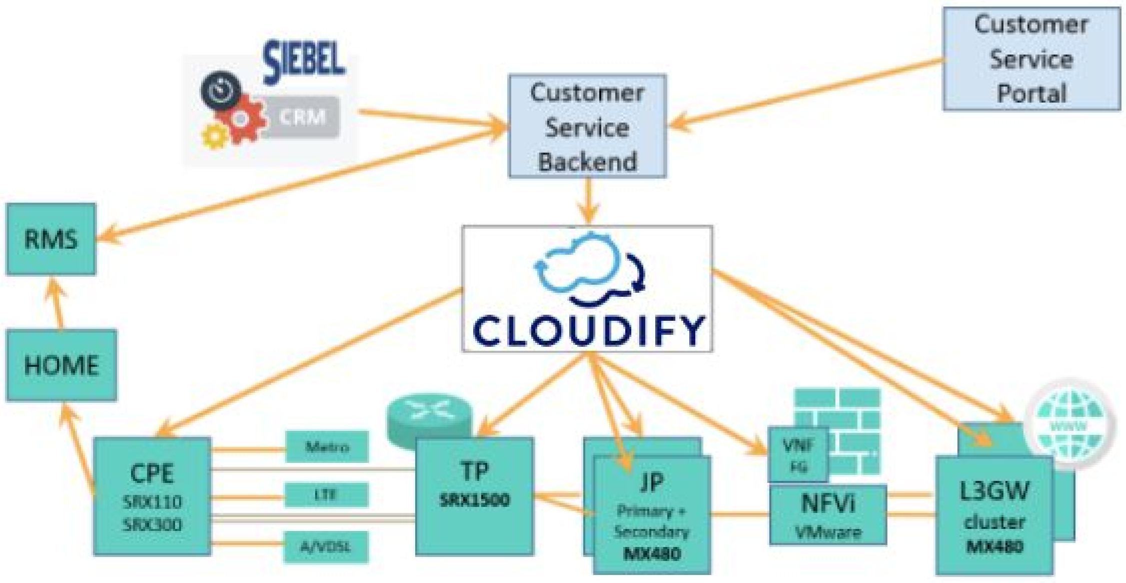


Figure 11

1. TOSCA describes the topology
2. Infrastructure is provisioned first - VNF and PNF
    a. Some of the components in Figure 11 are physical and some are virtual
    b. A TOSCA blueprint provisions the infrastructure, e.g. onboard the vFW according the VNF onboarding patterns described above
    c. Provision connections to L3GW that serves all tenants
    d. PNFs are also encapsulated in a TOSCA blueprint with TOSCA lifecycle events and are configured in the configure lifecycle.
3. Per Tenant service provisioning and chaining
    a. Each tenant's traffic should be carried on a separate VLAN ID.
    b. Provisioning VLAN IDs between PE router (TP), JP, and NFVi per tenant
4. Per CPE (remote branch) provisioning and chaining
    a. Provision IP overlays per VPE (remote branch)
    b. Each branch has separate links for voice and data traffic
5. Dynamic topology updates as new branches are added
    a. Keep a topology graph in memory
    b. As each CPE is added dynamically, update the graph with all attributes and relationships

In addition to the above, Cloudify supports enhanced Netconf plugins that can utilize templates and populate run time attributes and then generate a Netconf XML that in turn is pushed into network devices.

The pattern that we used here is a layered pattern:

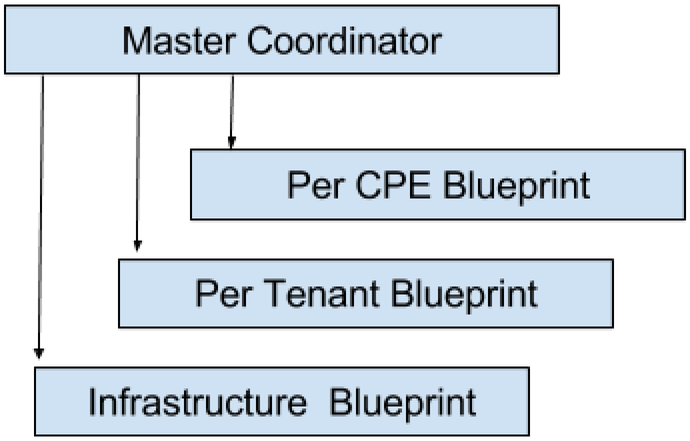


Figure 12

As seen in Figure 12:

1. There is a master coordinator and the solution is provisioned in stages
    a. First layer is provisioned - the infrastructure
    b. Second layer, per tenant, is provisioned per customer tenant
    c. Third layer is provisioned per CPE (remote branch)
2. Each layer is expressed in a TOSCA blueprint
3. TOSCA blueprints describe the topology and relationships between VNFs
    a. A VNF in a TOSCA blueprint can access another VNF instance (can be from a different type) and access its run time attributes e.g. IP addresses, VLAN IDs, etc.
4. Each TOSCA blueprint is layered on the previous one, meaning it adds functionality on top of the previous blueprints towards creation of the end-to-end service chain. In this case it is a secured internet access for each tenant's CPE.
5. Service chains are used here utilizing VLAN IDs and VRFs as depicted in Figure 13, but an additional TOSCA abstraction could be added with new TOSCA types to describe CPs, VLs, and Forwarding Graphs as will explained in the next chapter

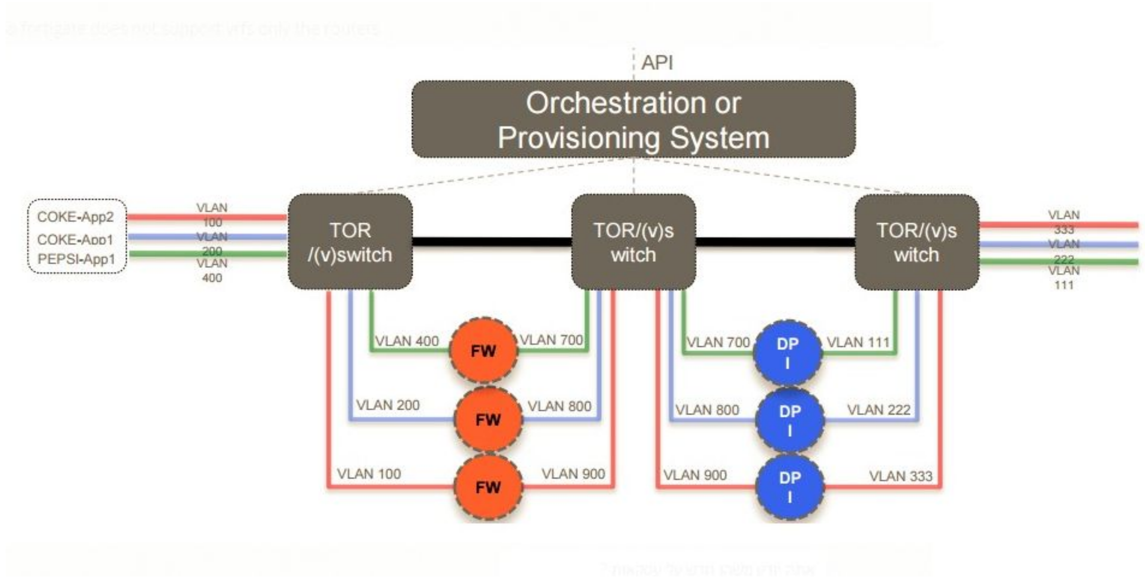
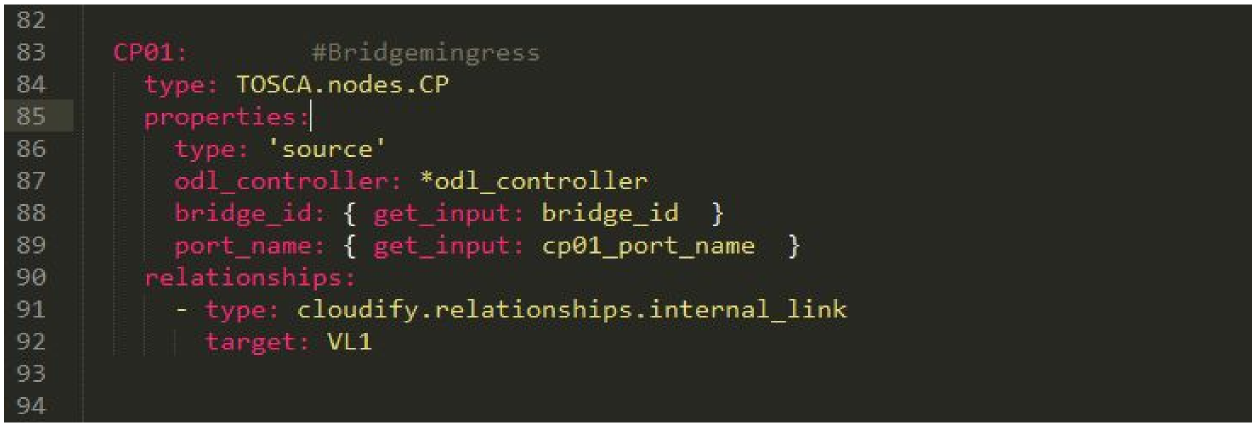


Figure 13

Forwarding Graph, Connection Points, and Virtual Links abstraction pattern, new TOSCA types and plugins

This design pattern abstracts the service chaining mechanism deployed. New TOSCA types for Connection Points, Virtual Links, and Forwarding Graphs are introduced. It implements the TOSCA NFV profile. Each TOSCA type is associated with a plugin that implements it.


```yaml

CP01:       #Bridgemingress
  type: TOSCA.nodes.CP
  properties:
    type: 'source'
    odl_controller: *odl_controller
    bridge_id: { get_input: bridge_id  }
    port_name: { get_input: cp01_port_name  }
  relationships:
    - type: cloudify.relationships.internal_link
      target: VL1


```

Figure 14

In Figure 14, we see a new TOSCA type, CP, which represents a Connection Point and is associated with another introduced TOSCA type - Virtual Link (VL). Each VL has two CPs at its ends. VL has attributes which can characterize the QoS or security attributes of the link.

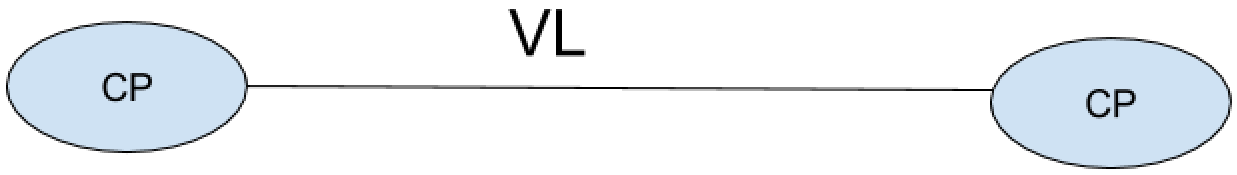
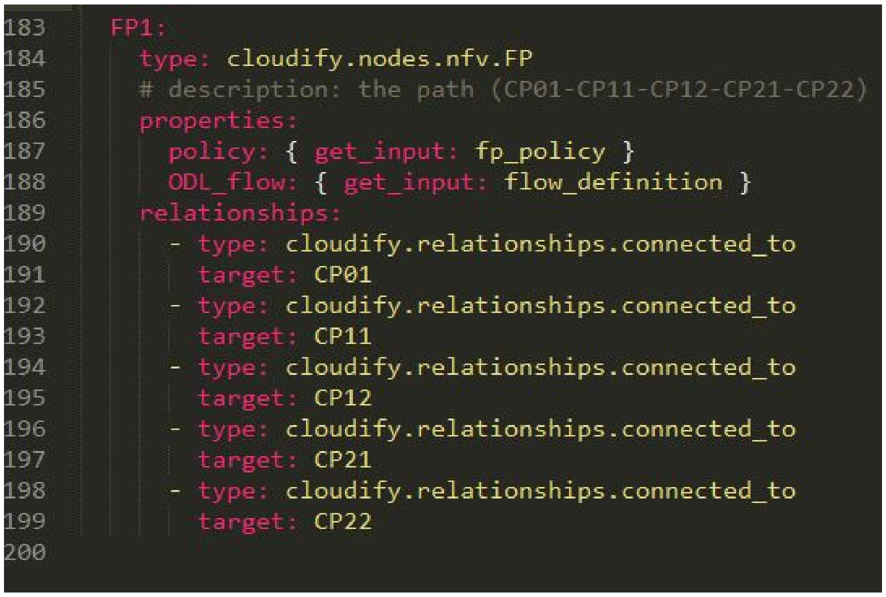


Figure 15

In this example, the CP node type is associated with an Open DayLight plugin and it creates a new entry in the FLOW TABLE utilizing a REST API call to a SDN controller.

It could establish a call to configure VLAN ID or a VRF, of course, with a different plugin. The purpose is to use the same TOSCA type to abstract the mechanism of the service chaining actual implementation and it is expressed by the plugin used.

TOSCA is like an object oriented DSL language and new types and interfaces can be defined based on a previous base TOSCA type (equivalent to a Class).



```
FP1:
  type: cloudify.nodes.nfv.FP
  # description: the path (CP01-CP11-CP12-CP21-CP22)
  properties:
    policy: { get_input: fp_policy }
    ODL_flow: { get_input: flow_definition }
  relationships:
    - type: cloudify.relationships.connected_to
      target: CP01
    - type: cloudify.relationships.connected_to
      target: CP11
    - type: cloudify.relationships.connected_to
      target: CP12
    - type: cloudify.relationships.connected_to
      target: CP21
    - type: cloudify.relationships.connected_to
      target: CP22

```

Figure 16

An ordered set of VLs, which is equivalent to a chain of CPs, defines a Forwarding Graph as depicted in Figure 16. It is based on an ordered set of TOSCA relationships.

Using these new TOSCA types we have a TOSCA design pattern for service abstraction.

Dynamic topology update

Cloudify, as a workflow engine, maintains a graph of the TOSCA representation of nodes and instances. As additional CPEs are added, they are introduced dynamically into the graph. Think of it as a dynamic Microservice graph where we can add or delete existing nodes. This is very useful and supports many use cases.

In the vCPE/SD-WAN use case, we start with an empty graph and grow it as new CPEs are added. In Figure 17, we see the first CPE added and additional ones are added as they are connected to the network and "call home". "Call home" is part of the ZTP (zero touch provisioning) process.

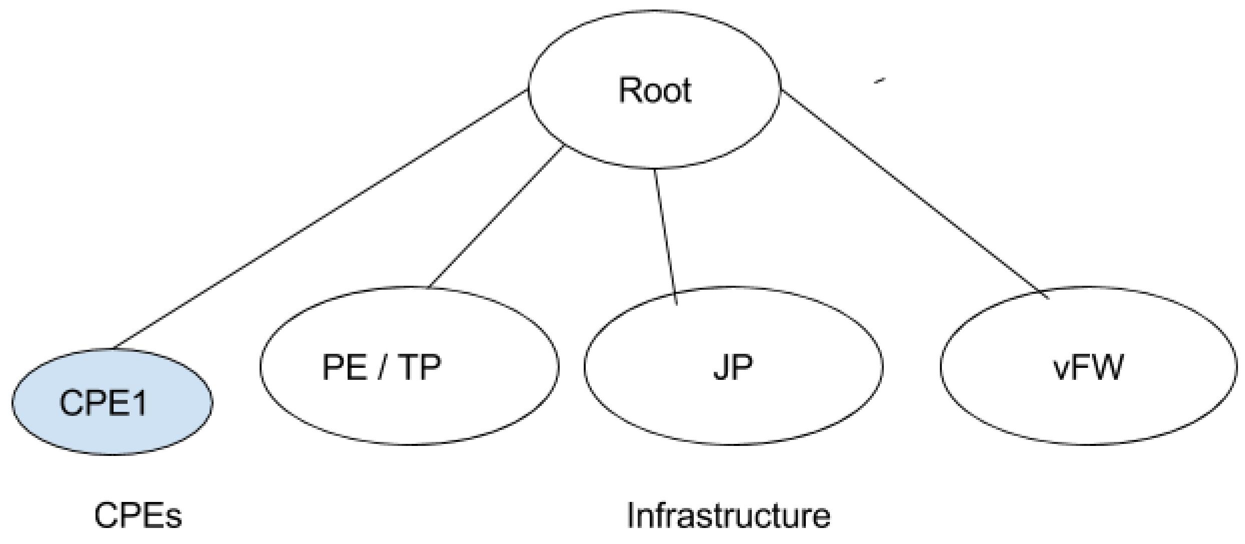


Figure 17

Dynamic topology updates support adding/deleting nodes from the graph, but also dynamically adding links, relationships in TOSCA, to the graph.

Use Case: Orchestration-first approach, utilizing existing infrastructure use case

To combine the different design patterns and best practices introduced here we demonstrate two use cases and their usage.

The first use case presents an open vCPE secured internet connection use case, utilizing existing provider infrastructure without the need to replace all the components end-to-end. Figure 18 represents the end result.

1. A new cloud infrastructure is provisioned
2. VNFs involved in the secure internet connection service are provisioned
3. Any VNF can be provisioned or chained in
4. A customer registers to the system through the Portal
5. A new tenant is provisioned
6. The customer connects remote branches (CPEs)
7. The customer can define the characteristics of each CPE connection including IPSEC overlays
8. CPEs are sent via email to remote branch and connected to the network
9. Cloudify configures the CPE and creates the service chains

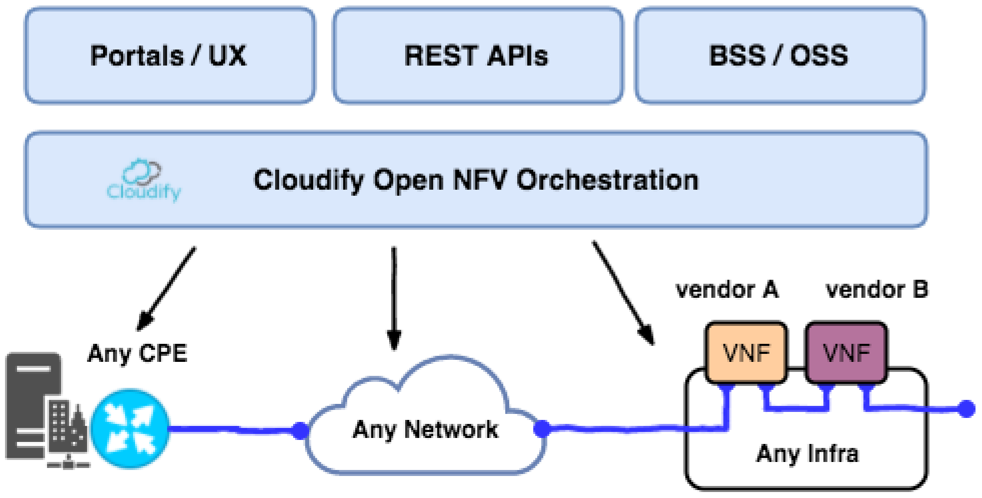


Figure 18

Let's see what TOSCA design patterns are involved:

1. VNF onboarding encapsulation

    a. You can onboard any VNF, it is an open architecture. For example, you can choose the vFW that fits you best - Fortigate, Palo Alto Networks, Checkpoint or even an open source one like pfSense
    b. Using the VNF onboarding pattern it’s easy to take any VNF, encapsulate it in a TOSCA blueprint, and onboard it to the VNFs catalog
    c. Lifecycle events for Init, Configure, Start, and Delete are exposed for VNF encapsulation
2. PNF encapsulation
    a. Any PNF along the path can be encapsulated and manipulated the same way as VNFs, which gives a simple approach to manipulate VNFs and PNFs the same way.
3. Basic service layered design pattern provisioning for the following layers:
    a. Master coordinator using the Proxy pattern to invoke and coordinate all blueprints
    b. Infrastructure provisioning
        i. TOSCA blueprint to provision infrastructure for all tenant customers
    c. Per tenant provisioning
        i. TOSCA blueprint to provision tenant resources for all tenants’ CPEs
    d. TOSCA blueprint per CPE
        i. Provision CPE resources
        ii. Establish service chains
        iii. Establish IPSEC overlays
4. Forwarding Graph and CPs pattern
    a. Could be added to abstract the service chaining mechanism
5. Dynamic deployment update patten
    a. To add CPEs on the fly as new ones are registered

Use Case: Service chaining Forwarding Graph abstraction

This use case presents the Forwarding Graph abstraction pattern. In this use case we use Openflow as the underlying mechanism for service chaining and traffic engineering.

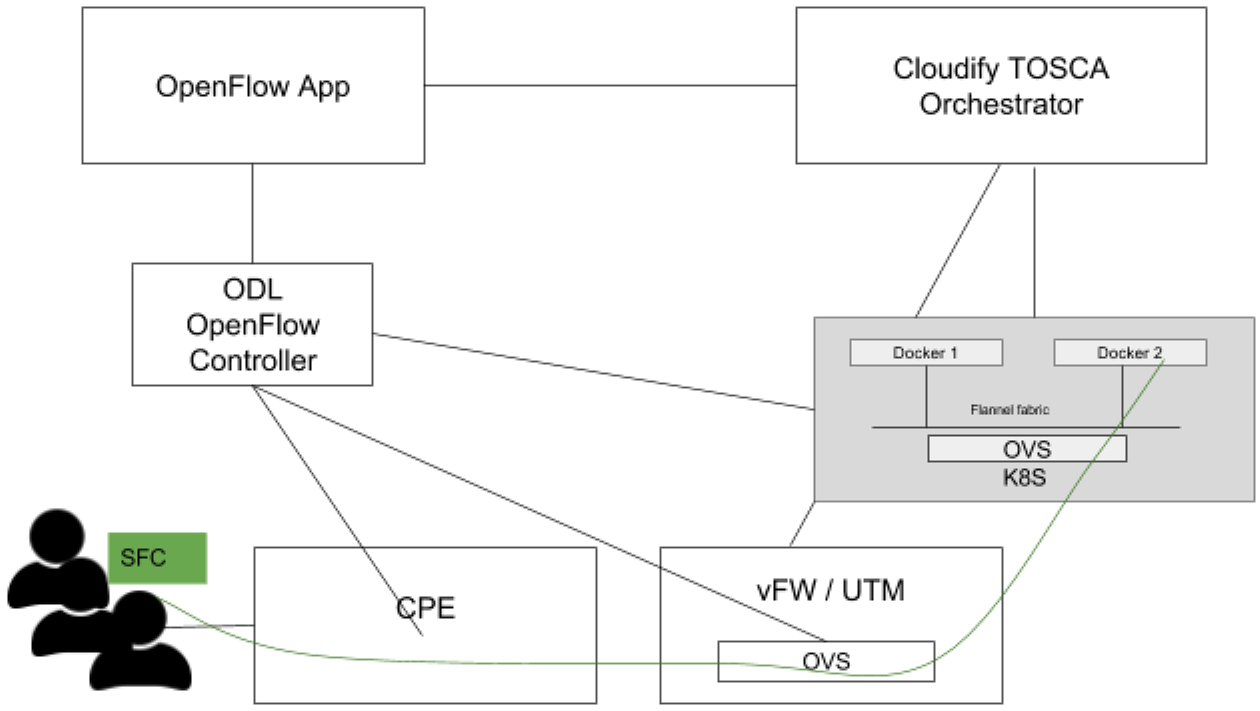


Figure 19

Utilizing SDN controller underlay abstraction

The Forwarding Graph abstraction is involved:

1. New TOSCA types are created
    a. TOSCA connection points
        i. Each CP calls the OpenFlow plugin to create a new entry in the OVS flow table
    b. TOSCA VLs - Virtual Links to connect to VNF VMs
    c. TOSCA Forwarding Graph as seen in Figure 16 to describe the SFC (Service Chain, seen in green in Figure 19)

Chaining VNFs, PNFs and Containers

For the creation of the end-to-end environment, the following TOSCA patterns are involved:

1. Proxy pattern
    a. Master coordinator to invoke and coordinate multiple blueprints
        i. Kubernetes provisioning
        ii. vFW and UTM onboarding
        iii. CPE setting
        iv. SFC creation

2. VNF encapsulation
    a. vFW and UTM onboarding
    b. Application Docker containers provisioning utilizing Cloudify's Kubernetes plugin

3. PNF encapsulation pattern
    a. CPE configuration - CPEs are described in a separate blueprint like a Microservice

4. Containers encapsulation pattern
    a. Docker Apps are described in a separate TOSCA blueprint per Docker application

This example demonstrates how to orchestrate a hybrid environment of VNFs, PNFs, and Docker containers as well as provision and manage a URL filtering + firewall service for remote end-users.

Summary

This paper describes a few useful TOSCA patterns:

1. VNF TOSCA encapsulation and onboarding
2. PNF TOSCA encapsulation pattern
3. Containers encapsulation pattern
4. A Proxy pattern to for blueprint composition
5. Basic service creation, layered TOSCA pattern
6. Forwarding Graph service chaining abstraction pattern
7. Dynamic topology update

Then we describe two use cases:

1. The first is an open vCPE solution based on the above design patterns. The open vCPE is an end-to-end solution which enables CSPs to choose any component and VNF they like, and Cloudify can onboard it, plug into it, provision the service, and create the service chains to make it work.

2. The second use case is also based on the above design patterns and shows how to abstract a service chain that utilize an underlay OpenFlow controller to establish a service chain across PNFs, VNFs, and Docker containers running under Kubernetes.