Closed Loop Design Proposal for R4

ONAP Casablanca allows users to create closed loops for the supported use cases, such as vFirewall (vFW), vLoadBalancer (vLB)/vDNS, and vCPE, but it makes it hard to create generic closed loops for new use cases. This problem is exacerbated by the fact that existing closed loops hardcode part of their logic in the Policy Engine and/or the APPC controller. For Dublin release (R4), we plan to redesign how closed loops are built, focusing on generality and flexibility.

The following sections propose approaches to build closed loops in ONAP R4.

***NOTE: The description below is not intended to catch all the flows and calls between ONAP components. Its only purpose is to describe options at high level. Detailed requirements and flows must be provided in separate documents.***

# Closed Loop Design and Deployment from CLAMP

Users connect to CLAMP to define operational and configuration policies that are part of a closed loop.

## Operational Policy

When building the operational policy, CLAMP allows users to select the “Target Type”, which represents the type of element that will receive the action as part of closed loop (Figure 1).



Figure 1 Target type

Although many options are available, VNF is the only type used in practice today, even when the action is executed against a VF Module or VM. CLAMP GUI also shows the “Target ResourceID” dropdown list, as shown in Figure 2. This parameter is currently not used.



Figure 2 Target Resource ID

One of the goals for R4 is to allow users to design closed loops for different resources in the service model. For example, a user may create a closed loop to scale a VF module in/out according to traffic load and another closed loop to change the configuration of a VNF according to traffic patterns. To do so, we propose to enhance CLAMP in the following way:

* The Target Type list shall contain different types, such as VNF, VF Module, VM, and PNF. VM and PNF may not be supported in R4. If so, they may temporarily be removed from the list.
* The Target ResourceID list shall show all the available options in the service model for a given target type.

For example, consider the following TOSCA template from the vFW use case. It has two VNFs (the vFW and the packet generator), with a VF Module each.

topology\_template:

 node\_templates:

 vFirewall 0:

 type: org.openecomp.resource.vf.Vfirewall

 metadata:

…

 vPacketGen 0:

 type: org.openecomp.resource.vf.Vpacketgen

 metadata:

…

 groups:

 vpacketgen0..Vpacketgen..base\_vpkg..module-0:

 type: org.openecomp.groups.VfModule

 metadata:

 vfModuleModelName: Vpacketgen..base\_vpkg..module-0

 vfModuleModelInvariantUUID: 72440136-4b0f-4fc4-ad18-3240f93daea0

 vfModuleModelUUID: d1553e83-557d-4a43-8b60-03c3f876a88a

 vfModuleModelVersion: '1'

 vfModuleModelCustomizationUUID: eeb36aa1-c171-4ab1-a89c-58ba20497db7

 properties:

 min\_vf\_module\_instances: 1

 vf\_module\_label: base\_vpkg

 max\_vf\_module\_instances: 1

 vf\_module\_type: Base

 isBase: true

 initial\_count: 1

 volume\_group: false

 vfirewall0..Vfirewall..base\_vfw..module-0:

 type: org.openecomp.groups.VfModule

 metadata:

 vfModuleModelName: Vfirewall..base\_vfw..module-0

 vfModuleModelInvariantUUID: ebf8102d-e5e1-4203-a67b-5e7632492537

 vfModuleModelUUID: a7e70bad-87b7-42df-8c7f-6312b8db58eb

 vfModuleModelVersion: '1'

 vfModuleModelCustomizationUUID: 41d31ad2-ed16-448f-a80b-5c3770488f0a

 properties:

 min\_vf\_module\_instances: 1

 vf\_module\_label: base\_vfw

 max\_vf\_module\_instances: 1

 vf\_module\_type: Base

 isBase: true

 initial\_count: 1

 volume\_group: false

The resources in green represent VNFs, while those in blue represent VF modules. Based on this model, when a user select VNF from the Target Type list, the Target ResourceID list should contain vFirewall 0 and vPacketGen 0. In the same way, if a user selects VF Module from the Target Type list, then the Target ResourceID list should contain vpacketgen0..Vpacketgen..base\_vpkg..module-0 and vfirewall0..Vfirewall..base\_vfw..module-0. This way, users can specify which resource the closed loop refers to.

The green and blue items above are NOT resource IDs, but they are human-readable and hence easier to understand. Note that the Target ResourceID list should contain only resources for which is possible to create a closed loop, given a specific “Recipe”. For example, ONAP doesn’t allow to scale in or out base VF modules. As such, if the recipe is “VF Module Create” or “VF Module Delete”, the Target ResourceID list should contain only the ID (or name) of the VF modules whose “isBase” field is set to false. In the example above, none of the VF modules (vpacketgen0..Vpacketgen..base\_vpkg..module-0 and vfirewall0..Vfirewall..base\_vfw..module-0) should be listed if the recipe is “VF Module Create” or “VF Module Delete” because isBase is set to true for both modules.

Consider the following snippet of Tosca model, taken from the vLB/vDNS use case. It has a single VNF with two VF modules (base VF module plus a non-base VF module used for scaling).

 vloadbalancerms0..Vloadbalancerms..dnsscaling..module-1:

 type: org.openecomp.groups.VfModule

 metadata:

 vfModuleModelName: Vloadbalancerms..dnsscaling..module-1

 vfModuleModelInvariantUUID: 82165a0f-430c-463c-bcef-e106d02af6c9

 vfModuleModelUUID: ddf753e3-c8cc-4247-b902-213fa2083e61

 vfModuleModelVersion: '1'

 vfModuleModelCustomizationUUID: 3878cce1-47c1-4ec8-812a-3e71e908f58e

 properties:

 min\_vf\_module\_instances: 0

 vf\_module\_label: dnsscaling

 max\_vf\_module\_instances: 50

 vf\_module\_type: Expansion

 isBase: false

 initial\_count: 0

 volume\_group: false

 vloadbalancerms0..Vloadbalancerms..base\_vlb..module-0:

 type: org.openecomp.groups.VfModule

 metadata:

 vfModuleModelName: Vloadbalancerms..base\_vlb..module-0

 vfModuleModelInvariantUUID: b28b4cfa-f793-445e-a8ea-b20299f5961c

 vfModuleModelUUID: 86c2fecc-9840-4341-8357-887049e84e53

 vfModuleModelVersion: '1'

 vfModuleModelCustomizationUUID: 8dc6ef75-7795-4ebd-a3d8-633be62fa698

 properties:

 min\_vf\_module\_instances: 1

 vf\_module\_label: base\_vlb

 max\_vf\_module\_instances: 1

 vf\_module\_type: Base

 isBase: true

 initial\_count: 1

 volume\_group: false

In this case, if the recipe is “VF Module Create” or “VF Module Delete”, based on the value of “isBase” flag (in yellow), the Target ResourceID list should contain vloadbalancerms0..Vloadbalancerms..dnsscaling..module-1 but not vloadbalancerms0..Vloadbalancerms..base\_vlb..module-0.

Finally, CLAMP shall pass VNF or VF module model metadata and property information defined in the TOSCA model to Policy. Policy and CLAMP Teams should work on CLAMP-Policy API modifications to support this strategy.

## Changes in the Policy

The operational policy YAML would be modified as below. The new fields are highlighted –

controlLoop:

  version: 2.0.0

  controlLoopName: ControlLoop-vDNS-6f37f56d-a87d-4b85-b6a9-cc953cf779b3

  services:

    - serviceName: d4738992-6497-4dca-9db9

      serviceInvariantUUID: dc112d6e-7e73-4777-9c6f-1a7fb5fd1b6f

      serviceUUID: 2eea06c6-e1d3-4c3a-b9c4-478c506eeedf

  trigger\_policy: unique-policy-id-1-scale-up

  timeout: 1200

policies:

  - id: unique-policy-id-1-scale-up

    name: Create a new VF Module

    description:

    actor: MSO

    recipe: VF Module Create

    target:

      type: VFModule

 modelInvariantId: 90b793b5-b8ae-4c36-b10b-4b6372859d3a

 modelVersionId: 2210154d-e61a-4d7f-8fb9-0face1aee3f8

 modelName: SproutScalingVf..scaling\_sprout..module-1

 modelVersion: 1

 modelCustomizationId: 3e2d67ad-3495-4732-82f6-b0b872791fff

    retry: 0

    timeout: 1200

    success: final\_success

    failure: final\_failure

    failure\_timeout: final\_failure\_timeout

    failure\_retries: final\_failure\_retries

    failure\_exception: final\_failure\_exception

    failure\_guard: final\_failure\_guard

The highlighted fields would be newly added in the operation policy YAML. This policy YAML would be distributed by CLAMP to the Policy.

The Policy PDP engine would need to process these additional fields and pass the same in the “modelInfo” JASON structure of the scaleout SO API payload.

## Configuration Policy

The configuration policy is used to specify one or more conditions that trigger a closed loop execution and the event name linked to that specific closed loop. This is used in DCAE when microservices receive network telemetry via the collectors. Current limitations include hardcoded event names and metrics (Figures 3 and 4, respectively).



Figure 3 Event Name selection



Figure 4 Metric selection

Users specify the event name in the EventName field in the CLAMP popup window (Figure 3). The event names used today (e.g. “vLoadBalancer”, “vFirewallBroadcastPackets”, and “Measurement\_vGMUX”) refer to existing use cases. If a user wants to build a new use case, they have to reuse one of the existing names. In the same way, users have limited choices when defining thresholds for threshold crossing applications (TCAs). At runtime, the TCA microservice in DCAE will check whether the threshold is crossed. If so, a new event is generated, so as to trigger closed loop execution in Policy. Currently, the CLAMP GUI only shows two metrics, “packetLossRate” and “receivedTotalPacketDelta”, as shown in Figure 4.

For R4, CLAMP will change the way configuration policies are created. Instead of using predefined options, as shown in Figures 3 and 4, CLAMP GUI will be created based on the policy model distributed by SDC as part of the CSAR package. As such, user can select event name, schema type, metric, closed loop status, etc. based on default values defined in the policy model, if defined. In fact, the fields in the policy model may or may not have default values. As an example, the following two definitions of the “fieldPath” object used for specifying metrics should be both valid.

Definition 1:

{

 "name":"fieldPath",

 "description":"Json field Path as per CEF message which needs to be analyzed for TCA",

 "type":"string",

 "value":"",

 "constraints":[

 {

 "valid\_values":[

 "receivedTotalPacketsDelta",

 "packetLossRate"

 ]

 }

 ]

}

Definition 2:

{

 "name":"fieldPath",

 "description":"Json field Path as per CEF message which needs to be analyzed for TCA",

 "type":"string",

 "value":""

}

When Definition 1 is used for a given variable (like “fieldPath” in the example above, but it can be any other variable), CLAMP GUI shall show a dropdown list containing the default values specified in the policy model (“receivedTotalPacketsDelta” and “packetLossRate” in the example above). When Definition 2 is used, instead, CLAMP GUI shall show a simple text box that users have to fill with the desired value (being default values not specified in the policy model). In both cases, CLAMP shall ensure that the fields are not empty when the user clicks the “Close” button (or the equivalent button in case “Close” gets redefined in R4).

# DCAE Processing

At runtime, DCAE evaluates the configuration policy built and deployed from CLAMP. The event name specified during creation of the configuration policy is linked to a control loop. When a policy is violated (for example, if using TCA, a threshold is crossed), DCAE generates an event that triggers closed loop operations in the Policy Engine. This event carries the reporting entity name, a field in the VES message that specifies who reported that telemetry. The reporting entity name is also used to “enrich” the event that DCAE sends to Policy. Enrichment means that additional information about that reporting entity is collected from AAI and sent to Policy. DCAE supports two types of enrichment:

* VM-based: users can use this enrichment by selecting VM in the “Control Loop Schema Type” in the CLAMP GUI (Figure 4). An example of VM-based enrichment is provided below:

"AAI":{

 "vserver.prov-status":"ACTIVE",

 "vserver.resource-version":"1548447279970",

 "vserver.is-closed-loop-disabled":false,

 "vserver.vserver-name2":"vlb-ms-0125-1",

 "vserver.vserver-id":"b63b83f5-9ad4-49d8-8d56-78284e29f47c",

 "vserver.vserver-selflink":"http://10.12.25.2:8774/v2.1/41d6d38489bd40b09ea8a6b6b852dcbd/servers/b63b83f5-9ad4-49d8-8d56-78284e29f47c",

 "vserver.in-maint":false,

 "vserver.vserver-name":"vlb-ms-0125-1"

 }

* VNF-based enrichment: users can use this enrichment by selecting VNF in the “Control Loop Schema Type” in the CLAMP GUI (Figure 4). An example of VNF-based enrichment is provided below:

"AAI":{

      "generic-vnf.in-maint":false,

      "generic-vnf.is-closed-loop-disabled":false,

      "generic-vnf.orchestration-status":"Created",

      "generic-vnf.prov-status":"ACTIVE",

      "generic-vnf.resource-version":"1504896046185",

      "generic-vnf.service-id":"e8cb8968-5411-478b-906a-f28747de72cd",

      "generic-vnf.vnf-id":"63b31229-9a3a-444f-9159-04ce2dca3be9",

      "generic-vnf.vnf-name":"vCPEInfraVNF13",

      "generic-vnf.vnf-type":"vCPEInfraService10/vCPEInfraService10 0"

   }

The reporting entity name is a VM (or vServer in ONAP terminology). As such, the VNF-based enrichment requires the reporting VM to have the same name as the VNF. This requirement may not be acceptable all the times, so for R4 we need to understand better how DCAE handles VNF vs. VM events, alarms, etc. Because of the limitation described above, we may decide to use VM-based enrichment only.

# Closed Loop Execution in the Policy Engine

Once Policy receives an event from DCAE, it starts closed loop by calling the closed loop executor (or “Actor”) and specifying what operations must be executed. Policy supports two actors: SO and APPC. SO is used when closed loop requires to create or delete network resources. At this time, SO is used for scaling out VF modules (Policy Recipe: “VF Module Create”). In the future, it can be used for scaling in VF modules and migrate VNFs. APPC is used to change the status of running resource, for example modifying VNF configuration or restarting a VM.

Policy uses SO and APPC direct APIs. Policy needs to have all the necessary information to fill the input request. When using APPC, according to the APPC LCM APIs, Policy needs to provide the VNF instance ID in case the operation is executed against a VNF, or the VNF instance ID and the VM ID (assigned by the cloud platform, for example OpenStack) in case the operation is executed against a VM.

When using SO for creating new VF modules (e.g. scale out), Policy needs to provide model IDs of service, VNF, and VF module (“model-invariant-id”, “module-customization-id”, “model-version-id”, “model-name” defined in the TOSCA model – not to be confused with the ID of the running instances created out of that model). In addition, Policy needs to provide the instance IDs of the service and VNF.

For R3 and before, this required some manual workaround (e.g. creation of a dummy VF module in AAI that contains all the required model IDs). To eliminate those workarounds, Policy needs to have access to model IDs. Policy will receive this information from CLAMP as part of operational policy creation and deployment. Policy and CLAMP Teams should work on CLAMP-Policy API modifications to support this strategy.

# AAI Custom Query for Closed Loop

For ONAP releases R1-R3, Policy uses the reporting entity name provided by DCAE to execute a named query against AAI. Policy then browses the returned JSON object to retrieve instance IDs of the service, VNFs, or VF modules, according to the closed loop operations it has to execute. Named queries are being deprecated in R4 and completely removed in R5. As such, we propose an alternative strategy in R4. This will apply to closed loop execution via APPC and SO.

AAI is replacing named queries with custom queries. However, the current custom query for closed loop offers only partial visibility into a service instance. Given the name of a vServer (reporting entity name from DCAE), the custom query returns information about the VNF it belongs to. With the term *information* we mean instance IDs, model IDs, and other fields that can be found in the AAI inventory, such as prov-status, in-maint, and relationship-lists. The query also returns information about all the VF Modules of that VNF and the service instance ID. While this allows to execute closed loop operations against the VF module and VNF which the reporting entity belongs to, it doesn’t allow to execute closed loop operations against other possible VNFs (and their modules) that are part of the same service.

Consider the service hierarchy in Figure 5. Let’s assume that the grey circle (vServer) is the reporting entity. The current custom query against AAI will return information about the yellow and purple VF modules (triangles), the orange VNF (square), and the service (blue oval). Thus, a closed loop operation is possible against those VNF and VF modules. On the contrary, the query doesn’t return any info about the subtree rooted at the green VNF (square). As such, operations against the VNF and VF modules that are part of that subtree aren’t possible.

To achieve flexibility, users should be allowed to execute closed loop operations against any VNF or VF module in the same service (at least) that contains the reporting entity or vServer. As such, we propose to slightly change the custom query for closed loop in the following way: given a vServer name, the custom query should return information about ALL the VNFs and VF modules that belong to the service instance.

With reference to Figure 5, if the reporting vServer name (gray circle) is used to issue a custom query against AAI, the query should return information about the service instance (blue circle), the VNFs (orange and green squares), and VF modules (yellow, purple, red, and black triangles).



Figure 5 Service hierarchy in AAI

## Custom Query Response: JSON Object Structure

With respect to Figure 5, AAI returns the follow JSON object when using the gray vServer name as argument of the closed loop custom query (current implementation):

{

 "results":[

 {

 "vserver":

{ info about grey vServer }

 },

 {

 "generic-vnf":

{ info about orange VNF},

 "vf-modules":[

 {

 "vf-module":

{ info about yellow VF module }

 },

 {

 "vf-module":

{ info about purple VF module }

 }

 ]

 },

 {

 "service-instance":

{ info about service instance }

 },

 {

 "vf-module":

{ info about yellow VF module }

 },

 {

 "vf-module":

{ info about purple VF module }

 }

 ]

}

Note that all the details about the resources (service, VNFs, and VF modules), as well as extra information that the actual JSON object contains, such as tenants, flavors, images, etc. are not reported for clarity’s sake.

As discussed in the previous section, the returned object provides only partial view into the service instance. We propose to modify the JSON object that AAI returns as output of closed loop custom query as follows:

{

 "results":[

 {

 "vserver":{

 { info about grey vServer }

 }

 },

 {

 "generic-vnf":

{ info about orange VNF},

 "vf-modules":[

 {

 "vf-module":

{ info about yellow VF module }

 },

 {

 "vf-module":

{ info about purple VF module }

 }

 ]

 },

 {

 "service-instance":{

 { info about service instance }

 "generic-vnfs":{

 "generic-vnf":[

 { info about orange VNF },

 { info about green VNF }

 ]

 },

 "vf-modules":{

 "vf-module":[

 { info about yellow VF module },

 { info about purple VF module },

 { info about red VF module },

 { info about black VF module }

 ]

 }

 }

 }

 ]

}

The first block of the JSON object (highlighted in green) is the same as the one currently returned. No modifications are expected. Assuming the vServer name is the gray circle in Figure 5, the green block of the JSON object reports info about VNF and VF-modules in the vicinity of that vServer. This is expected to not change.

The second part of the JSON object (highlighted in yellow) includes the modifications that are part of this proposal. While the current implementation only reports information about the service, we propose to change that part of the JSON object so as to report a tree-based representation of the service, VNF and VF module instances. Each indented item in the following list represents an inner object/array of the custom query response (the part in yellow):

* **service-instance**: generic information about the service, such as service-instance-id, service-instance-name, service-type, relationship-lists, etc.
	+ **generic-vnfs**: array of generic-vnf objects
		- **generic-vnf**: generic information about the VNF, such as vnf-id, vnf-name, vnf-type, relationship-lists, etc. A generic-vnf element SHOULD NOT contain vf-module objects. All the generic-vnf objects in the service should be part of this array
	+ **vf-modules**: array of vf-module objects
		- **vf-module**: generic information about the VF module, such as vf-module-id, vf-module-name, etc. All the vf-module objects in the service should be part of this array

At runtime, Policy will look for the service-instance element in the JSON object returned by AAI and will browse that element to retrieve service IDs of VNFs and VF modules. These IDs will be used to populate input requests to SO or APPC, depending on the specific closed loop actor.