# Table of Contents

- Introduction .................................................................................................................. 3
- Terminology .................................................................................................................. 3
- Defining a New Service Type .......................................................................................... 3
- Changing Service Models ............................................................................................... 6
- Benefits of the Tail-f Approach .................................................................................... 7
- Appendix: A Detailed Example ....................................................................................... 8
Introduction

The purpose of this solution brief is to illustrate the processes of introducing new service types and modifying existing service types with Tail-f NCS.

The main difference in approach with Tail-f NCS is that services are defined in data models, rather than hard-coded in software. Tail-f NCS then uses these data models to automatically generate the corresponding user interfaces, north-bound APIs, database schemas, and south-bound command sequences.

This whitepaper describes the development process used when defining new network services and modifying existing services. The model-driven approach used with Tail-f NCS is compared to that of traditional provisioning and orchestration systems. A detailed example describing the development process of implementing an L3 MPLS VPN service in a network with Cisco IOS CE routers and Cisco IOS-XR PE routers is given in the Appendix.

Terminology

- **Service type**: A specific type of service like L2-VPN, L3-VPN, etc.
- **Service instance**: A specific instance of a service type like “ACME L3-VPN”.
- **Service model**: A service model specifies the attributes of the service type. Service models are used in different contexts/systems and therefore have slightly different meanings. In the context of Tail-f NCS a service model is a black-box specification of the attributes required to instantiate the service. This is different from service models in ITIL-based CMDBs or OSS inventory systems, where a service model is more of a white-box model that describes the complete structure.
- **Device configuration**: Network devices are configured to perform network functions. Every service instance results in corresponding device configuration changes. The dominating way to represent and change device configurations in current networks are CLI representations and sequences.

Defining a New Service Type

Tail-f NCS addresses the complete automation chain from a service request (issued by an Order Management system or a self-service portal) all the way to committed device configuration changes in the network.

Every step in this automation chain is explicitly and declaratively defined in a way that can be performed by network engineers. Nothing is handed over to software development projects or ad-hoc manual configuration.
The process of defining a new service type starts with asking two fundamental questions:

1. When we create the service, what parameters do we enter? That is, what are the parameters sent from the Order Management System (or, equivalently, what are the parameters entered in a self-service port)?

2. What are the resulting device configurations for the service.

Informal service models. The answer to the first question is captured in a service model. The first version of this model is informal bullet items on paper (here and in the rest of this whitepaper, L3-VPN is used as the example service):

- VPN name
- AS number
- End-points:
  - CE device, interface, IP address
  - PE device, interface, IP address

In this first version we list the PE devices as input. However, if we make sure that Tail-f NCS knows about CE-PE links then the PE devices don’t have to be given as input. This illustrates the power of model-driven systems: Tail-f NCS can handle both scenarios. The decision boils down to the classical question of how system borders are defined.

In this example the model where CEs and PEs are input to the service model will be used.
Device configurations. To answer the second question above – what are the resulting device configurations in the network corresponding to a service instance – we work with network engineers that know how devices would be configured manually to realize a L3-VPN.

This step can be done either after or before informally defining the service model and its attributes.

It might seem that device configurations is a too low-level concern during service implementation. It is not – rather it is the most important step, since it is required to make the service operational in the network. Customers are not happy just because the service exists in the BSS and bills are sent. It must be up and running in the network.

The following is a typical L3-VPN configuration for a Cisco PE router (iOS-XR) and a Cisco CE router (IOS):

Mapping services into device configurations. The next step in the process of defining a new service type is to associate each service attribute with the corresponding parts of the device configurations. We refer to this as mapping the service model into device configurations.

YANG data models. The next step is to transform the informal service model into a formal model expressed in the standardized YANG data modeling language (RFC 6020).

The devices used by the service also have YANG data models describing their possible configurations. These data models are provided either by the device vendor or by Tail-f.
Data model mapping. The final step is to express the mapping of the service model into device configurations as a data model transformation. Data in the service model is mapped into data in the device data models. There are two ways to specify this mapping in Tail-f NCS:

- Declarative templates: In many cases a straightforward template that maps service attributes to abstract device configurations is sufficient.
- Programmatic mapping: In some cases algorithmic expressions are needed to map service attributes to abstract device configurations. In these cases a programmatic mapping can be used.

Generating device-specific configuration commands. Tail-f NCS automatically translates operations on YANG device models into concrete command sequences that are sent to the devices in the network (vendor-specific CLI commands, NETCONF commands, SNMP requests, REST calls, etc). Thus, there is no need for CLI scripting or complex “adapter” development for this task.

Changing Service Models

To allow for rapid responses to new market requirements it is crucial that services can easily be modified.

Tail-f NCS has explicit support for changes to service models and automatic upgrade of the system accordingly. Service models and corresponding templates are part of a package structure that can be loaded and reloaded to Tail-f NCS at run-time. So modifying the template or service model can be done at any time and the package can be reloaded. The changes will take immediate effect, including database schema upgrade.
In the L3-VPN example, assume that we realize that a “Customer ID” field is needed for the service model. We add the field to the service model and ask Tail-f NCS to upgrade the packages. Now all existing and future instances of the L3-VPN service have an empty Customer ID field that can be configured.

**Benefits of the Tail-f Approach**

The model-driven design of Tail-f NCS provides several benefits compared to traditional provisioning or orchestration systems, which require complex software projects for service implementation.

**Correctness**

With Tail-f NCS the specification (data model) is the implementation.

With other systems, there is always a significant risk that software developers misinterpret informal service specifications and implement them incorrectly. This can lead to errors that are very difficult to find and correct in a large network.

**Completeness**

With Tail-f NCS the model mapping specify only how a service is created; the other CRUD operations (Read, Update, Delete) are automatically generated by the system.

With other systems, only a subset of the CRUD operations are typically implemented completely. This leads to a need for manual configuration to complement the tasks supported by the system. This, in turn, introduces the possibilities of human mistakes and slows down operations.

**Service agility**

With Tail-f NCS the north-bound and south-bound interfaces, as well as database schemas, are automatically generated from data models. The development of service data models and the model mapping is done by a network engineer in a matter of days.

With other systems, these interfaces, schemas and CRUD operations are implemented in software projects that typically run for weeks or months.
Appendix: A Detailed Example

Overview

This example describes the development process of implementing an L3 MPLS VPN service in a network with Cisco IOS CE routers and Cisco IOS-XR PE routers. The process described below typically takes a couple of days for a network engineer. The service model is 90 lines of YANG and the service mapping template is 180 lines of XML. This result in a fully functional L3 MPLS VPN provisioning system with service-aware north-bound REST interface, CLI, web interface, and database schema, as well as south-bound interfaces to the routers. The system has automatically derived CRUD operations like adding end-points, changing AS number, and decommissioning of service instances with automatic clean-up of associated device configurations.

The first question is whether to start with an informal service model or with the device configurations. In many cases it makes sense to start bottom-up by looking at the device configurations done by the network engineers. In other cases it makes sense start top-down by looking at parameters in Order Managers or self-service portals to sketch a service model.

In this example we will start bottom-up with the device configurations for the L3 MPLS VPN.

Device configuration

We start by provisioning a service instance in a network of real or simulated devices (NCS comes with a device simulator called ncs-netsim). We use the VRF name “volvo” and the Service Provider AS 100.

```
interface GigabitEthernet0/1
description Link to CE router
ip address 10.1.1.2 255.255.255.252
exit

vrf volvo
address-family ipv4 unicast
import route-target 65101:1
exit

interface GigabitEthernet0/1
description Link to PE
ip address 10.1.1.1 255.255.255.252
exit

vrf volvo
address-family ipv4 unicast
export route-target 65101:1
exit

global
router bgp 100
vrf volvo
ip 65101:1
address-family ipv4 unicast
exit
neighbor 10.1.1.1 remote-as 65101
address-family ipv4 unicast
as-override
exit
exit

interface GigabitEthernet0/20
description Local network
ip address 192.168.1.0 255.255.255.0
exit

global
router bgp 65101
neighbor 10.1.1.2 remote-as 100
neighbor 10.1.1.2 activate
redistribute connected!```
CE Router (IOS):

```bash
interface GigabitEthernet0/1
description Link to PE
ip address 10.1.1.1 255.255.255.252
exit
interface GigabitEthernet0/20
description Local network
ip address 192.168.1.0 255.255.255.0
exit
router bgp 65101
neighbor 10.1.1.2 remote-as 100
neighbor 10.1.1.2 activate
redistribute connected
```

PE Router (IOS-XR):

```bash
vrf volvo
address-family ipv4 unicast
   import route-target 65101:1
exit
interface GigabitEthernet 0/0/0/1
description link to CE
ipv4 address 10.1.1.2 255.255.255.252
exit
router bgp 100
vrf volvo
   rd 65101:1
   address-family ipv4 unicast
   neighbor 10.1.1.1 remote-as 65101
   neighbor 10.1.1.2 activate
   as-override
   redistribute connected
```

So the above is a concrete representation of the service in the network. The first step is to load these configurations into NCS. There are two ways to do this:

1. Create the configuration on the devices and sync the configuration into NCS. The devices can be real devices or ncs-netsim can be used to create the configuration on simulated devices.

2. Use NCS to set up the configuration on the devices (simulated with ncs-netsim or real).

The following is a command sequence to use ncs-netsim to setup the configuration on simulated devices and then synchronize into NCS.

Start an IOS CLI session towards ce0:

```
$ ncs-netsim cli-i ce0
```

Enable configuration:

```
admin connected from 127.0.0.1 using console on wallair.
local
ce0> enable
ce0# configure
Enter configuration commands, one per line. End with CNTL/Z.
```

Do the following configuration:

```
interface GigabitEthernet0/1
description Link to PE
ip address 10.1.1.1 255.255.255.252
exit
interface GigabitEthernet0/20
description Local network
ip address 192.168.1.0 255.255.255.0
exit
router bgp 65101
neighbor 10.1.1.2 remote-as 100
neighbor 10.1.1.2 activate
redistribute connected
```

```
Exit and start an IOS-XR CLI session towards pe0:

$ ncs-netsim cli-c pe0

Enable configuration mode:

# config
Entering configuration mode terminal

Enter the following configuration:

vrf volvo
  address-family ipv4 unicast
    import route-target
      65101:1
    exit
  export route-target
    65101:1
    exit
exit
interface GigabitEthernet 0/0/0/1
  description link to CE
  ipv4 address 10.1.1.2 255.255.255.252
exit
router bgp 100
  vrf volvo
    rd 65101:1
    address-family ipv4 unicast
    exit
  neighbor 10.1.1.1
    remote-as 65101
    address-family ipv4 unicast
    as-override
    exit
  exit
exit
exit

So the above sequence illustrated the current manual configuration process, working directly with the (simulated) devices. The next steps will import that into NCS and use it as a base for service mapping later.

Start the NCS CLI:

$ ncs_cli -u admin -C

admin connected from 127.0.0.1 using console on wallair. local

Import the configuration of all devices (ce0 and pe0) into NCS and enter config mode:

# devices sync-from
# config
Entering configuration mode terminal

This completes the first step, establishing an exact representation of the desired device configuration in NCS. It is now possible to read and manipulate the configuration locally in NCS. For example, the following command inspects the configuration of device ce0:

# show full-configuration devices device ce0 config

NCS abstracts away the underlying device CLIs and can present a generic device tree. For example the following command views the configuration of device ce0 in XML format:

# show full-configuration devices device ce0 config | display xml
Service model

The next question is which parts of the device configurations should be parameters in the service model. Reasonable candidates are shown bold face in the following figure:
That is, the VPN service attributes are:

- Name
- AS number
- End-points
  - CE routers
    - LINK: Interface Name, Number, IP-number
  - LOCAL: Interface Name, Number, IP-number
- PE routers
  - LINK: Interface Name, Number, IP-number

The following figure shows the structure of the data model:

```
module l3vpn

list l3vpn {
  key name;
  leaf name {
    type string;
  }
  list endpoint {
    key "id";
    leaf id {
      type string;
    }
    leaf as-number {
      description "AS used within all VRF of the VPN";
      mandatory true;
      type uint32;
    }
    container ce {
      leaf device {
        mandatory true;
        type leafref {
          path "/ncs:devices/ncs:device/ncs:name";
        }
      }
      container local {
        uses endpoint-grouping:
      }
      container link {
        uses endpoint-grouping;
      }
    }
    container pe {
      leaf device {
        mandatory true;
        type leafref {
          path "/ncs:devices/ncs:device/ncs:name";
        }
      }
      container link {
        uses endpoint-grouping;
      }
      container link {
        uses endpoint-grouping;
      }
    }
  }
}
```

This is modeled in YANG as follows:

```
module l3vpn {
  ...
  list l3vpn {
    key name;
    leaf name [type string];
    list endpoint {
      key "id";
      leaf id [type string];
      leaf as-number {
        description "AS used within all VRF of the VPN";
        mandatory true;
        type uint32;
      }
    }
    container ce {
      leaf device [mandatory true;
                  type leafref {
                      path "/ncs:devices/ncs:device/ncs:name";
                  }]
      container local {
        uses endpoint-grouping:
      }
      container link {
        uses endpoint-grouping;
      }
    }
    container pe {
      leaf device [mandatory true;
                  type leafref {
                      path "/ncs:devices/ncs:device/ncs:name";
                  }]
      container link {
        uses endpoint-grouping;
      }
    }
  }
}
```
The **endpoint** grouping is a YANG grouping (similar to a macro) expanding to:

```yaml
grouping endpoint-grouping {
  leaf interface-name {
    type string;
  }
  leaf interface-number {
    type string;
  }
  leaf ip-address {
    type inet:ipv4-address;
  }
}
```

This grouping is used at several places in the service model to refer to an interface name, number and IP number, such as GigabitEthernet 0/11 10.10.1.0/24

The YANG model above basically says:

- There is a list of VPN service instances
- Each VPN instance is identified by “name”
- Each VPN instance has a list of end-points with:
  - CE local interface and IP address
  - CE link to PE interface and IP address
  - PE interface and IP address

At this point it makes sense to load the initial service model into NCS in order to test various north-bound interfaces (REST, CLI, web interface). This provides support for an iterative modify-reload-test development cycle, where the service model is incrementally refined and the service mapping is defined. This makes it easy to test user interfaces and programmatic APIs to validate the service model attributes and the service mapping.

### Service mapping

The next step is to define the mapping of service attributes to device configurations. A low-tech but reliable way to do this is by defining a “napkin mapping” on paper, where lines from the service attributes are drawn to the device configurations.

Once the mapping has been defined on paper it is time to put it in an XML template. This is a straightforward three-step process:

**Step 1.** Save the device configuration as an XML document:

```
# show full-configuration devices device ce0 config | display xml
# show full-configuration devices device pe0 config | display xml
```

Merge these two outputs into one XML file, removing parts that are not related to the service according to the “napkin mapping”. The following is a snippet of the XML file (values corresponding to service attributes are in bold face):

**Step 2.** Replace the values identified in the “napkin mapping” with variables representing attributes from the service model:
In the above XML template snippet the bold-face variables are paths in the service model. Recall that the service model has the following structure:

```
<config xmlns="http://tail-f.com/ns/config/1.0">
  <devices xmlns="http://tail-f.com/ns/ncs">
    <name>ce0</name>
    <config>
      <ip xmlns="urn:ios">
        <interface xmlns="urn:ios">
          <GigabitEthernet>
            <name>0/1</name>
            <description>Link to PE</description>
            <ip>
              <primary>
                <address>10.1.1.1</address>
                <mask>255.255.255.252</mask>
              </primary>
            </ip>
          </GigabitEthernet>
        </interface>
      </ip>
    </config>
  </devices>
</config>
```

Thus, the paths in the template above are paths in the service model.

The template can support different device types. In the template above, the device type Cisco IOS is indicated by the namespace attribute "xmlns="urn:ios". Support for Juniper JUNOS can be added by just adding a "xmlns=urn:junos" part to the template.

**Step 3.** Drop this as a template into NCS.

Save the template XML file under the templates folder in a package `i3vpn` and reload the packages using the following command:

```
# packages reload
```