NetBrain® Integrated Edition 10.0

Feature Intent Template Tutorial

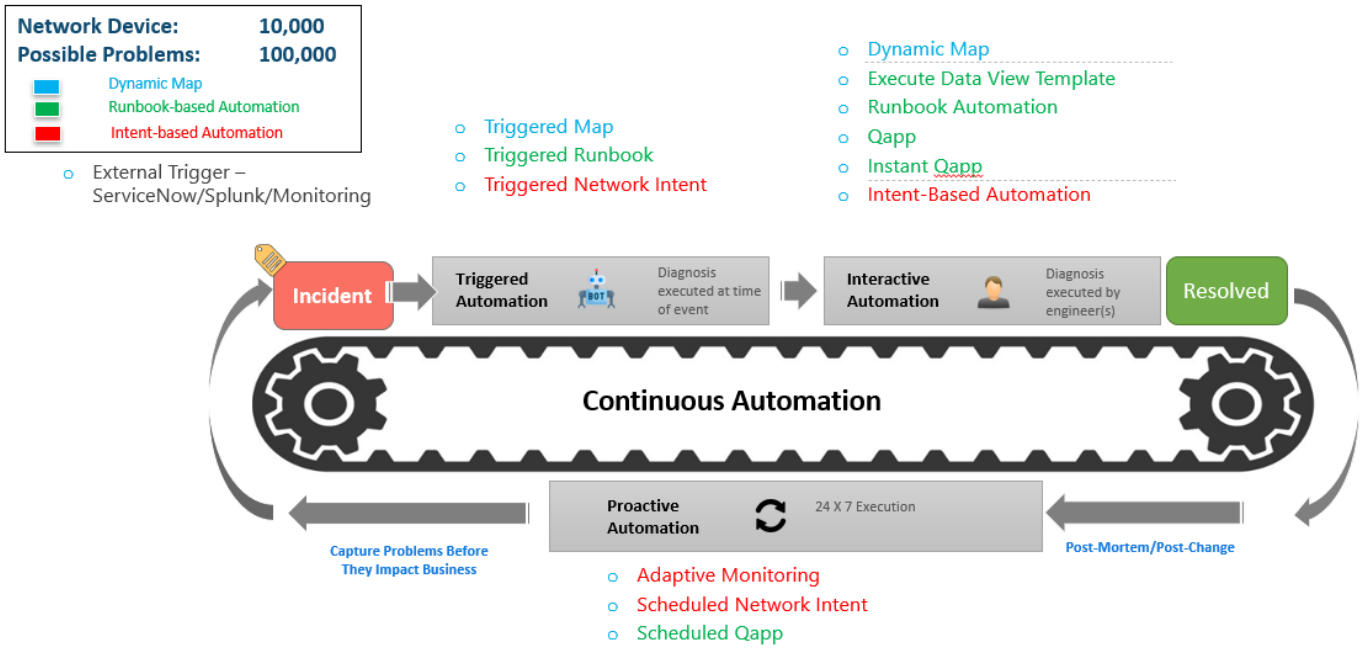
Version 10.0a | Last Updated 2021-07-19

Contents

1. Introduction

Network Troubleshooting, which requires enormous knowledge, individual efforts and team collaborations, could be demanding and time-consuming. NetBrain's reference workflow with the new feature, intent-based automation, aims to automate every incident ticket. The automation feature includes triggered automation and interactive automation. The key components are listed below:

1) Adaptive monitoring probes
2) Dynamic Map (Enhanced with DVT)
3) Triggered runbook execution
4) Triggered NI execution
5) Interactive Runbook execution
6) Interact NI execution
7) Guidebook execution (which consists DVT, RBT and NI)

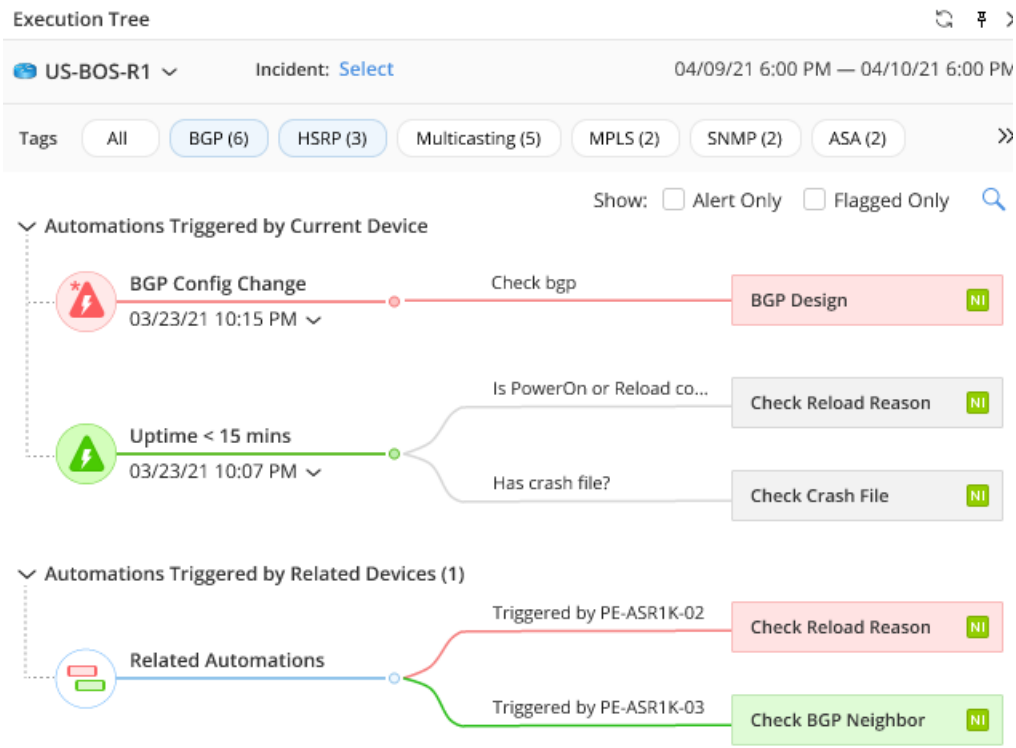


However, building the automation system is not a simple matter with the following challenges:

- Network Intent is device-based automation for end users. As it is designed purposefully, end users can easily utilize it. Also, end uses can define it with deep automation analysis logic, which can be applicable in any scenarios. But it is difficult to be applied to other devices with similar intents.

- A better way needs to be found for the entire intent-based automation (adaptive monitoring, decision tree) to scale to a large network and to keep up with the network change (intent change) in the meantime.

To overcome the challenges, the engineers need to build the intent-based automation device-by-device, and intent-by-intent, then view the results from the decision tree. It is quite difficult to build the intent-based automation for a large network with complex technologies.

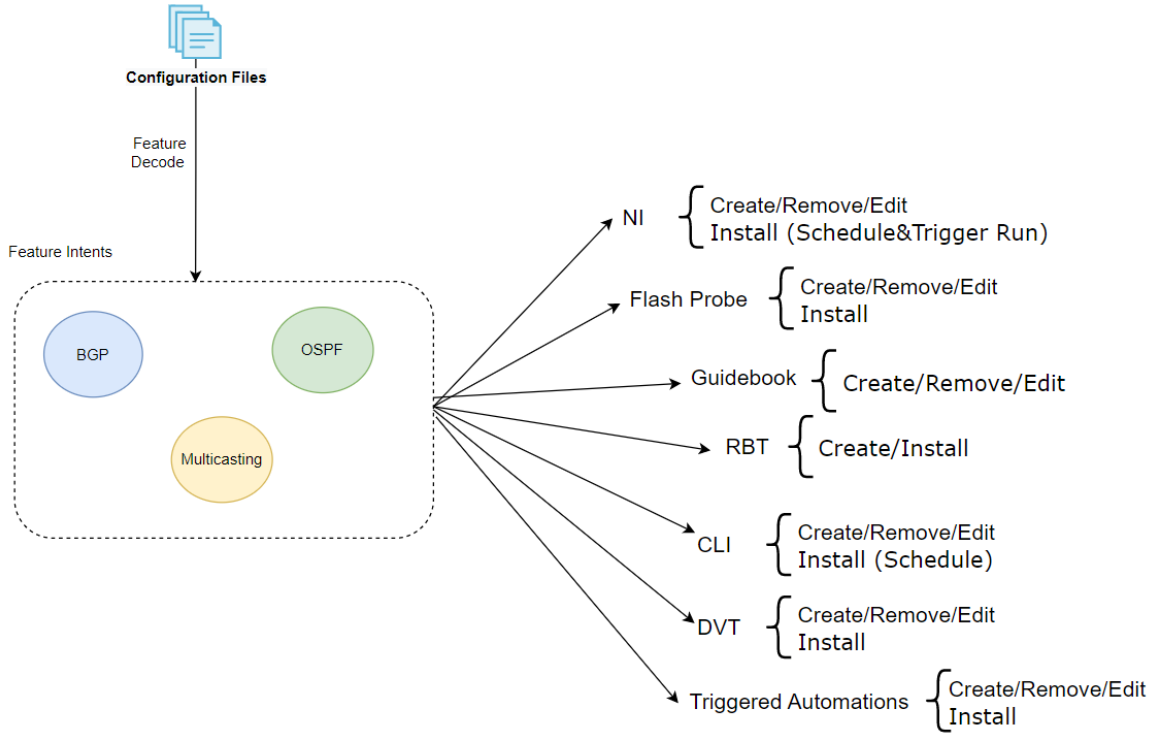


Feature Intent Template is aiming to resolve the problems mentioned above with the following key characteristics:

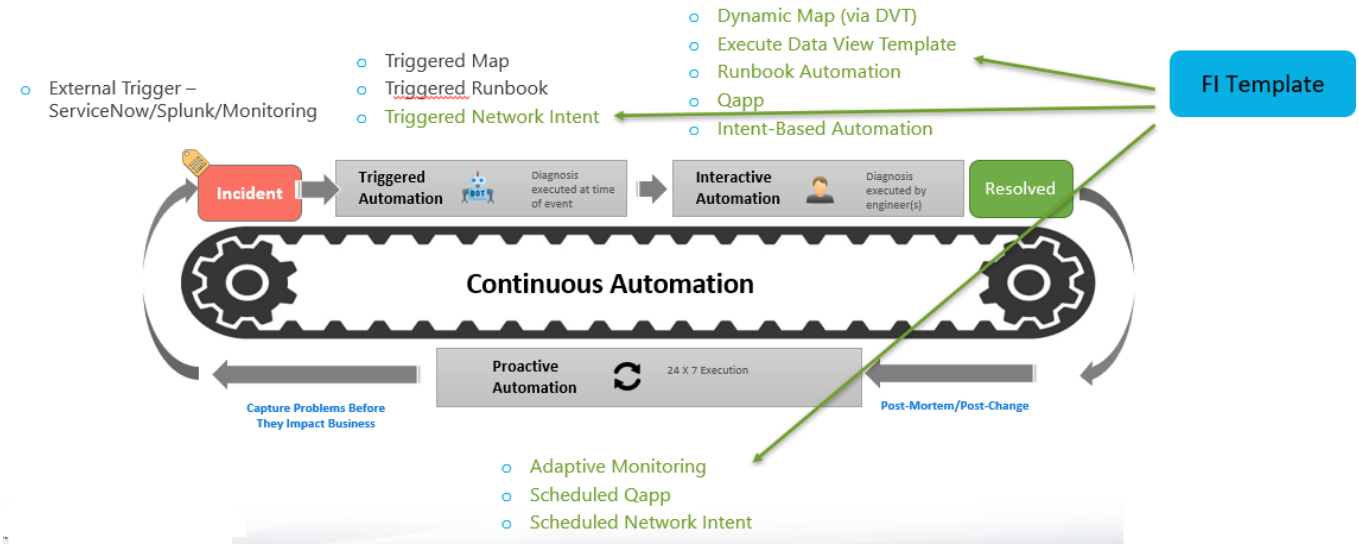
- Decode network features using line pattern for accurate device feature match.
- Scale intent-based automation to the entire network with the device matched.
- Maintaining the intent by executed periodically.

Feature Intent Template defined inside YAML-Format Feature Intent Definition File (FID file) is a set of automation technology to define NetBrain automation across the entire network.

With the Config line pattern, various network technologies can be decoded from device configuration files. Furthermore, the targeted device can be matched, and the key parameters can be stored in the line pattern for future use. This will help to identify across the entire network which devices are running certain network technologies (BGP, QOS, Multicasting etc.) and create the related automation resources in NetBrain's system (Network Intent for BGP design, Flash Probe for BGP flapping check etc.). Execution methods can be further defined as either triggered by the system or interactively executed by users.



The main purpose of the feature intent template is to decode network feature and build/install automations across the entire network to support the reference workflow.



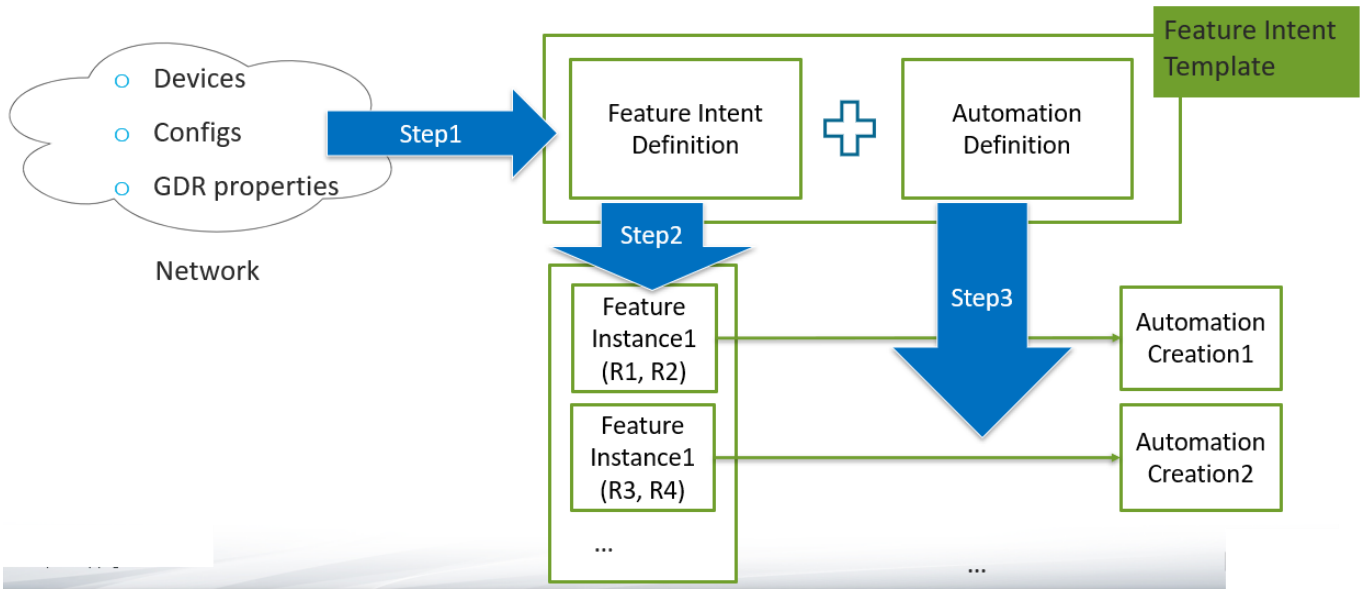
In this tutorial, we will first introduce the Feature Intent main flow. Then, move on to the feature decode concept and creating default networking intent. After that we'll show how the system can create network intent based on network intent template. And further we'll explore how the entire intent-based automation functions can be built via feature intent template.

2. FI Template Main Execution Flow

In this chapter, we'll briefly illustrate how feature intent template works in your environment. Feature Intent Template can be used in the following scenarios:

- NetBrain Service Engineer: NetBrain's service engineer can leverage the Feature Intent Template to create automation based on the needs and apply the feature intent template easily to other customers if needed.
- Customer DevOp Engineer: Customers' DevOp engineers can create Feature Intent Templates and apply the Feature Intent Template to the current network.

The following diagram illustrates how the Feature Intent Template works in your environment and support your network consisting of multiple network devices with their corresponding configuration files. NetBrain will further create data model and part of the data will be stored via GDR properties.



With all the information, you can define the Feature Intent Template which includes the two parts:

1) **Feature Intent Definition**: This is the main component that defines how device config should be matched with the defined line patterns. Devices along with their configs and GDRs will be evaluated by the feature intent definition.

```yaml
name: HSRP Essential - [Cisco IOS]
unique_id: 71bcb750-9cfc-4751-b25d-b2f8ed6be258
version: "1.1" # FID YAML structure version number, it should now be 1.1
description: "HSRP essential for cisco ios "
tags: [HSRP]
feature:
  qualification: {}
  configlet:
    sample: ""
    match_rules:
      - regexes: {}
        patterns:
          group1: |- #ss
            M: interface $str:intfName1
            M: ip address $ip:ip1 $ip:mask1
            M: standby $str:standbyName1 ip $ip:ip2
            M: standby $str:standbyName2 priority $str:priority
        split_keys:
          group1: []
        relation: Equals($fi.standbyName1, $fi.standbyName2)
        merge_groups: []
    eigen_variables:
      - name: crossRelationKey
        expression: Combine($fi.standbyName1, IP($fi.ip1, $fi.mask1).GetSubnet() )
      - name: hsrp_name
        expression: Str($fi.ip1).ReplaceSpecialChar() + "_" + $fi.standbyName1

    commands:
      - show standby
      - show standby brief
      - show ip interface brief
      - show logging | in %STANDBY
  cross_relation:
    group_by: $fi.crossRelationKey
```

2) **Automation Definition**: This part defines what automation assets need to be created/installed based on the matched feature intent including network intent/flash probe/triggered automation, etc.

A network including multiple devices will match devices with the network feature specified within the Feature Intent Template and a set of automation assets will be created accordingly.

Feature Intent Template builds the data model group up from the raw configuration files, and it is not intended to be created by end users. End users should only run the Feature Intent Templates, see the generated results and use the created automations.

3. Understanding Feature Decode using YAML File

Network Troubleshooting requires deep understanding of different network technologies (i.e. ,HSRP, QOS or BGP) configured on each device. Based on different network features, the knowledge and automation needed for further troubleshooting may vary. To automate the automation assets required for troubleshooting, the key and fundamental is to understand network features. This is how feature intent template decode network features.

By looking into the configuration files of devices, we use the line pattern concept to find the matched devices for that specific feature. One simple example is to look for whether BGP routing protocol is configured on Cisco IOS device by searching for the following config lines:

```
router bgp 2
  neighbor 10.10.10.10 remote-as 1
```

Each of the line contains two types of words, one is **network keyword** and the other one is **variable**. If we take the first line as an example, `router` and `bgp` are network keywords while `2` is a variable. As different router may configure different routing process, we need to combine the keyword with the variable to search for whether BGP is configured for specific devices. By combining keywords and variables into a single line, we have created a unique line pattern that serves as the feature decode unit. In NetBrain's implementation, the variable is represented by `$<variable type>:<variable name>`, so the line pattern that could be used to find the above configuration file lines are:

```
router bgp $num:bgp_as
  neighbor $ip:neighbor_ip remote-as $num:remote_as_number
```

Since the BGP number here is an integer, we'll need to define the variable type as number (integer or float), abbreviated as num. IP address is a built-in variable type in NetBrain, so you can use it to represent the IP address. Remote-as number can be defined as num as well.

Must-have Line and Optional Line

The configuration for a specific network technology varies and it's not always the same. To use the line pattern to find the maximum number of matches for a specific feature, you can leverage the must-have line and optional line concept to tag your line pattern. Let's take the following configuration file snippet as an example:

```
interface GigabitEthernet0/21

description HSRP-GROUP

no switchport

ip address 192.168.2.1 255.255.255.0 secondary

ip address 192.168.1.1 255.255.255.0

standby 1 ip 192.168.1.100

standby 1 priority 150
```

In order to find the device with the HSRP configured and match as many configlets as possible, we will need to define the following lines as must-have lines:

```
interface GigabitEthernet0/21

ip address 192.168.1.1 255.255.255.0

standby 1 ip 192.168.1.100
```

The above must-have lines are the key line patterns which determine whether the device has the HSRP configured. You may or may not have the priority field configured by standby group, therefore you can configure the following line as an optional line.

```
standby 1 priority 150
```

To specify whether a line is must-have or optional, you can use the `M` or `O` as flap ahead of the line patterns. Putting them together, you'll have the following line pattern to match devices.

```
M: interface $str:intf

M: ip address $ip:ip_address $ip:ip_mask

M: standby $num:standby_group ip $ip:standby_ip

O: standby $num:standby_group2 priority $num:standby_value
```

> **Note:** Only devices that include all the must-have lines sequentially will be deemed as a match, so using the optional line here can help you match devices with or without priority defined for standby group. To match the devices that have priority explicitly defined, you will need to make the last line as must-have line.

Since the default behavior of line property is a must-have line, you can leave the must-have lines untagged and the system will recognize them as must-have. The below pattern means the first three lines are must-have lines and the last one is an optional line.

```
   interface $str:intf

   ip address $ip:ip_address $ip:ip_mask

   standby $num:standby_group ip $ip:standby_ip

O: standby $num:standby_group2 priority $num:standby_value
```

The configuration must match the line pattern definition sequentially to be identified as a match, if any line of the configurations doesn't match the defined line pattern, it will not be considered as a match. The following modified configlet won't be considered as a match for the defined line pattern as the lines cannot match the exact order.

```
interface GigabitEthernet0/21

standby 1 ip 192.168.1.100
```

```
ip address 192.168.1.1 255.255.255.0
```

Organizing Line Pattern into Different Groups

With the exact line pattern match rule by order, sometimes you will need to find repetitive lines for certain line patterns to find all the matched config lines. To support grouping several lines into a unique matching unit, we introduced the group concept to better match device config files. The previous defined line pattern can be recognized as single group and we can give it a simple group name group1 to indicate its uniqueness:

```
Group1:

       M: interface $str:intf

       M: ip address $ip:ip_address $ip:ip_mask

       M: standby $num:standby_group ip $ip:standby_ip

       O: standby $num:standby_group2 priority $num:standby_value
```

By grouping these line patterns together, all interfaces with hsrp configured will be identified and extracted.

Another use case is to divide your line patterns into different groups so each group can be used as a unit to match separately. For example, you want to find OSPF configuration files for Cisco devices that contain interfaces with OSPF configured, the line pattern will look like below:

```
Group1:

       interface GigabitEthernet2/1

       ip address $ip:ip1 $ip:ip2

       ip ospf authentication-key 7 011208034E18

       ip ospf network point-to-point

       router ospf 1

       router-id $str:router_id

       passive-interface default

       no passive-interface GigabitEthernet2/1

       network 10.41.1.64 0.0.0.1 area 0

       network 10.41.2.0 0.0.0.255 area 0

       maximum-paths 2
```

As the previous rule stated, if you put all these lines into a single group, NetBrain will search through the configuration lines for a match. Therefore, a configuration file that may include multiple OSPF interfaces configured may only be matched once. In order to support this case, you can use the group logic to divide the line pattern into different OSPF groups as below:

```
Group1:
```

```
       interface GigabitEthernet2/1

       ip address $ip:ip1 $ip:ip2

       ip ospf authentication-key 7 011208034E18

       ip ospf network point-to-point

Group2:

       router ospf 1

       router-id $str:router_id

       passive-interface default

       no passive-interface GigabitEthernet2/1

       network 10.41.1.64 0.0.0.1 area 0

       network 10.41.2.0 0.0.0.255 area 0

       maximum-paths 2
```

By dividing the line patterns into two different groups, NetBrain will search for the exact match for each group separately, so a configuration file that includes multiple interfaces can quickly match the group1 definition and the global OSPF configuration.

> **Note:** The sequence of the groups doesn't matter, so if the defined pattern starting from group1 then group2, while the real configuration file starts with group2 and then group1, the device will still be recognized as a match.

3.1. Test Feature Decode Result using Feature Intent Template

With the basic line pattern match rules, as explained previously, we can test the match result for certain devices. From the NetBrain end user page, opening the feature intent template, you can easily create a new feature intent template. NetBrain has populated a lot of sections in the default feature intent template but you don't need to worry about them at this stage.

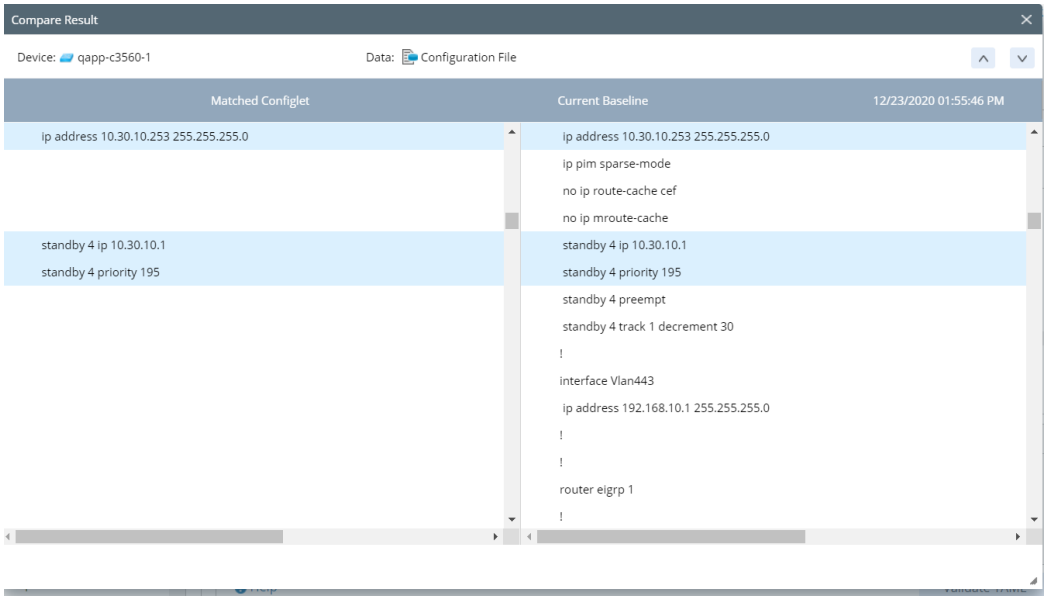
YAML Reference Network Intents

```yaml
name: General_HSRP - Step1
version: 1.0
source: ""
description: ""
tags:
  - HSRP
feature:
  qualification: {}
  configlet:
    sample: ""
    match_rules:
      - regexes: {}
        patterns:
          group1: |- #ss
            M: interface $str:intfName1
            M: ip address $ip:ip1 $ip:mask1
            M: standby $str:standbyName1 ip $ip:ip2
            M: standby $str:standbyName2 priority $str:priority
```

We can use the default settings and only modify the `match_rules` section to add the line patterns as explained, then group the line patterns into different groups if needed.

After adding the line patterns, click **Validate YAML** to verify whether the YAML format and content are correct. Fix any possible errors based on the error messages shown below until you see no errors.

Then you can select the Test Device Config Match to test whether a device is recognized as a match. If yes, you can see what configuration file lines are matched. Click  at the top right corner to navigate to previous/next matched configuration lines.



This function can help you quickly determine whether the defined line pattern matches with specific devices, as a result you can modify the line pattern effectively based on the results.

> **Note:** The configuration file is retrieved from the current baseline.

3.2. Line Pattern Match Logic

While there are a lot of detailed matching rules, we will not cover all of them in this tutorial. We will focus on the important logic.

Repetitive Matching

You will often see repetitive lines within a real configuration files, such as ACL configuration for Cisco devices. As it's impossible to define the exact line patterns for the repetitive contents, we'll need to use one single line pattern to match all repetitive contents. Let's take the following configlet as an example:

```
access-list 95 permit 135.89.176.120
access-list 95 permit 135.89.176.121
access-list 95 permit 135.89.176.79
access-list 95 permit 135.89.176.84
access-list 95 permit 135.89.176.82
```

In the above example, as you have one ACL configured with multiple statements, in order to match all these access-list, instead of stating 5 repetitive line patterns, you could simply use the following line pattern to match all the contents listed:

```
access-list $num:acl_id permit $ip:ip_addr
```

> **Note:** The keyword used to specify the repetitive matching behavior is "R". It is the default value and can be omitted. Use the keyword "S" to match the first line only.
>
> **Note:** Repetitive matching only works for non-first line of the line pattern. As if the line pattern appears in the first line of the line pattern, it will be used as the key to set the matched lines into different groups instead of all repetitive lines within a single group.

Fuzzy Matching

There are two types of line pattern matching method: exact match and fuzzy match. Exact match is relatively straightforward, meaning all the defined keywords and variables within a line pattern need to be precisely matched. Fuzzy matching means the line pattern only requires the first part of a real config line to be matched.

> **Note:** NetBrain's default line pattern behavior is fuzzy matching.

With the fuzzy matching, you can use a universal line pattern to match two or more different configuration lines. Let's use a simple example below to explain the concept:

We have the line pattern ip address `$ip:ip1 $ip:ip2` attempting to match the following config line:

```
ip address 10.10.10.10 255.255.255.0 secondary
```

Since the default behavior is fuzzy matching, this will also be recognized as a match. In case you want the line pattern to be precisely match and doesn't want to use fuzzy matching, you can modify the matching rule to exact matching so this line will no longer be recognized as a match. To mark a line pattern as exact matching, you can mark the line pattern with "`E`" flag in the beginning of the line:

```
E: ip address $ip:ip1 $ip:ip2
```

If the line needs to be marked both optional line and exact matching, mark the optional line and then the exact matching, an example is shown as below:

```
OE: ip address $ip:ip1 $ip:ip2
```

> **Note:** First line of each group does not support fuzzy matching by default. If the neither F flag nor E flag is specified, exact match will be performed by default.
>
> **Note:** F flag is required to specified in order to support fuzzy match.
>
> **Note:** If both F/E flags are written at the same time, they will be treated as F and fuzzy match will be performed.

3.3. Device Qualification Basics

Device feature decoding through configuration files is a powerful tool to quickly decode the network features from the network devices. However, it requires massive calculation across all the devices in the calculation scope. In some cases, you may need a light-weight method to quickly find devices. The following example

illustrates how you can leverage the device properties that have already been visualized by NetBrain, or use the regex as a qualification to achieve this goal.

```yaml
feature:
  # qualification: All
  qualification: # support All
    device_scope: # the default is null, null means all
      type: DeviceGroup # Site or DeviceGroup
      groups: ["Shared Device Groups/Group1", "Shared Device Groups/folder1"]
    dynamic_search:
      conditions:
        - property: mainTypeName
          values: [Router, L3 Switch, LAN Switch]
        - property: subTypeName
          values: [Cisco IOS Switch]
        - property: hasEIGRPConfig
          # support types:
          #   string: Matches(default), Does not match, Contains, Does not contain
          #   bool: Is/is(default)
          #   number: Matches(default), Does not match
          #   array: Matches/Matches any(default), Does not match/Does not match any
          operator: Matches
          values: false
        - property: name
          operator: Contains
          values: ["12", "3550"]
      boolean_expression: A and B and C and D
```

Qualification section allows you to use all device GDR properties to filter the related devices. For a full list of supported properties, refer to Qualification Properties and Condition.

`regexs` section allows you to define one or more conditions to match related devices. `mregex` is supported in this section.

Using the qualification and regex rule as preliminary filters can significantly improve the accuracy and performance.

In some use cases you may only need to define the qualification and regex match without using the config line pattern for feature decoding, which is totally fine, but you'll need to make sure you have at least one of the three matching methods defined in order for the system to match devices and execute properly.

4. Understanding Different Components of FIT

In the previous chapter, we explained the feature decode basics and how you can use the line patterns to match the configlet from configuration files. In this chapter, we will explain how you can divide the feature intent into sub-feature intent (SubFI for short) then generate default network intent with the sub-feature intent.

4.1. Sub Feature Intent Introduction

Feature Intent stands for all the configuration lines matched for line patterns. However, most of the times you could match a lot of repetitive patterns and you do want to divide the Feature Intent into sub Feature Intent for further network intent creation. Let's take a simple example for the line patterns created for HSRP feature:

```
        patterns:

          group1: |-

            M: interface $str:intfName1

            M: ip address $ip:ip1 $ip:mask1

            M: standby $str:standbyName1 ip $ip:ip2

            O: standby $str:standbyName2 priority $str:priority
```

The above pattern is created for HSRP feature match to match devices that have HSRP configured on its interfaces, but one interface may have multiple HSRP groups configured, each with its own ip address and priority. The following example shows a real configuration files with two HSRP groups configured on a single interface and we need to split the groups into two different network intents.

```
interface GigabitEthernet0/21

 description HSRP-GROUP

 no switchport

 ip address 192.168.2.1 255.255.255.0 secondary

 ip address 192.168.1.1 255.255.255.0

 udld port aggressive

 standby 1 ip 192.168.1.100

 standby 1 priority 150

 standby 1 preempt

 standby 2 ip 192.168.2.100

 standby 2 preempt
```

```
!
```

In order to divide different HSRP group into different sub Feature Intent and further create network intent based on certain HSRP group, we can divide the feature intent into SubFIs based on the following parameter in YAML:

- **Split_keys**: A line pattern could match multiple config line instances in the configuration file, and thus some line pattern variables may have multiple possible values. Defining variable here will make sure that variable only has one instance value in the subFI.

In the above sample, since we want to split the feature intent by group names, we can specify the `split_keys` as follows:

```
        split_keys: # variable signature concept, optional

         group1: [$intfName1, $standbyName1]
```

By defining the `spit_keys`, assuming we only have this interface with the HSRP configured, the subFIs we get from the end results is:

SubFI	Content
First SubFI	interface GigabitEthernet0/21 ip address 192.168.2.1 255.255.255.0 secondary ip address 192.168.1.1 255.255.255.0 standby 1 ip 192.168.1.100 standby 1 priority 150 standby 1 preempt
Second SubFI	interface GigabitEthernet0/21 ip address 192.168.2.1 255.255.255.0 secondary ip address 192.168.1.1 255.255.255.0 standby 2 ip 192.168.2.100 standby 2 preempt

> **Note:** The line patterns for interface configuration will be shown as the same in each for the subFIs.

```
interface GigabitEthernet0/21

 ip address 192.168.2.1 255.255.255.0 secondary

 ip address 192.168.1.1 255.255.255.0
```

> **Note:** If you don't define the `$standbyName1` as `split_keys`, you will only have 1 Feature Intent/SubFI and the content is shown as below:

SubFI	Content

First SubFI	interface GigabitEthernet0/21 ip address 192.168.2.1 255.255.255.0 secondary ip address 192.168.1.1 255.255.255.0 standby 1 ip 192.168.1.100 standby 1 priority 150 standby 1 preempt standby 2 ip 192.168.2.100 standby 2 preempt

> **Note:** Standby group 2 is also matched here as a result of group stacking feature.

Use Relation to group line patterns into SubFI

The previous example only contains one group in the pattern field, in case you have multiple groups in the pattern, and you want to group them together, you will need to have the relation defined.

Relation: relation is used to filter and keep the SubFI matching the relation definition, the only function you can use is `Equals($var1, $var2)` which means they should be the same.

```
relation: Equals($fi.intfName1, $fi.intfName2) && Equals($fi.ip2, $fi.ip3)
```

In case you may have multiple groups defined within the pattern:

```
patterns:

  group1:

  group2:
```

We'll skip the line patterns here to illustrate the concept, both `group1` or `group2` could find multiple matches within a configuration files. We'll mark the match as `group1.1`, `group1.2` … `group1.m` for `group1`, `group2.1`, `group2.2`… `group2.n` for `group2`.

For the subFIs to be generated, by default there would be m*n subFIs by default if there's no rule or relation defined. By defining the relation using `Equals($group1var, $group2var)`, we will only keep the SubFIs that match our definition.

The output value type of relation expression must be boolean.

> **Note:** Besides the function we explained here, you can also define your relationship by expression. The sample below is how you can define the same interface name as the relationship to achieve the same function.

```
$fi.intfName1 == $fi.intfName2
```

Setting generating SubFI Flag

By default if you use multiple groups or define the `split_keys`, NetBrain will generate multiple SubFIs according to your definition. But in some cases even if you find all related configlets, you still want to generate a single Feature Intent instead of multiple subFIs, in this case you can use the `merge_groups` flag:

```
merge_groups: ["groupX"]
```

You can define whether you want to create one instance for a single group for multiple groups. If you want all groups to be generated as a single Feature Intent, list all group names here so the system will only generate one single FI.

4.2. Generate FI Group for Multiple Devices

Once we have generated the FI and SubFIs for multiple devices, we'll need to group them together to generate FI group. FI group contains a couple of devices with network relationship, a few samples are provided as below:

- HSRP pair which includes active device and the standby device.
- ASA cluster which includes two ASA devices.

FI group can be recognized automatically to divide devices into different clusters based on the network feature, and it is the equivalence of network intent.

To generate FI group across multiple devices, we must find unique characteristics for these devices. From a networking perspective, it can be easily explained by:

- HSRP pair of devices share the same virtual IP address and the primary device/secondary devices are within the same subnet.
- Devices within an ASA cluster share the same IP address.

The unique characteristics of each device to generate FI group is denoted with the "`eigen`" variables, identified with the following statements:

```
eigen_variables:

  - name: crossRelationKey

    expression: expression: Combine($standbyName1, Str(IP($fi.ip1, $fi.mask1)))

  - name: site

    expression: $device.GetSiteName()
```

There are different ways to define the eigen variable expression:

1. SubFI variable: Use the SubFI variable directly if you want to use the SubFI variable extracted for eigen variable.
2. Function calls: In case you want to group several SubFI variables, you can use the function call to merge several SubFI variables into a new variable. In the above case, we used the `combine()` function which is designed to combine several variables into a new one. As we want to make sure the FI group we are dealing with should have the same IP address as the same HSRP group. We don't want to mix different HSRP groups into a single FI group.
3. NetBrain's GDR properties: A lot of NetBrain's buit-in properties can be used in eigen variable for verification purposes. In this case, if you have different sites that may have the same HSRP ip address or HSRP group, you'll probably want to differentiate these devices by further criteria. Here we use the

`device.site` property to ensure that the same physical container site does not have same HSRP ip addresses.

You can use one or more eigen variables for device clustering. One of key eigen variables can be for cross device grouping and the others for complementary verification. The eigen variable for cross device grouping is denoted with the `cross_relation` section:

```
cross_relation:

group_by: $crossRelationKey

fi_qualification: $site!=null

group_type: ExactMatch
```

By using the `cross relation` key, we are grouping the SubFIs into FI group. The system will use the `group_by` field to identify SubFIs that have the same eigen variables and group them together.

The `fi_qualification` field can be used to further filter unwanted SubFIs based on eigen variables. In this case, we only want to generate FI group, if the devices are within the site that we created. And we don't want to generate FI group for devices that we haven't allocated to certain sites as that may introduce inaccuracy. We can use the `$site` as the qualification to filter devices that don't belong to any site.

The last one `group_type` define the method to group devices into the same FI group, and there are three types:

1. Match: Eigen variable in this field needs to be the same, please note in case eigen variable may contain array includes multiple values, and it only requires single value to be matched between different devices to be recognized as a match.
2. ExactMatch: This is the default type and the eigen variable in this field needs to be precisely the same, so SubFIs can be grouped into a single FI group. This includes the situation where you use the `combine()` function to combine several eigen variables together, all eigen variables value need to be the same to be grouped into a single FI group. In case eigen variable may contain array that includes multiple values, all items within the eigen variable must be matched so different devices will be further grouped into a single FI group.
3. Contain: there are cases that you may want to group subFIs together if the eigen variable contain the same eigen variable value, and they don't need to be the same. A simple use case is one P device has several BPG neighbors to the PE devices and you want to group them into a single FI group. The eigen variable is defined as the bgp neighbor IP address, while the bgp neighbor IP address is defined in a list which contains all eigen variable value of all PE devices.

4.3. Resource Generation Logic

From the last two sections, we have explained the logic for Sub FI and FI Group, which belong to single device and multiple devices respectively. Based on the feature decode results, NetBrain will further generate automation resources and each automation resource has its own rule based on different feature decode results.

Let's use the following results as an example: suppose we have one single Feature Intent Template to run against 2 devices with HSRP configurations. Based on the feature decode definition, we will be able to get the following components with sample data:

Terminology	Explanation	Sample1(HSRP Case)
Line Pattern	Used to express certain network features and further based on this feature to match configlet in devices.	M: interface $str:intfName1 M: ip address $ip:ip1 $ip:mask1 M: standby $str:standbyName1 ip $ip:ip2 M: standby $str:standbyName2 priority $str:priority
FI (Feature Intent)	Stands for all configlets matched from configuration files based on line patterns for a single device. May include one or more SubFIs	Device: US-BOS-SW1 interface Vlan100 ip address 10.8.1.2 255.255.255.240 standby 1 ip 10.8.1.1 standby 1 priority 90 standby 2 ip 10.8.1.8 standby 2 priority 105 interface Vlan101 ip address 10.8.1.18 255.255.255.240 standby 1 ip 10.8.1.17 standby 1 priority 105
SubFI (Sub Feature Intent)	One instance of repetitive patterns extracted from feature intent. Used as an individual automation analysis unit. Created with grouping and split-keys.**Use Case**: Need automation creation/analysis logic based on different configlets of FI. For example, a single device configured with different HSRP groups may assume different roles (active/standby) for each of its group, and you want to create different automation logic based on its active/standby status.	Device: US-BOS-SW1SubFI1 interface Vlan100 ip address 10.8.1.2 255.255.255.240 standby 1 ip 10.8.1.1 standby 1 priority 90 SubFI2 interface Vlan100 ip address 10.8.1.2 255.255.255.240 standby 2 ip 10.8.1.8 standby 2 priority 105 SubFI3 interface Vlan101 ip address 10.8.1.18 255.255.255.240 standby 1 ip 10.8.1.17 standby 1 priority 105
FIG (Sub Feature Intent Group)	A cluster of devices with each of their SubFI based on their **common network characteristics** (represented by Eigen Variables)Samples: • HSRP pair of devices they share the **same virtual IP address**, and the primary device/secondary devices are within the **same subnet**.	FIG1:Device: US-BOS-SW1 interface Vlan100 ip address 10.8.1.2 255.255.255.240 standby 1 ip 10.8.1.1 standby 1 priority 90 Device: US-BOS-SW2 interface Vlan100 ip address 10.8.1.3 255.255.255.240 standby 1 ip 10.8.1.1

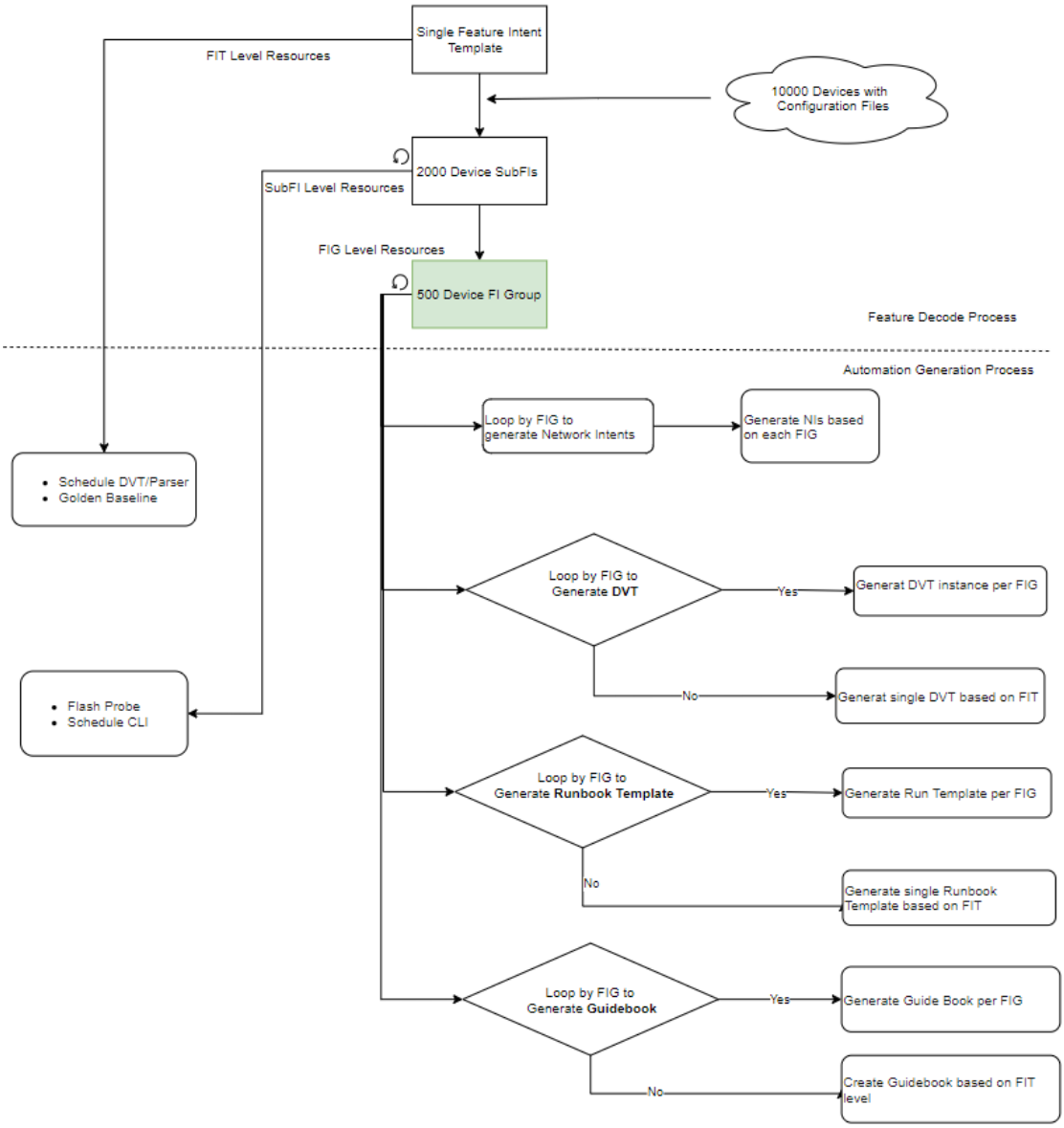
	• Devices within an ASA cluster share the **same cluster IP address**. • Devices with **same BGP as number** clustered into a unique FI Group.	standby 1 priority 105 FIG2: Device: US-BOS-SW1 interface Vlan100 ip address 10.8.1.2 255.255.255.240 standby 2 ip 10.8.1.8 standby 2 priority 105 Device: US-BOS-SW2 interface Vlan100 ip address 10.8.1.3 255.255.255.240 standby 2 ip 10.8.1.8 standby 2 priority 90 FIG3: Device: US-BOS-SW1 interface Vlan101 ip address 10.8.1.18 255.255.255.240 standby 1 ip 10.8.1.17 standby 1 priority 105 Device: US-BOS-SW2 interface Vlan101 ip address 10.8.1.19 255.255.255.240 standby 1 ip 10.8.1.17 standby 1 priority 110
Eigen Variable	Used for FIG creation logic. Defined via subFI variables. One Eigen variable (cross-relation key) is used to group subFIs while other Eigen variables can be used for further analysis.	eigen_variables: - name: crossRelationKey expression: Combine($standbyName1, IP($ip1, $mask1).GetSubnet())
Cross-relation Key	Used to group subFIs into FI group. subFIs with the exact value of cross-relation key will be grouped into a unique FI group.	cross_relation: group_by: $crossRelationKey

The automation will be created based on the following components:

- FIT (Feature Intent Template): this means the resource are generated based on the feature intent template. A single feature intent template may generate multiple FIs and FIGs for multiple devices, but some resources are generated only based on the single Feature Intent Template being used, no matter how many different subFIs or FIGs have been generated.
  The followings are a few sample resources that can be generated based on FIT:
  - Schedule DVT/Parser Tasks
  - Golden Baseline Definition
  - DVT/Runbook Template configured based on FIT level.
  - Guidebook configured based on FIT level.
- SubFI: this means the resources are generated based on the sub Feature Intent calculated for each device, when a single device has multiple SubFIs, the following resources can be generated based on each SubFI that has been created:
  - Flash Probe
  - Schedule CLI Command

- FIG: this means the resources are generated based on the calculated FI group. When a FIT runs against devices and generate multiple FIGs, the following resources can be further generated based on each FIG that has been created:
    - Network Intent
    - Guidebook configured based on FIG level
    - DVT/Runbook Template configured based on FIG level

We will introduce examples on how these resources are generated based on different levels. The following diagram gives you a high-level view of how different automations can be generated based on different resources:



4.4. Reference Variables of Different Levels

As discussed, you can use line patterns to decode features for devices and reference variables in different automation assets. Now it's time to take a look at different types of variables you can use.

We know that you can define variables within line patterns and these variables are called line pattern variables. You can use them by including the FI as prefix.

The following sample shows line patterns used to match interfaces with HSRP configuration with variables defined:

```
group1: |- #ss

            M: interface $str:intfName1

            M: ip address $ip:ip1 $ip:mask1

            M: standby $str:standbyName1 ip $ip:ip2

            M: standby $str:standbyName2 priority $str:priority
```

In order to use CLI commands that include the interface names, we can use the following statement to reference the interfaces that are matched:

```
                  - show standby interface {$fi.intfName1}
```

The same format also applies to eigen variables.

The following sample shows how you can define an eigen variable:

```
    eigen_variables:

      - name: crossRelationKey

        expression: Combine($fi.standbyName1, IP($fi.ip1, $fi.mask1).GetSubnet() )
```

In order to use these variables, you will need to use `$fi` as prefix to reference these variables.

```
                cross_relation:   group_by: $fi.crossRelationKey
```

Both line pattern variables and eigen variables can be recognized as SubFI level variables and you will need to reference these variables by using `$fi` as prefix. All the supported variables are listed as below:

Variable Level	Variable Source	Reference Prefix	Sample
SubFI	• Line Pattern Variable • Eigen Variable	`$fi`	`$fi.bgp`
Device	Device GDR variable	`$device`	`$device.name`
Interface	Interface GDR variable	`$intf`	`$intf.name`
FIG	Built-in variables in FIG	`$fig`	`$fig.crossRelationHash` `$fig.deviceNamesStr`

NI	NI role variables defined in NI	`$ni`	`$ni.roleVariable2`

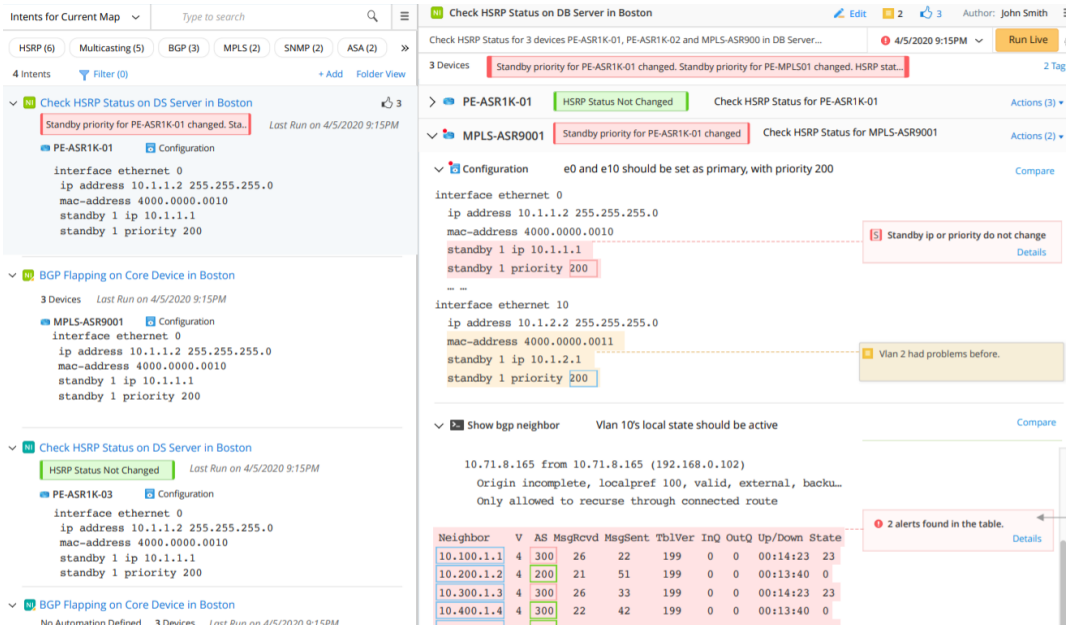
We will cover samples of how to use them in the next few chapters.

5. Create Network Intent

Network Intent is device-based automation, allowing you to define the desired status for a couple of devices (usually devices that have direct relationships from a network perspective)from configuration files and CLI commands, and also define the alert logic to check if the desired status is violated. For more network intent details, please refer to the document: Network Intent Tutorials.

Network intent may include very complex automation logic of defining how to check the desired status. It may only include the basic configlets and CLI commands without automation logic, which is also known as the default Network Intent.

The configuration files decoded can be used to fulfill the configlet displayed in the network pane, as well as the configlet stored in network intent detail pane if there's no automation logic defined for the network intent.



We can also define the show command within the YAML file to generate CLI command used in Network Intent.

Defining CLI Command in YAML File

You can also define the show commands for feature verification. And the CLI command will be passed on to the network intent CLI command when the default network intent is created based on the feature intent. To define the CLI command section, simply put the CLI commands in it. Then, the CLI commands will be used for the HSRP check, as shown below:

```
match_rules:

    patterns:

      group1: |- #ss

        M: interface $str:intfName1

        M: ip address $ip:ip1 $ip:mask1

        M: standby $str:standbyName1 ip $ip:ip2
```

```
            M: standby $str:standbyName2 priority $str:priority

   commands:

      - show standby

      - show standby brief

      - show standby interface {$fi.intfName1}

      - show standby interface {$fi.intfName1} {$fi.standbyName1}
```

You can see that not only the general CLI commands without parameters can be used here. You can also use the CLI commands with the parameters referenced from the line patterns. In the above example, we use the show standby interface `{$intName1}` command.This actually means from the configuration files, we use the line pattern to match the interfaces with the HSRP configuration, then we only check the interface HSRP status for these interfaces. By specifying parameters used in line pattern, we can significantly improve the CLI command accuracy for future use.

Before getting into the details of using feature intent template to create network intent with automation logic, we expect you have the basic understanding of network intent components and its various automation logic basics; please refer to Network Intent Tutorial if you need to get familiar with these concepts.

The concept of using Feature Intent Template to create much more network intents based on the existing network intent is simple. As network intent is device-based automation, you will need to select specific devices and then define the network intent automation. So, if you have a lot of devices with the similar network technology and need to define the similar automation logic, it is tedious to manually replicate the logic to all other devices. With feature intent template, devices can be found automatically, and the automation analysis logic can be applied to the new network intents.

In this chapter, we will first demonstrate how to define the network intent template parameters within existing network intent, and then move on to how you can define these parameters in feature intent template. Later we'll show you how to run the feature intent template and view the results.

After having created the SubFIs and the FI groups based on eigen variables, we can convert the FI groups into Network Intents. There are three ways to convert FI group into network intents:

1. **Generating default network intent**: In this step, we directly convert the configlets and CLI commands to network intent. As there's no automation logic is defined further, you will have network intents that only include the configlets and CLI commands.
2. **Generating network intent with analysis logic directly**: In this step, we will use the devices and configlets generated for each FIG and define the analysis logic to generate Network Intents directly.
3. **Generating network intent with NI template**: In this step, we convert the FI groups into network intents based on the NI template. We can leverage the automation logic defined in the existing NI template to create new NIs with the similar logic. In this section, we will mainly cover the steps to generate default network intent. And we will go through more details regarding generating network intents with NI template in the next chapter.

In order to generate default network intent, we'll need to define the related network intent contents:

```
network_intents:

  path: xxxx/xxxx/General_BGP_{$crossRelationHash}

  conflict_mode: Skip
```

```
lock_after_created: false

create_default_NI: true

baseline_update_type: LatestFromDE
```

The path field shows where to input the newly generated default network intents.

> **Note:** We need to make each network intent unique so we can attach the CrossRelation field to the network intent name.

The `conflict_mode` section is to determine if the network intent with the same name already exists even it was not modified by this feature intent templet. In this case you can either overwrite the existing network intent by using override flap or skip flag to simply skip it.

The `lock_after_created` field can be configured to lock network intent and prevent anyone else from modifying it.

> **Note:** Setting this field to be true needs to occur after the creation of network intent for the first time.

The feature intent template will not be able to update the network intent when it's locked, and it has higher priority over the `conflict_mode` field. In this case, you won't be able to update the network intent and you have to modify the network intent manually.

The `create_default_NI` field, as explained previously, is to specify which type of network intent aims to generate. As in this case we are trying to generate default network intent without automation logic, we'll set this field to be true.

`baseline_update_type`: This field specifies how you would like to set the CLI baseline data, as the network intents include the CLI commands defined in the YAML file. Since executing the feature intent template won't retrieve the CLI commands from the live network, two methods are provided as below to add CLI command output to the network intent:

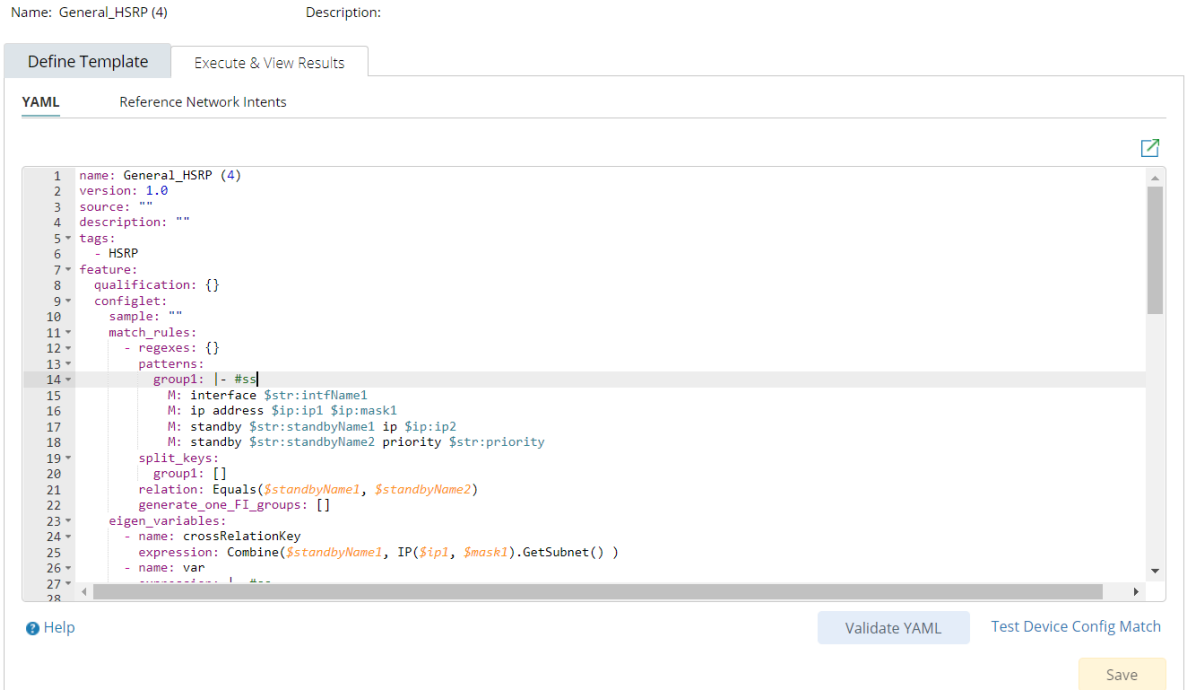
1. `LatestFromDE`: Use this setting if you want to make the current CLI command output as the baseline of network intent. You need to make sure you have already retrieved the CLI command previously before executing the feature intent template.
2. `LiveFromNextRun`: Use this setting if you want the system to set the CLI command baseline from the live network when running the network intent.

As we don't need to use the network intent template for reference, it is not necessary to define the `ni_template` and `ni_inputs` sections.

5.1. Generate Default Network Intent

In the previous chapters, we have explained all the necessary contents needed to generate default network intent; In this section let's put them together and generate our first default network intent based on the Cisco HSRP feature.

We can create a new feature intent template and reuse the related contents as explained earlier.



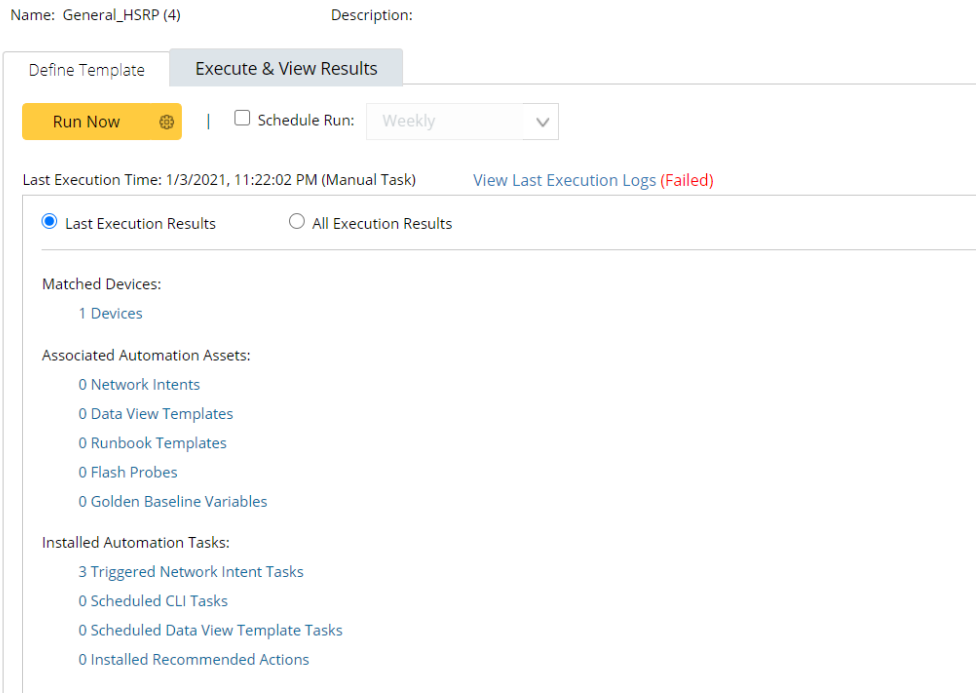
There is some basic information to be modified based on our feature intent template function.

- **Name:** the name of this YAML file; you can modify this filed and the name on the folder tree will be modified accordingly.
- **Description:** define the description for this YAML file.
- **Tags:** specify the network technology in the YAML file. The settings here will also be inherited once we have the network intents creating from this feature intent template.

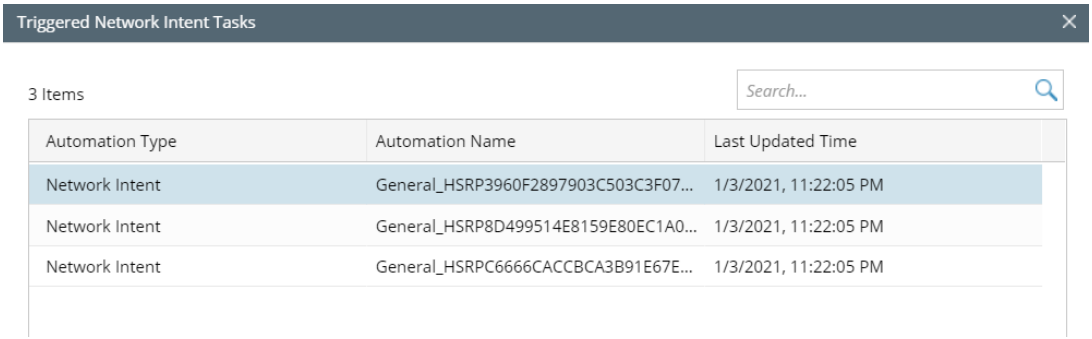
Use the Validate YAML function to test if there's any YAML syntax error or content error. You can also use the Test Device Config Match to view the matched feature intent. The device match function here only works on a single device for feature intent. It is not intended to view the subFIs or FI groups in current design.

Once the YAML file is properly defined and the testing results match our expectations, we can execute this feature intent template across several devices.

Click and navigate to **Execute & View Results** tab; click **Run Now** and select the devices you want to run against, then you will be able to view the results of current execution momentarily.

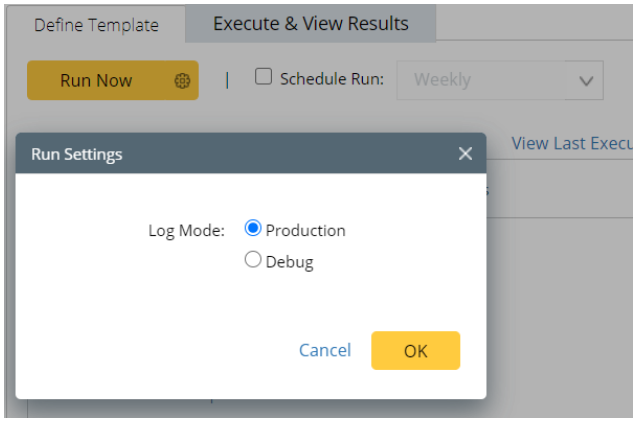


You will be able to view all the automation resources generated by this feature intent template. Click each item to view the details for each type of resources. You will find the default Network Intents created by this feature intent template.

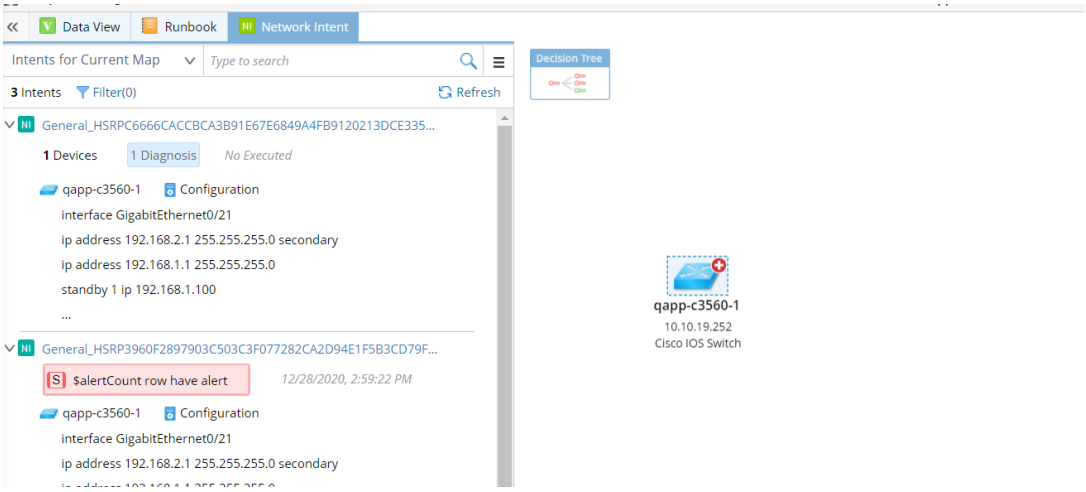


You can also click the **View last Execution Logs** to find the latest execution logs and view the execution details.

> **Note:** If you want to debug the execution of feature intent template, click the **Run Now** settings and switch the **Production Mode** to **Debug mode** in the **Run Setting**.



To view the network intents details, you can create a map to include the devices within the network intent,; and then from the left network intent pane, you will find all the default network intents the system just created based on our definition.



For more information regarding using and viewing network intent, please refer to the document Network Intent Tutorial.

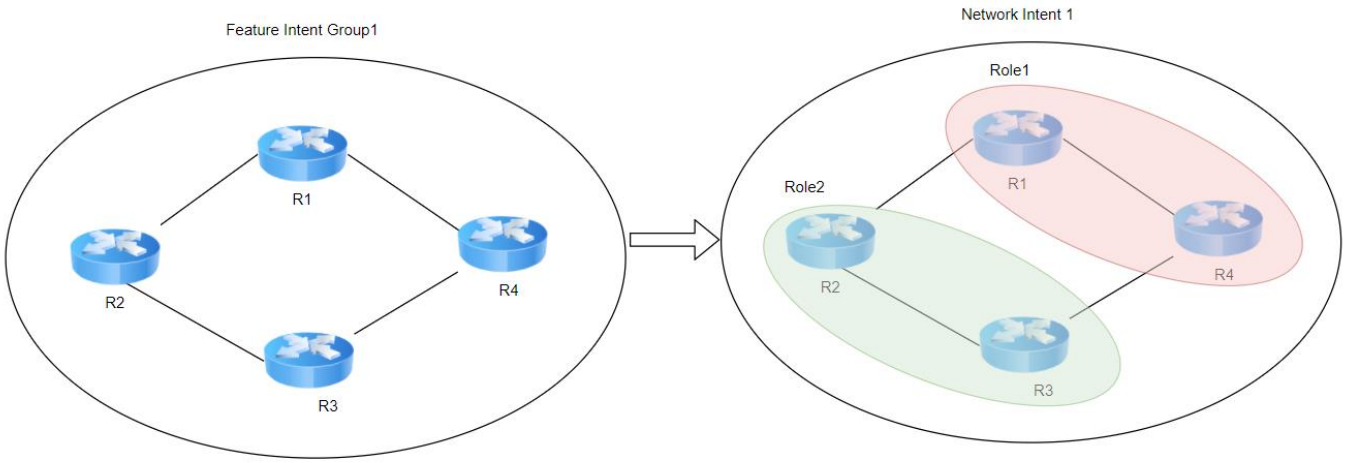
5.2. Create Network Intent Directly

Previously we have explained how you can create the default Network Intent without automation, as well as how you can create Network Intent with the full automation function. To understand how you can create Network Intent with the full automation function, first you will need to understand how devices within the FIG are matched and then mapped to different devices of NI with analysis logic.

The following diagram shows a single FIG with a couple of devices (R1, R2, R3 and R4). To create the corresponding network intent based on the FIG, we need to think about the analysis logic for devices within the FIG and group devices with the same analysis logic into a single role.

The NI role shown below represents devices that share the same analysis logic within the network intent. In the YAML file we can define the analysis logic for a single role and define the conditions for the role to match

the corresponding devices. When the system executes the YAML file, it will find the matched devices and generate the analysis logic for these devices.



Let's take a look at how YAML file looks like in this case:

The sample YAML code shown as below only serves for illustration purpose and may not include all the sections.

```
network_intents: # Definition for generating NI.

  tags: [BGP, HSRP]

  lock_after_created: true # the default is false

  baseline_update_type: LatestFromDE #

  create_default_NI: false # Whether to create Default NI for the FIG, the default is false.

  common_role_variables:

  analysis_devices:

    - device_role: role1

      device_condition: expression($fi.xxx)=="yyyy1"

      commands:

        - type: Command # Command or Config

          id: Conmand1 # samle: cmd1, config1

          command: show interface { $fi.intfName1 } # only for type=Command

          description: xxxxx

          parser: "./VisualParser.xpar"

          diagnoses:

            - anchor: Paragraph1.intf_name # optional, only support the variables from current
config/command
```

```
    note: xxxxxxxx
  - anchor: Paragraph1.ip
    name: Check IP
    description: check the ip accuracy
    if:
      rule:
        loop_table_rows: true # the default is false
        conditions:
          - operand1: Command1.Paragraph1.ip[Current]
            operator: Does Not Equal #  Equals, Contains....
            operand2_type: Var
            operand2: Command1.ip[Baseline]
          - operand1: Command1.Paragraph2.mask
            operator: Does Not Equal
            operand2: ni.roleVarible1[Current]
        boolean_expression: not (A or B)
      then:
        note:
          color: red
          text: Diagnosis Alert $Command1.Paragraph2.mask,$ni.roleVarible1
        status_code_for_this_device:
          enable: true
          type: Alert # Success or Alert
        status_code_for_ni:
          enable: true
          type: Alert # Success or Alert
          note: Diagnosis Alert $Paragraph1.ip
```

From the analysis device section (highlighted), you can define one or multiple roles with each of its own conditions to match devices within FIG.

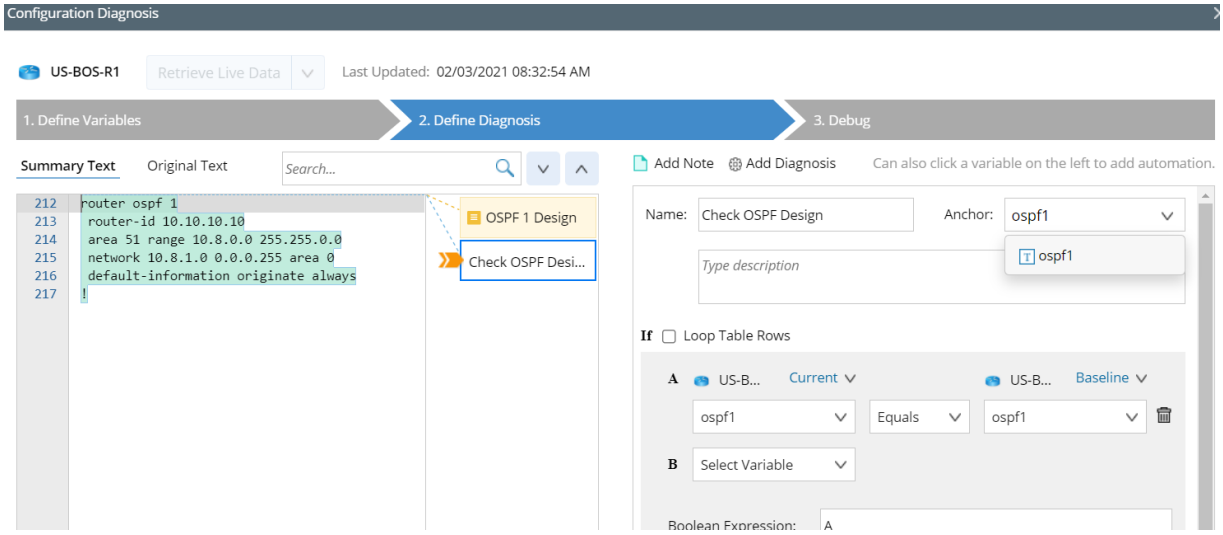
Device condition means how you can use the Sub FI variables and GDR variables as your condition to match devices.

You will then need to define the analysis logic. Depending on the need of performing analysis based on configuration file or CLI command, you can specify the type with the id.

```
commands:

  - type: Command # Command or Config

    id: Conmand1 # samle: cmd1, config1

    command: show interface { $fi.intfName1 } # only for type=Command

    description: xxxxx

    parser: "./VisualParser.xpar"
```

If you use CLI command here, specify the CLI command so you can continue using the sub FI variables. Please note that you cannot create visual parser within the YAML file directly, instead, you will need to import the Visual parser and reference the visual parser with its path. For more information on how to reference visual parsers, please find the instructions on the chapter: Referencing Visual Parsers

For the diagnosis section, you will need to define the anchor as well as the analysis logic.



The analysis rules here are straight forward and you can use compound variable and loop table like the way you use in the UI. Defining simple intra-device analysis logic is not complex, but if you want to define cross device analysis logic, please refer to: Cross Device Analysis in Network Intent

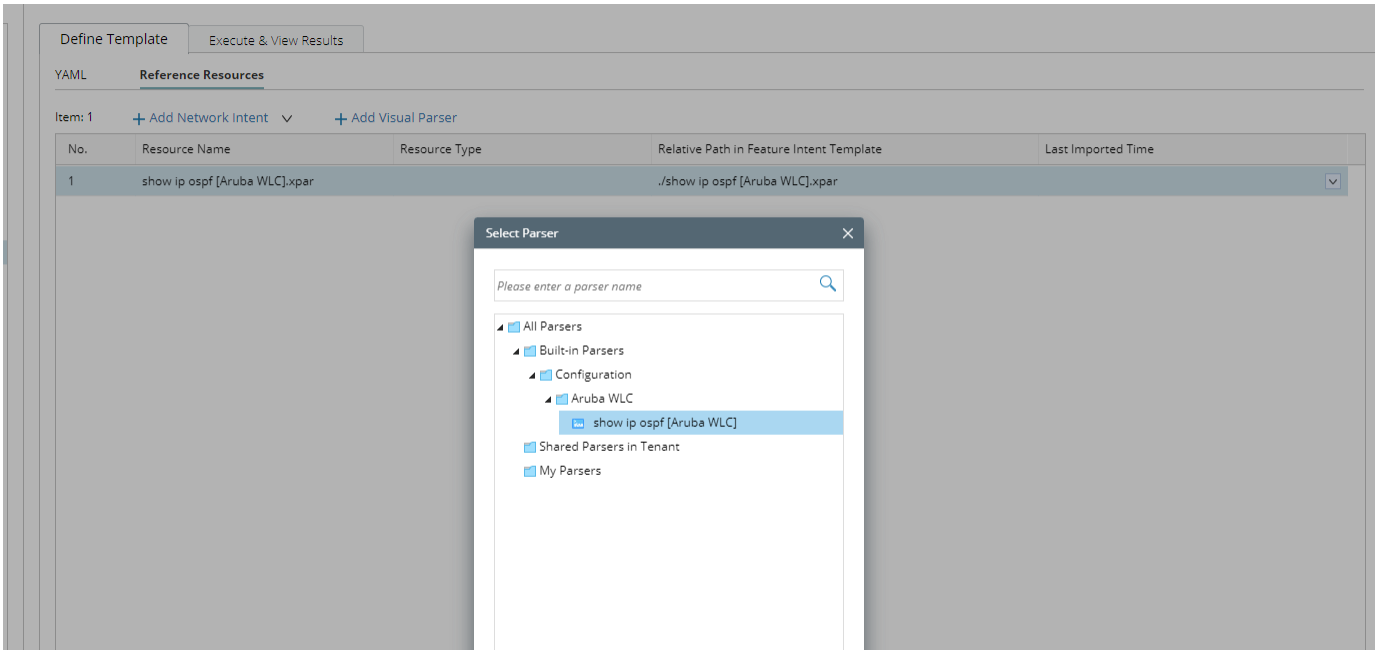
Then you can define the status code and the notes for the diagnosis.

If you want to define multiple roles each with its own device matching logic, simply add another role at the end with its device conditions.

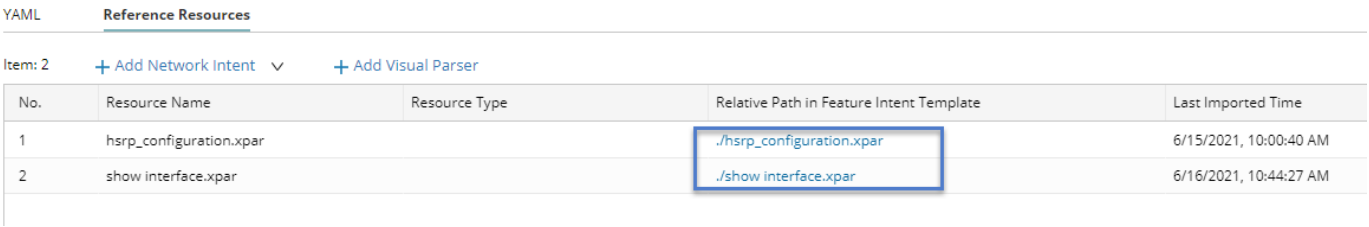
5.2.1.Referencing Visual Parsers

Since currently we don't support creating Visual Parsers within the YAML file directly, you can import an existing visual parser into the FIT and reference it within the YAML file by following the steps below:

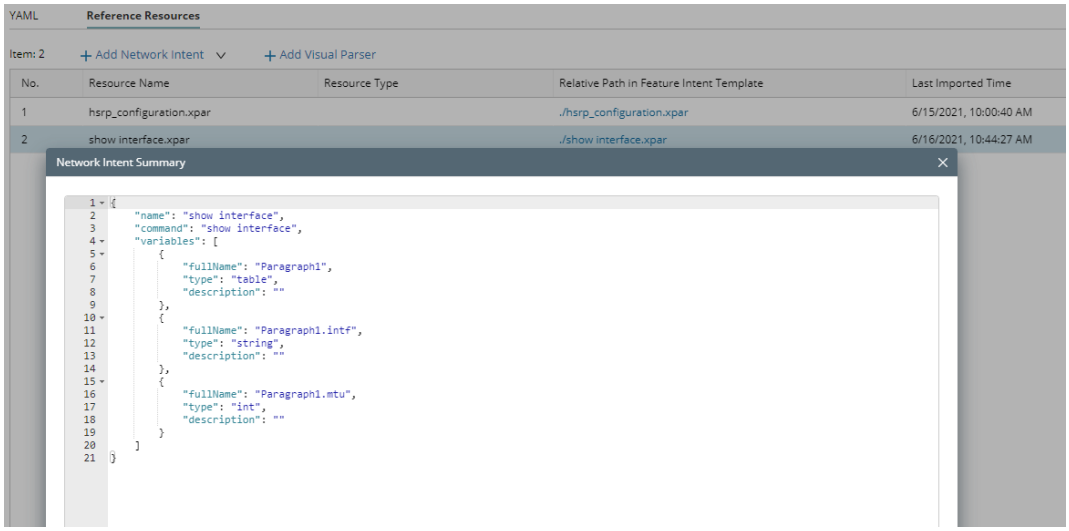
1. On **Feature Intent Template Manager** page, navigate to **Reference Resources** tab.
2. Click **+Add Visual Parser.**
3. Select the desired Visual Parsers and add them into the current FIT.



After adding the Visual Parser into the FIT, you can find the relative path that can be used within the YAMIL file.



To find out the parser variables within the current visual parser, simply click the **Resource Name**.



5.2.2.Cross Device Analysis in Network Intent

In the previous sections, we have explained the basics of how to define the analysis logic for intra-device analysis. However, it does not cover the scenario of defining the analysis logic for cross device check. You can leverage the `common_role_variable` to achieve this.

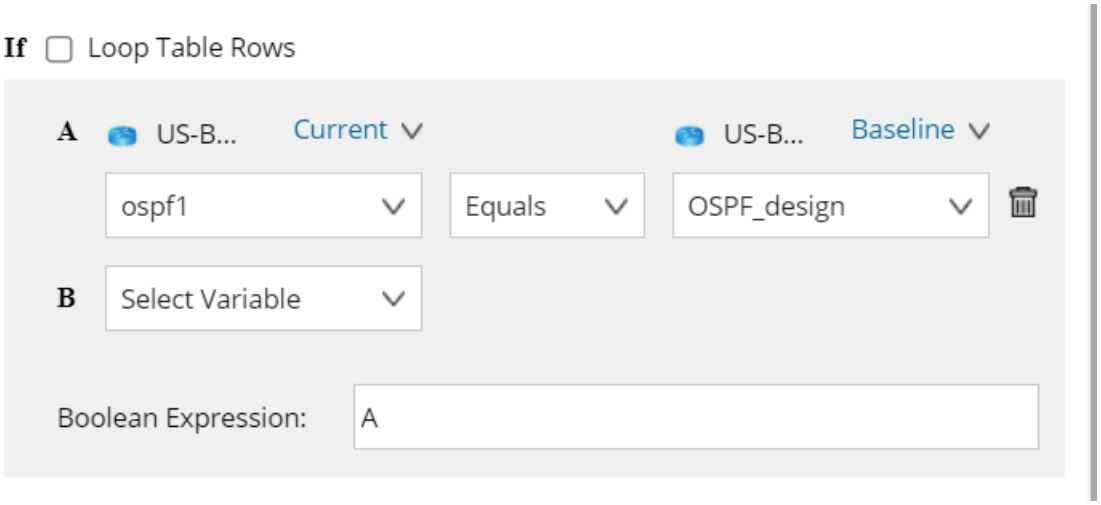
```
network_intents: # Definition for generating NI.

  ref_label: ospf

  path: xxxx/xxxx/General_BGP_{$fi.crossRelationHash} # the path to generate NI, here can use
variables of FIG level

  tags: [BGP, HSRP]

  lock_after_created: true # the default is false

  baseline_update_type: LatestFromDE # support values: LatestFromDE(Relative to the time of
running FID), LiveFromNextRun, the default is LatestFromDE.

  create_default_NI: false # Whether to create Default NI for the FIG, the default is false.

  common_role_variables:

    - name: roleVarible1

      from_role: role1 # role which the variable come from

      device_condition: true # expression OrderIndex("vanId")==0

      variable: "Conmand2.var3"

  analysis_devices:

    - device_role: role1

      device_condition: expression($fi.xxx)=="yyyy1"
```

```
      local_role_variables:

        - name: roleVarible2

          from_role: Role2 # role which the variable come from

          variable: "Conmand2.var3"

          device_condition: $device.xxx == $ni.role2.current.device.xxx # any or
expression   OrderIndex("vanId")==$this.xxx
```

In the highlighted sections, we have defined a `common_role_variables` as a variable that can be referenced in the section of device analysis. By defining the `common_role_variable` and device condition, you are specifying what device this variable belongs to. In the diagnosis section, you can use the common role variable as below.

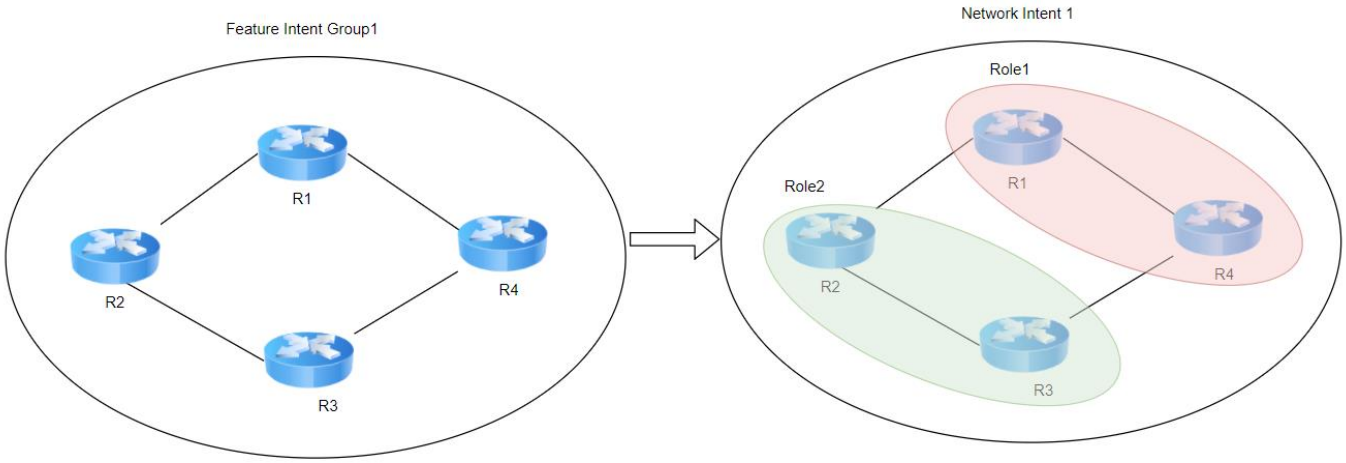
```
  diagnoses:

            - anchor: Paragraph1.intf_name # optional, only support the variables from current
config/command

              note: xxxxxxxx # donot support variable

            - anchor: Paragraph1.ip

              name: Check IP

              description: check the ip accuracy

              if:

                rule:

                  loop_table_rows: true # the default is false

                  conditions:

                    - operand1: Command1.Paragraph1.ip[Current] # [] support values:
Current,Last,Baseline, the default is Current

                      operator: Does Not Equal #  Equals, Contains....

                      operand2_type: Var # support types: Auto, Const, Var, ConstFromFITExp,
the default is Auto, Auto will be automatically recognized

                      operand2: Command1.ip[Baseline] # support to use FI or GDR variables if
operand2_type=ConstFromFITExp

                    - operand1: Command1.Paragraph2.mask

                      operator: Does Not Equal

                      operand2: ni.roleVarible1[Current]
```

After YAML file is executed and the Network Intent is generated, it will be automatically converted to different devices as specified so you will be able to define cross-device analysis.



Using local role variable:

Common role variable is a static variable that can be used by any device role. And it doesn't change no matter where and how you reference it. There are cases where you may want to define cross device analysis based on the different device conditions. Let's use the earlier example to illustrate the concept. In the picture below, we have two different NI roles (Role1 ad Role 2). And we want to define the cross relation device check for device R1 with R2, R4 with R3 as they share some identical characteristics. We can use the local role variables to achieve this:




```
analysis_devices:

  - device_role: role1

    device_condition: expression($fi.xxx)=="yyyy1"
```

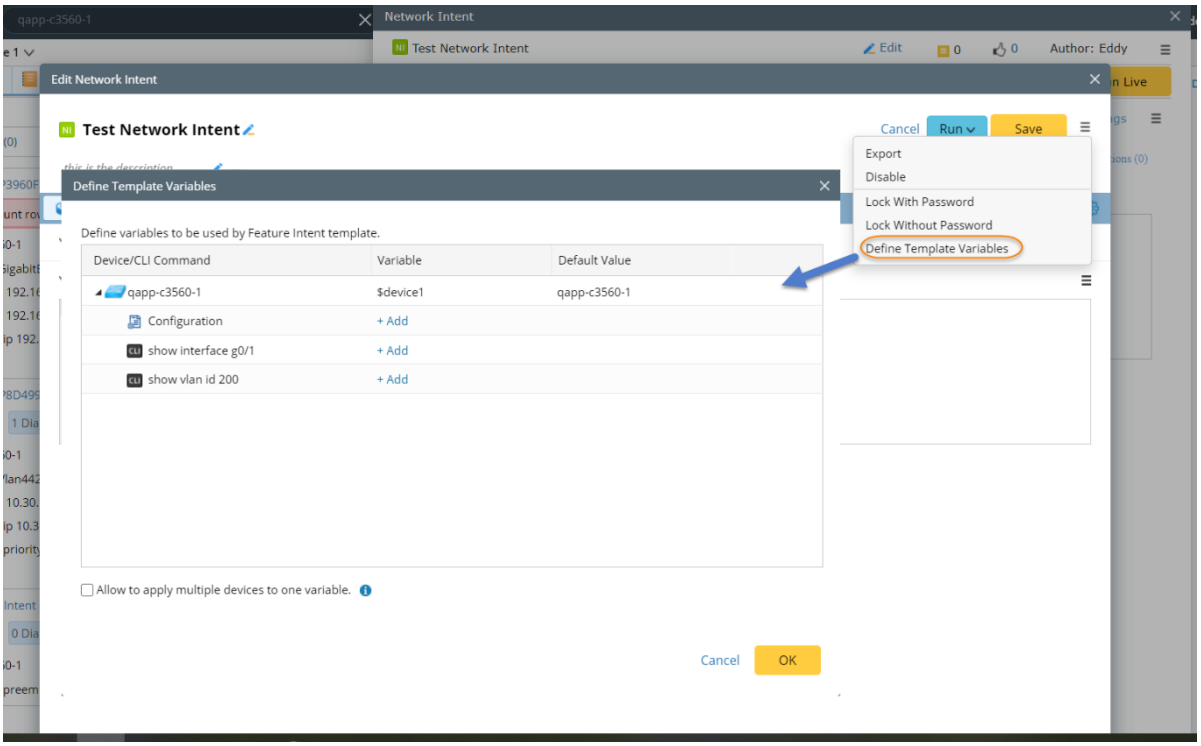
```
      local_role_variables:

        - name: roleVarible2

          from_role: Role2 # role which the variable come from

          variable: "Conmand2.var3"

          device_condition: $device.xxx == $ni.role2.current.device.xxx # any or
expression   OrderIndex("vanId")==$this.xxx
```

Under the analysis of role1, you can define a local role variable to find the expected variable of a device. When creating the analysis logic for device R1, the local role variable will loop through device R2 and R3 and based on the condition defined here, it will choose the device R2 as the expected device of the variable. When creating analysis for device R4, based on the condition defined, it will choose device R3 as the device of the variable.

5.3. Using Network Intent Template

Besides defining Network Intent directly within the YAML, there's another way to define the Network Intent with the analysis logic. That is using Network Intent Template.

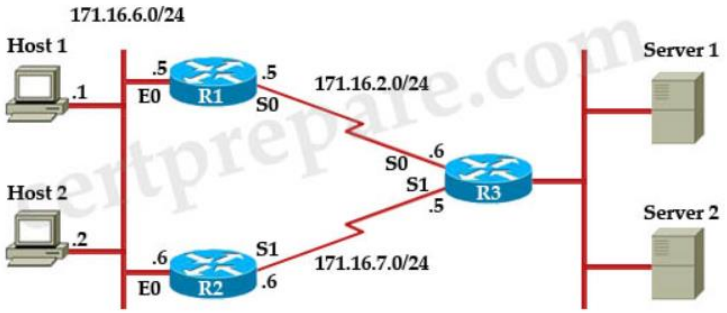
In the tutorial, we use the word network intent template to explicitly display the template function, but in the software there's no clear distinction between the regular network intents and network intent templates. You can use any network intent as the network intent template if the template variables are set up correctly. To set up the network intent template variables, you can turn on the edit mode of any network intent, the click **Define Template Variables** to open the **Define Template Variables** window.



You will find all devices defined in this network intent are listed here along with the their own CLI commands. Before looking into details on how variables can be used for other network intents, let's look at 2 different ways to create new network intents:

- Exact device count match：This means the device count of the newly created network intents needs to be precisely match the device count of the existing network intent when the automation analysis logic is applied.
- Adjustable device count match: This means the device count of newly created network intents can differ from what's defined within the network intent template. Check the option **Allow to apply multiple devices to one variable** in this case.

Let's look at the first method in which requires exact device count match. When the network intents require the exact same device count based on the network technology, you can use this method. Also, the analysis logic may require you to differentiate devices of the network intent. For example, HSRP requires 2 devices; one is an active device and the other one is a standby device. In the automation check logic of the network intent, you may define different checking logic depending on whether it's an active device or a standby device.

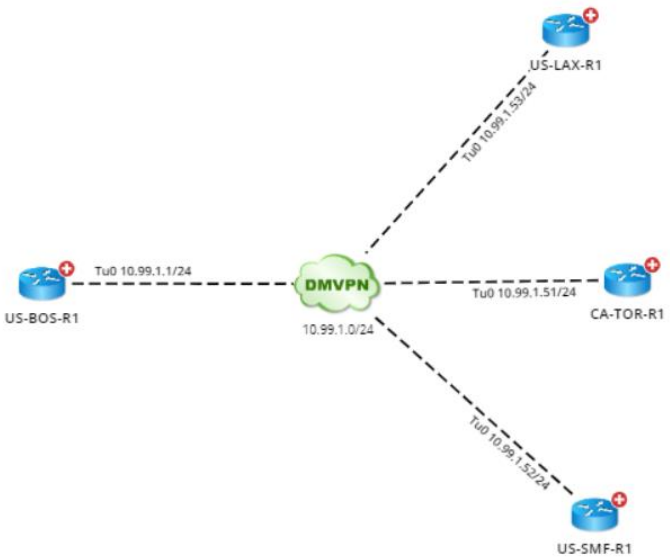



```
R1# show run
!
interface Ethernet0
 ip address 171.16.6.5 255.255.255.0
 no ip redirects
 standby 1 ip 171.16.6.100
 standby 1 priority 115
 standby 1 preempt
 standby 1 track Serial0
```

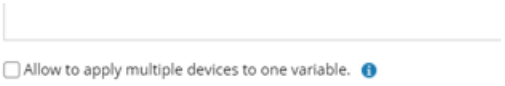
```
R2# show run
!
interface Ethernet0
 ip address 171.16.6.6 255.255.255.0
 no ip redirects
 standby 1 ip
 standby 1 preempt
 standby 1 track Serial1
```

(Use NetBrain map to replace picture later)

The second method doesn't require the exact device count match, and it can be used when you have a couple of devices grouped together for network technology that doesn't require exact device count. The following is a simple example of the IPSec designs for the WAN connections. You may define a network intent that includes the sites connected via IPsec tunnels. The network intent may consist of several devices. You can define the universal checking logic for all these devices. To create network intents for other WAN connections, you can simply reuse the automation check logic as the logic itself is device independent, and the device count could be greater or less.



For the network intents for HSRP check, we'll need to keep the exact device count so we'll leave the **Allow to apply multiple devices to one variable** option unchecked as shown below:

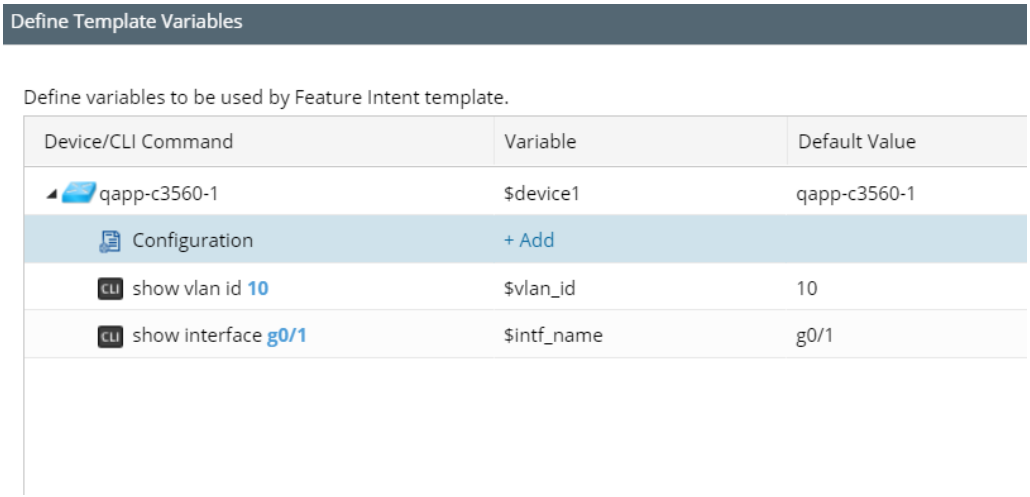


The next step is for you to define how devices and related show commands to be replaced by the new FI group we have found. The parameters we'll need to replace include the following parts:

- **Device Parameter**: replaced with the new name. The device variables are pre-defined and you can modify the variable name if needed.

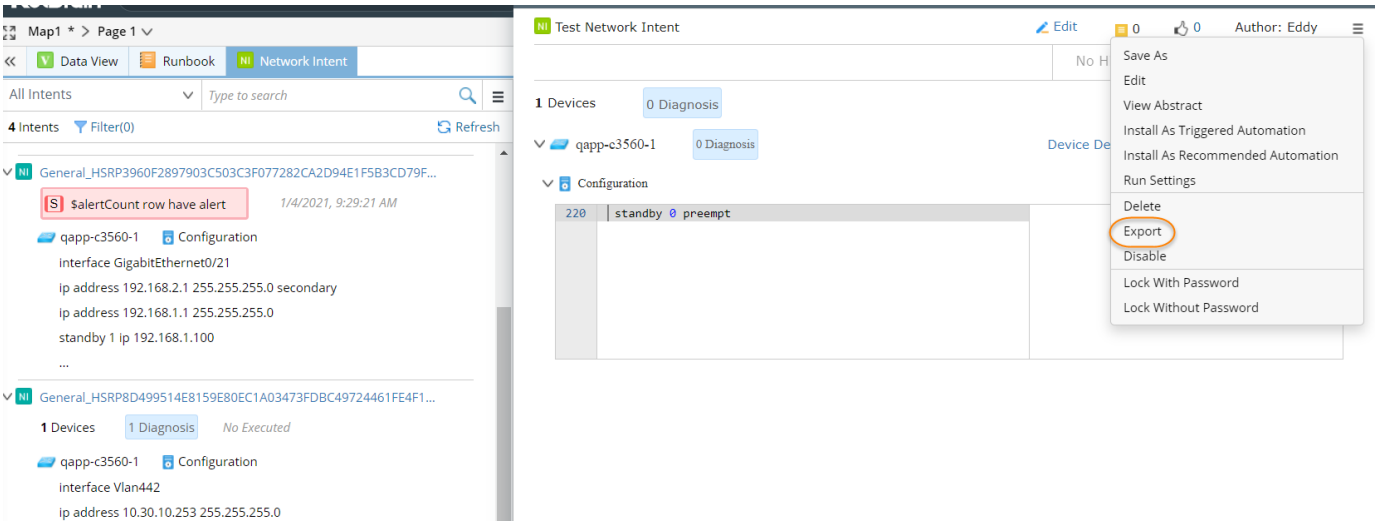
- **CLI Command Parameter**: if you use the CLI commands for certain config related to the current device, you will need to define it as a variable so the value can be replaced with the respective values of new devices.

    - **Interface Name**: In the network intent template, we'll need to check the interface status for WAN interfaces that faces the internet. We can use the show interface S0/1 command to achieve this goal. However, the WAN interfaces used is S0/1, for the new devices to be matched, the interface name may be different; you will need to define the interface name as a variable so it can be replaced with the new value.
    - **VLAN ID**: In the network intent template, we use the show interface id 10 to check the status for VLAN 10. However, for the new device to be matched, the VLAN id may vary. We'll need to define it as a variable so it can be replaced with the new value.



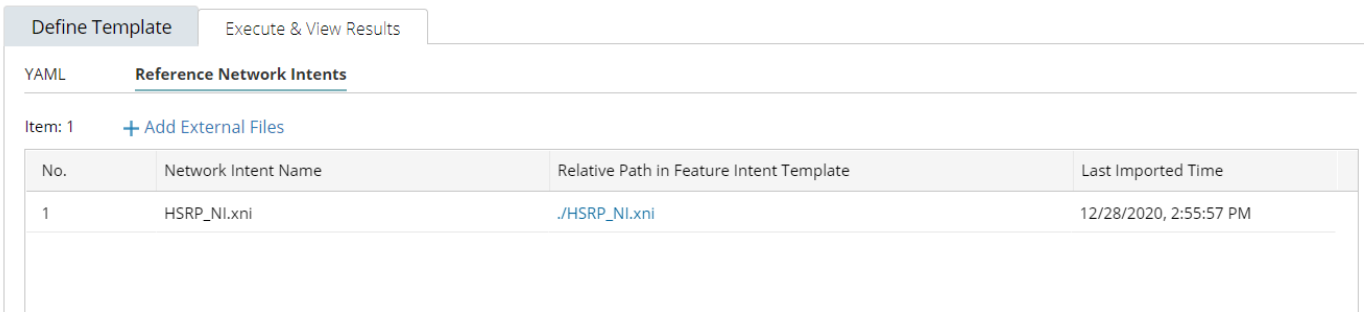
After defining the Network Intent Template Parameters, we will continue defining the variable mapping logic in the feature intent template.

After we define the template variables, this Network Intent can act as a template to generate network intents. You can also export the network intent and share with others.



5.3.1.Define Network Intent Parameters in Feature Intent Template

In the previous chapter, we have already defined the network intent template. Now it's time to implement it into our feature intent template. To reference the existing network intent template, we'll select the **Reference Network Intent** tab and then select the network intent template. You can either import the network intent from the external files or select an existing one from the current IE systems. After importing the network intent template into the feature intent template, you should be able to view the basic information for the NI template.



The **Relative Path in Feature Intent Template** is the directory where you can reference the relevant resources from the YAML file. Click the hyperlink, you will be able to view the NI template parameters that we defined previously. These parameters are organized in Json format which you can easily view and use in YAML file.

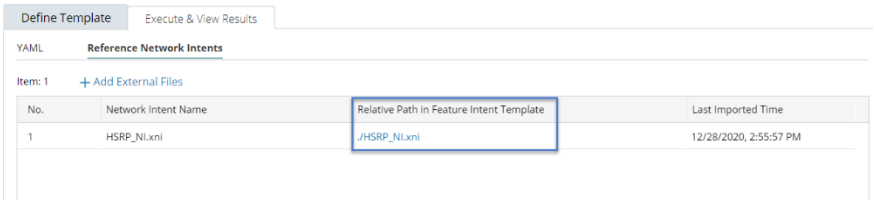
```
{
    "name": "HSRP_NI",
    "description": "",
    "author": "admin",
    "multipleDeviceToOneVariable": false,
    "devices": [
        {
            "varName": "$device1",
            "deviceName": "qapp-c3560-1",
            "commands": [
                {
                    "name": "Configuration",
                    "commandTemplate": "",
                    "variables": {},
                    "statusCodes": []
                },
                {
                    "name": "show standby",
                    "commandTemplate": "",
                    "variables": {},
                    "statusCodes": [
                        {
                            "alertCondition": "IfTrue",
                            "ni_status_code": true,
                            "ni_status_code_message": "$alertCount row have alert",
                            "device_status_code": false,
                            "alert_message": "this  is  a test"
                        }
                    ]
                }
            ]
        }
    ]
}
```

Now let's go back to the YAML file section and define how to use the NI parameters. Within the network intent section, we have explained some of these fields in the previous chapters. When using the network intent template, you will need to set the flag of `creat_dafault_NI` to false.

```
network_intents:

  path: xxxx/xxxx/General_BGP_{$crossRelationHash}

  conflict_mode: Skip

  lock_after_created: true

  create_default_NI: false

  cli_baseline_update_type: LatestFromDE

  ni_template: "./NI_template1.ni"

  ni_inputs:

    devices:
```

```
    Device1:
      device_condition: OrderIndex('$priority')==0
      parameters:
        intf: $intfName1
        ip: $ip1
    Device2:
      device_condition: OrderIndex('$priority')==1
      parameters:
        intf: $intfName1
        ip: $ip1
```

As we copy the **Relative Path from Feature Intent Template**, we can use the network intent by referencing its relevant path.



The `ni_inputs` field is where you can define matching rules of the `ni` template variables.

- **device_condition**: specify how the devices in the FI group should be mapped to the devices within NI template. You could also use the global functions provided by NetBrain to differentiate different devices within the FI group.
- **Parameters**: define the parameters to be mapped to the NI template.

As we explained previously, in this HSRP example, we want to use the `OrderIndex()` function to identify the active device as well as the standby device. This function will index the devices within FI group by its priority in ascending order. The first device with the lowest priority can be referenced with the first index: `OrderIndex('$priority')==0` , so in the example above, device1 is the standby device in the NI template and in this way we can match the same device for the FI group we have generated.

Device2 is the active device and we can use the `OrderIndex('$priority')==1` to reference the active device created from the FI group.

Mapping multiple FI Group device to NI Template Device

In case the matching logic within the Network Intent can be used for multiple devices within the FI group, you can use the following statements and all matched devices in the FI group will be mapped to device1.

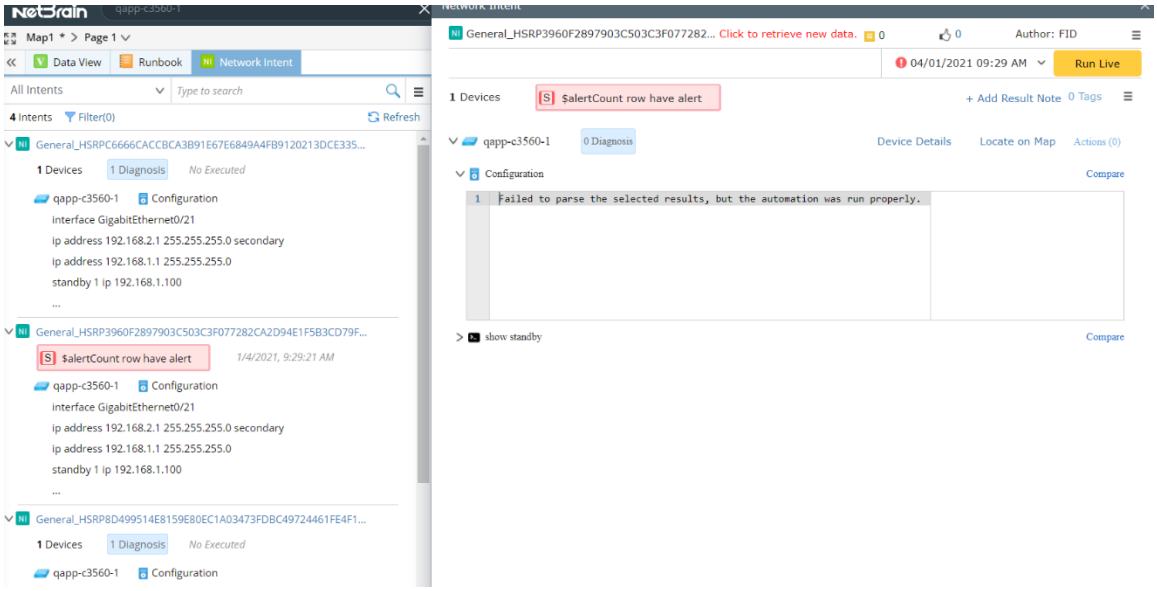
```
ni_inputs:
    devices:
      Device1:
```

```
    device_condition: True

    parameters:

      intf: $intfName1

      ip: $ip1
```

5.4. Generate Network Intents via NI Template

Once we have the feature intent template defined, we can execute the feature intent template and it will generate the network intent with automation logic.

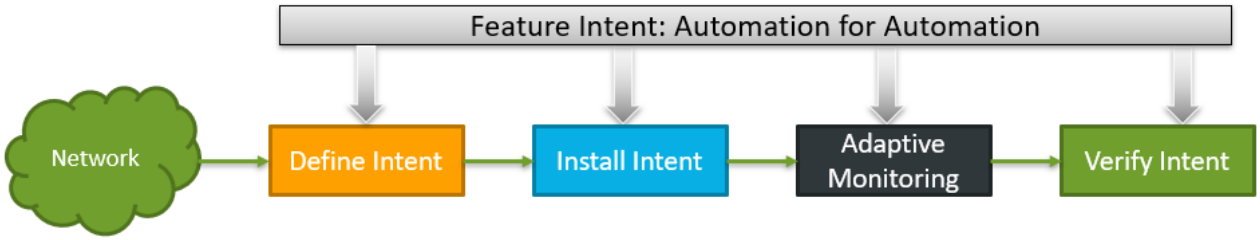
If opening one of the new network intents we have generated, you will find the CLI command has been replaced with the new values of the specific device.



Please also beware that the following contents from the previous network intent template were also shown in the newly created network intents:
·Drill-down CLI and Runbook Template
·Device level and global level status code

6. Building Intent-based Automation via Feature Intent Template

Network intent is the most complicated resource built by feature intent template. However, creating network intent itself is not adequate, as we need to define the proper ways to execute these network intents. In R10.0 release, we have invented the intent based automation functions that circle around network intent, allowing you to create powerful triggered automation functions based on network intent relying on adaptive monitoring function. We'll explain how you can create the adaptive monitoring probes with feature intent template and how to install the network intent into the adaptive monitoring systems. We'll also explain how to build the interactive automations and show the results by using decision tree.



6.1. Creating Flash Probe

Creating flash probe with feature intent template is quite straightforward. You need to define the similar fields within the YAML file by simply following the UI. Unlike network intent that requires the template to create new network intents, you can use all available fields in the UI to create flash probe from scratch. Let's look at a simple sample to illustrate the key components of flash probe. In the following example, we are trying to build a device level flash probe to check the CPU status of the device, and the key design points are shown as below:

- Use show processes cpu to retrieve the CPU status for Cisco IOS devices
- Primary flash probe that retrieves CLI command from live network every 30 minutes
- Alert will be raised if the CPU usage is greater than 90%
- CPU usage will be set to monitor variable so the historical data can be tracked

A sample YAML code is shown as below; we will explain details for each section:

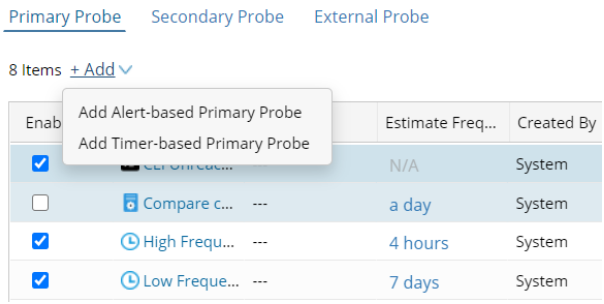
```
flash_probes:

  - name: overall_monitor_cpu_usage_check[Cisco IOS]

    description: "this is device CPU flash probe"

    target_type: Device

    qualification: |-

        $device.subTypeName  == "Cisco Router" && $device.subTypeName  == "Cisco IOS Switch"

    conflict_mode: Skip

    type: Primary
```

```
trigger_type: AlertBased

alert_source: ""

frequency_multiple: 0.5

frequency_interval: 1

variable_defines:

  - parser: "Built-in Files/Network Vendors/Cisco/Cisco IOS/show processes cpu [Cisco IOS]"

    parameters: {}

    variables:

      - name: five_min_cpu_usage

        alias: five_min_cpu_usage

        monitor:

          display_name: five_min_cpu_usage

          unit: "%"

rule:

  verify_table: false

  conditions:

    - operand1: $five_min_cpu_usage

      operator: GTE

      operand2: 90

  boolean_expression: A

alert_message: "CPU Is HIGH $five_min_cpu_usage ...."

enable: true
```

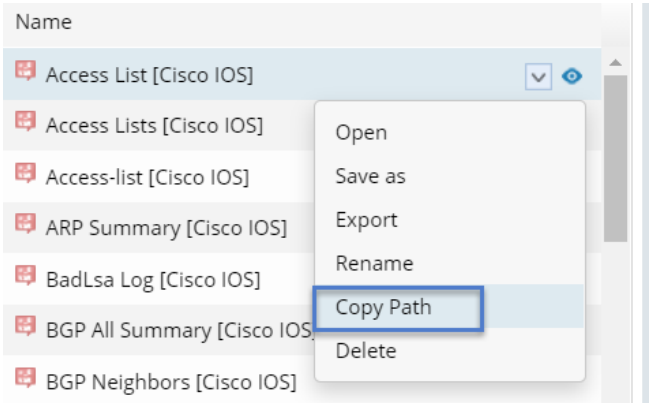
1) **Basic Information**: You need to define the basic information such as `name`, `display_name` and `description`.

2) **Target type**: Specify whether the probe is used for device level parameter check or interface level parameter check. CPU usage alert is for device level check, so we specify this as device level check.
   `target_type: Device`
3) **Qualification**: You can use the device level properties to filter devices. In this case as the flash probe is for Cisco devices, we'll filter the devices with device type.

4) **Type:** `Primary`, `Secondary` or `External`; as we want to retrieve the CPU data periodically, we'll set the type as `Primary`.
5) **Trigger type:** `AlertBased` or `TimerBased`, which correspond to alert-based flash probe or timer-based flash probe in the UI. CPU high usage is used for generating alerts, so we'll set this as `AlertBased`.



6) **Alert_source:** This is only for External Probe to specify the 3rd party name such as Splunk, Solarwinds, PRTG, etc.
7) **Frequency:** the multiple of the period of the base frequency.
8) **Variable_define**: You need to specify the parser directory as well as the variables. To easily define the parser directory in here, you can right click on a parser and select **Copy Path** to find the path directory.

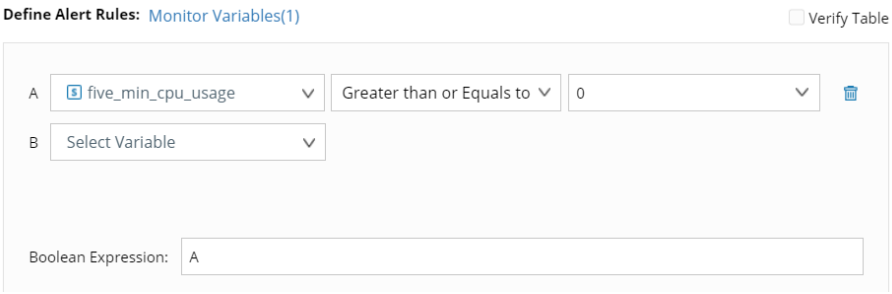


- **Parameters**: this needs to be defined in the CLI command
- **Variables**:
  - **Name**: the original variable name from the parser.
  - **Alias:** the alias for this variable to appear in alert check. As multiple parsers may include same variable name, you can use the alias to make the variable unique when referencing them within the alert check section.
  - **Monitor:** specify whether this variable will be monitored to determine if the historical data can be tracked. In this case, as we want to keep track of the historical data for CPU usage, we'll specify this as the monitoring variable.

```
- parser: "Built-in Files/Network Vendors/Cisco/Cisco IOS/show processes cpu [Cisco IOS]"
  parameters: {}
  variables:
    - name: five_min_cpu_usage
      alias: five_min_cpu_usage
      monitor:
        display_name: five_min_cpu_usage
        unit: "%"
```

9) **Define Rules**: Define the alert rules in the rule section.
   - **Operand1**: Stands for the variable needed to be checked

- **Operator**: Operator is the same as what can be defined using UI. You can find from the UI for a complete list of operators and how to use them.
- **Operand2:** Stands for the threshold value you define for the variable. This field is omitted for some of the operator types.

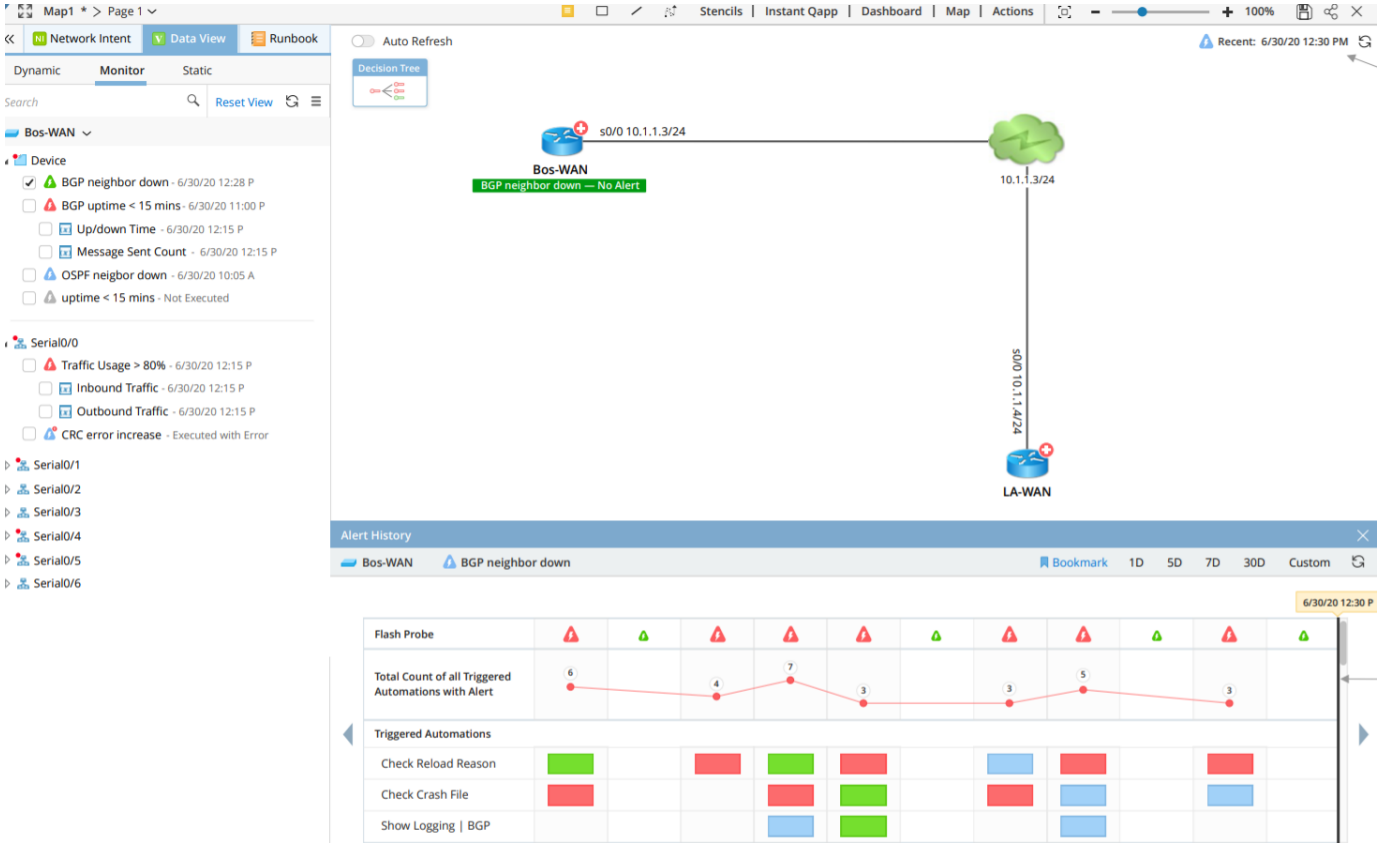



```
rule:
  verify_table: false
  conditions:
    - operand1: $five_min_cpu_usage
      operator: GTE
      operand2: 90
  boolean_expression: A
```

10) **Alert message**: Define the alert message that will be generated.

Run Feature Intent Template to create Flash Probes:

After defining the flash probe, you can run the feature intent template to generate the related flash probes. As these probes are enabled according to the definition, you will see these variables are executed based on the defined frequency. To view the results for these flash probes, you can open a map and use the monitoring data view to see the probe execution results.



6.1.1.Creating Flash Probe for Interface Variable Check

We have explained how to create a primary flash probe for device level check, now let's take a look at how to create a flash probe for interface variable check. In this example, we are trying to create a new interface level flash probe to check if link error is increasing. And the design points are shown as below:

- Use show interface to retrieve the interface error for Cisco devices
- Primary flash probe that retrieves CLI command from live network every 30 minutes
- Alert will be raised if the number of the interface errors retrieved during the current polling cycle is greater than that of the last polling cycle.
- We will only enable the check on interfaces that are in up status and at the same time filter loopback interfaces that we are not interested in.
- Interface errors will be set as monitoring variable so the historical data can be tracked.

The sample YAML code is shown as below, we will explain details for each section:

```
- name: overall_monitor_interface_link_error_check

    description: "this is interface link error flash probe"

target_type: Interface

target_interface:

    qualification: |-
```

```
        $device.subTypeName == "Cisco Router" && $device.subTypeName == "Cisco IOS Switch" &&
intf.isLoopback == false && $intf.intfStatus =="up/up"

    split_by_interface: false

    conflict_mode: Skip # support values: Override and Skip, the default value is Skip

    type: Primary # Primary or Secondary or External

    trigger_type: AlertBased

    alert_source: ""

    frequency_multiple: 16

    variable_defines:

      - parser: "Built-in Files/Network Vendors/Cisco/Cisco IOS/show interface [Cisco IOS]"

        parameters: {}

        variables:

          - name: intfs_table.input_error

            alias: input_error

            monitor:

              display_name: input_error

              unit: ""

          - name: intfs_table.output_error

            alias: output_error

            monitor:

              display_name: output_error

              unit: ""

      - compound_variable: input_link_errors

        value_type: int

        expression: $input_error - (last)

      - compound_variable: output_link_errors

        value_type: int

        expression: $otput_error - (last)

    rule:

      verify_table: false

      conditions:

        - operand1: $input_link_errors
```

```
          operator: GT # 1. UseGB, 2...

          operand2: 0

        - operand1: $output_link_errors

          operator: GT # 1. UseGB, 2...

          operand2: 0

      boolean_expression: A or B

    alert_message: "exist input link errors or output link errors,,,"

enable: true
```

1) **Target_type:** As this case focuses on interface level error check, we'll use the interface level variable, so we'll need to use interface to define `Target_type`.
2) **Target_interface:** Use this field to specify the interfaces you want to check for errors. All interfaces will be checked by default. If you would like to have all interfaces checked, you can leave this filed empty. In our case we keep this field empty in order to check all interfaces.
   You can also use the SubFI variables decoded so the system will create flash probe and include.

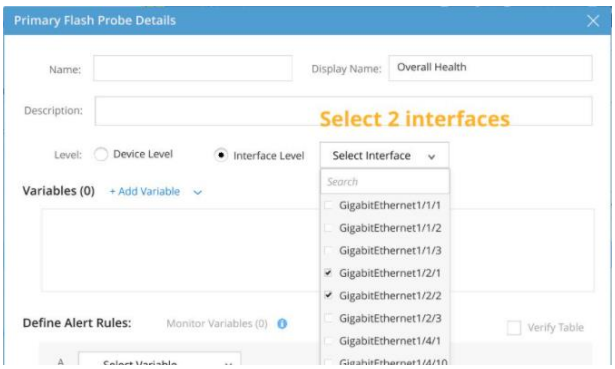
```
target_type: Interface

target_interface: $intfName
```

   Suppose you use the line pattern to decode devices with interfaces that has OPSF configured, you can further reference the `$intf_name` in the flash probe so the created flash probe will only include the OSPF interface status.

3) **Qualification:** Interface level check allows you to use interface properties. In this case as we want to focus on the interfaces that are in up status and filter all loopback interfaces, we can achieve our goals with the following statement.

```
          $device.subTypeName == "Cisco Router" && $device.subTypeName == "Cisco IOS
Switch" && intf.isLoopback == false && $intf.intfStatus =="up/up
```

4) **split_by_interface:** For flash probe that may include multiple interfaces, you will need to decide whether these interfaces should be grouped together into a single flash probe or separated into different flash probes. In this case, as we' want to use a single flash probe to include all these interfaces, we set the value to false so the flash probe can include all the matched interfaces.



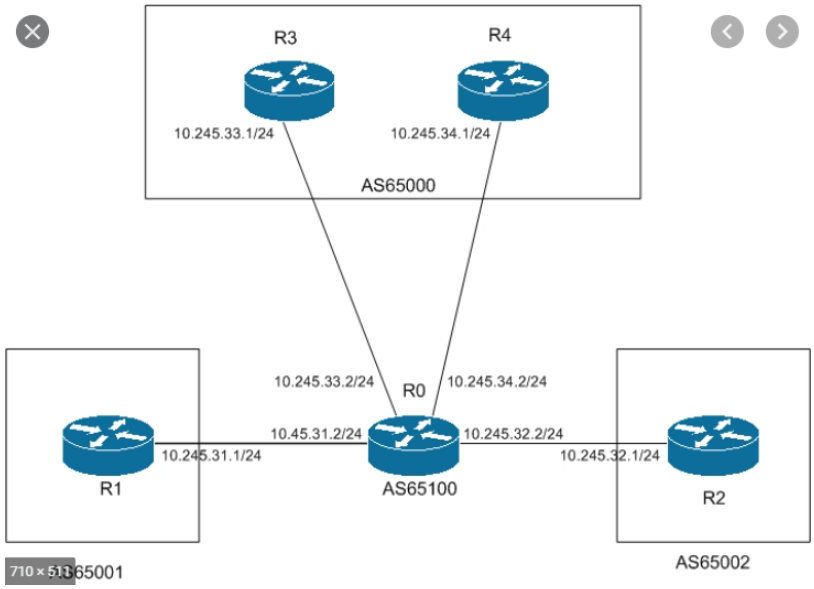
5) **Compound variable:** In this case, as we need to calculate the error increase count, the value of current errors status minus the value of error status retrieved last time. The compound variable can achieve this with the following definition:

```
    - compound_variable: input_link_errors

      value_type: int

      expression: $input_error - (last)
```

> **Note:** The last value can be referenced simply with `(last)`.

6.1.2.Creating Flash Probe using Sub FI variables

One of the most powerful function feature decoding provides is that the variables are reusable. We can extract them from the configuration files, and then reuse them in other automation resources. In this section, we'll see how to leverage the variables pared from the configuration files to use in flash probe. The following example demonstrates the concept:



In the above example, we have devices with BGP connections configured under vrf and we want to make sure the advertised routes don't change. In this case we can create flash probes to track the advertised route change. The YAML file to decode the BGP config files is shown as below:

```
name: Internal Test Feature BGP - Training

version: 1.0

source: ""

description: ""

tags: []
```

```
feature:
  qualification: {}
  configlet:
    sample: ""
    match_rules:
      - regexes: {}
        patterns:
          group1: |-
            interface $str:intfName1
            ip address $ip:ip1 $ip:mask1
          group2: |-
            MF: router bgp $num:bgp
            address_family ipv4 vrf $str:vrf_name
            O: bgp log-neighbor-changes
            F: neighbor $ip:ip2 remote-as $num:remoteBGP
            OE: neighbor $ip:ip3 update-source $str:intfName2
        split_keys:
          group1: []
          group2: [$fi.remoteBGP, $fi.vrf_name, $fi.ip2]
        relation: Equals($fi.intfName1, $fi.intfName2) && Equals($fi.ip2, $fi.ip3)
        merge_groups: []
```

You can see that all devices with desired configlet will be matched. And the flash probe we are trying to create is shown as below:

```
flash_probes:
  - name: BGP_adv_routes_change {$fi.ip2} {$fi.vrf_name}
    display_name: BGP_adv_routes_change {$fi.ip2} {$fi.vrf_name}
    target_type: Device
    qualification:
    type: Primary
    frequency_multiple: 1
```

```
variable_defines:
  - parser: "Built-in Files/CLI Command/Cisco IOS/BGP_adv_routes [Cisco IOS]"
    parameters:
        vrf: $fi.vrf_name
        ip: $fi.ip2
    variables:
      - name: fi.BGP_adv_route
        alis: BGP_adv_route
        monitor:
            display_name: bgp adv route table
    rule:
        verify_table: false
        conditions:
          - operand1: $BGP_adv_route
            operator: UseGB
        boolean_expression: A
    alert_message: BPG advertised route change detected
    enable: true
```

This example demonstrates the power of using line pattern to decode the configuration files and using the variable in our flash probe definition. From the configuration files decode, we are able to get the BGP neighbor IP address and corresponding vrf information for each neighbor, represented by the variables `$ip2` and `$vrf_name`.

In this section, as we are trying to create flash probe for all bpg neighbors, each with a unique flash probe name. To achieve that, we need to add the `ip` address and `vrf` name into the name field.

The CLI command to parser the data is:

```
show ip bgp neighbors $ip advertised-routes vrf $vrf
```

And the sample output is shown as below.

Alg.	Dest.Addr	Mask	Distance	Metric	Interface	Next Hop IP ...	Next Hop Dev...	Age
	104.207.208....	27		0				
	104.207.208....	28		0				
	104.207.208....	29		0		104.207.208....	BUR12-LAB-F...	
	104.207.208....	29		0		104.207.208....	BUR12-LAB-F...	
	172.16.8.0	22		100		169.254.241....		
	172.26.0.0	24		100		169.254.241....		
	172.85.32.40	29		0		104.207.208....	Bur12-bdf-fw...	
	199.188.232.0	22		0		172.16.1.99		

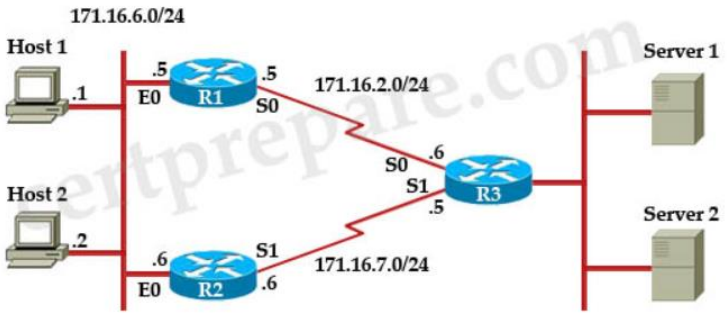
We are able to assign these variables with the value parsed from configuration files. Following the golden baseline concept, we automatically create baseline for the bgp-advertsied route table and generates alert if needed.

6.2. Install Network Intent into Flash Probe

With the network intents we have created, we'll need to define how they should be executed, the feature intent template provides you the method to install them into the flash probe. So when there's alert generated by the flash probe, the network intent diagnosis will be further executed.

In the following example, we are trying to achieve the network intent check based on flash probe:

- We have defined a network intent for HSRP check of active and standby devices.
- We have defined the flash probe to track the interface utilization.
    - WAN link utilization for standby device should be less than 1%.




```
R1# show run
!
interface Ethernet0
 ip address 171.16.6.5 255.255.255.0
 no ip redirects
 standby 1 ip 171.16.6.100
 standby 1 priority 115
 standby 1 preempt
 standby 1 track Serial0
```

```
R2# show run
!
interface Ethernet0
 ip address 171.16.6.6 255.255.255.0
 no ip redirects
 standby 1 ip
 standby 1 preempt
 standby 1 track Serial1
```

The purpose of this case is to track the WAN link utilization for standby devices periodically. If the traffic is greater than the defined threshold, we'll trigger the network intent to check whether the HSRP failover happens. To achieve this goal, we can define the flash automation as follows:

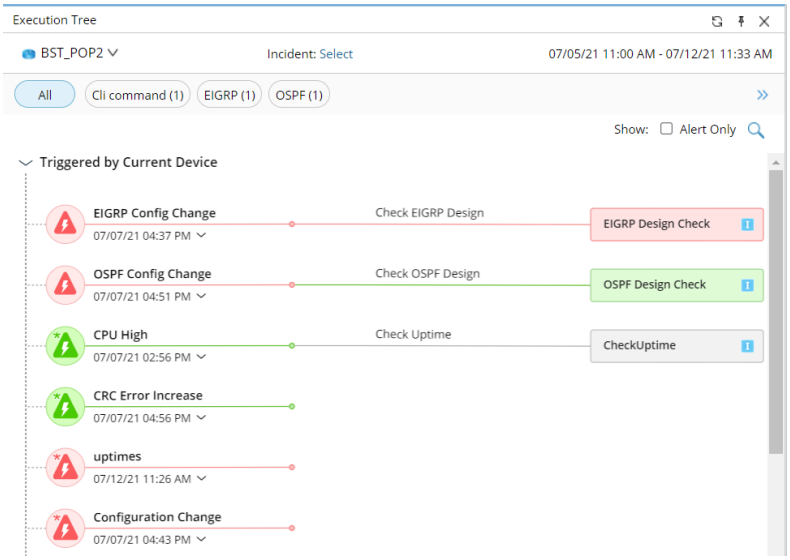
```
triggered_automation:
```

```
- description: xxxx
  automation:
    network_intents:
    auto_append_created_NI: true
  triggered_by_flash_probes:
    - name: wan_link_ultilzation_spike
      alert_source: NetBrain
      qualification:
        $device.subTypeName=="Cisco Router"
      note: Check bgp
      trigger_rule:
        run_type: Once
        frequency:
          interval: 2
          times: 3
        suppression:
          enable: true
          wont_run_twice_within: 2
  enable: true # whether to enable, the default is true
  conflict_mode: Skip
```

The key fields are explained as below:

1) **Automation**: You can define the network intents by specifying the directory in NetBrain. If network intents are created within this feature intent template, you can set the flag of `auto_append_createed_NI` as true to install the automation to specific flash probes.
   In this case, we assume the network intent for HSRP check already created within this feature intent template, so we keep it as true.
2) **Triggered_by_flash_probes**: Specify the flash probes you want to install to. The flash probe can be created within this feature intent template or from other feature intent templates. In this case, as we want the HSRP check to be triggered by `wan_link_ultilzation_spike`, we'll specify the flash probe name here.

   - Note: note shown in execution tree, usually indicates the reason why the network intent checked based on flash probe alert.
   - Trigger_rule: define how you would like the NI to be executed when there's flash alert, run once or run multiple times.

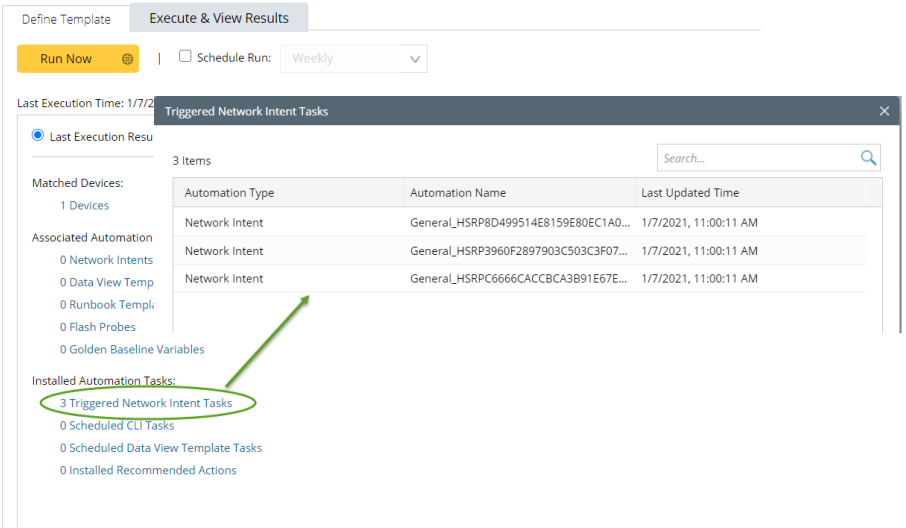


Schedule Network Intent

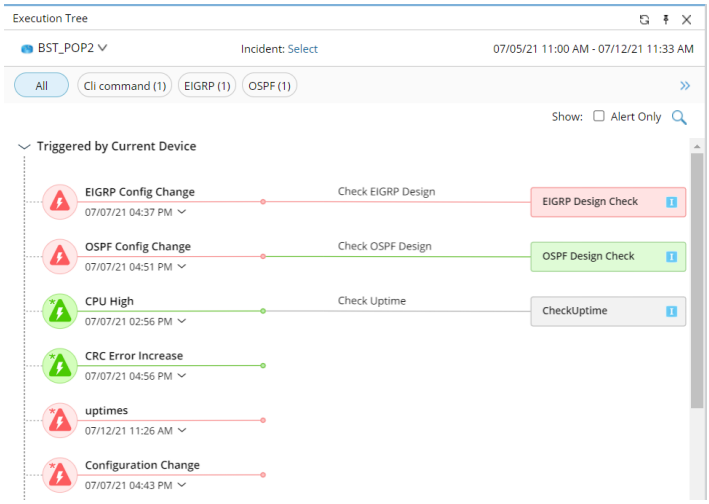
In addition to installing networking intent into an alert-based flash probe, you can also install the network intent into a timer-based flash probe. That's basically the same concept as scheduling network intent. The settings are the same as installing network intents into the alert-based flash probe, just that you will need the timer-based flash probe to trigger the network intent.

6.3. View Triggered Intent Results

After defining the install automation contents and executing the FI template, you can view the execution results right from there. All detailed information about how many automations and what automations are installed can be easily found.



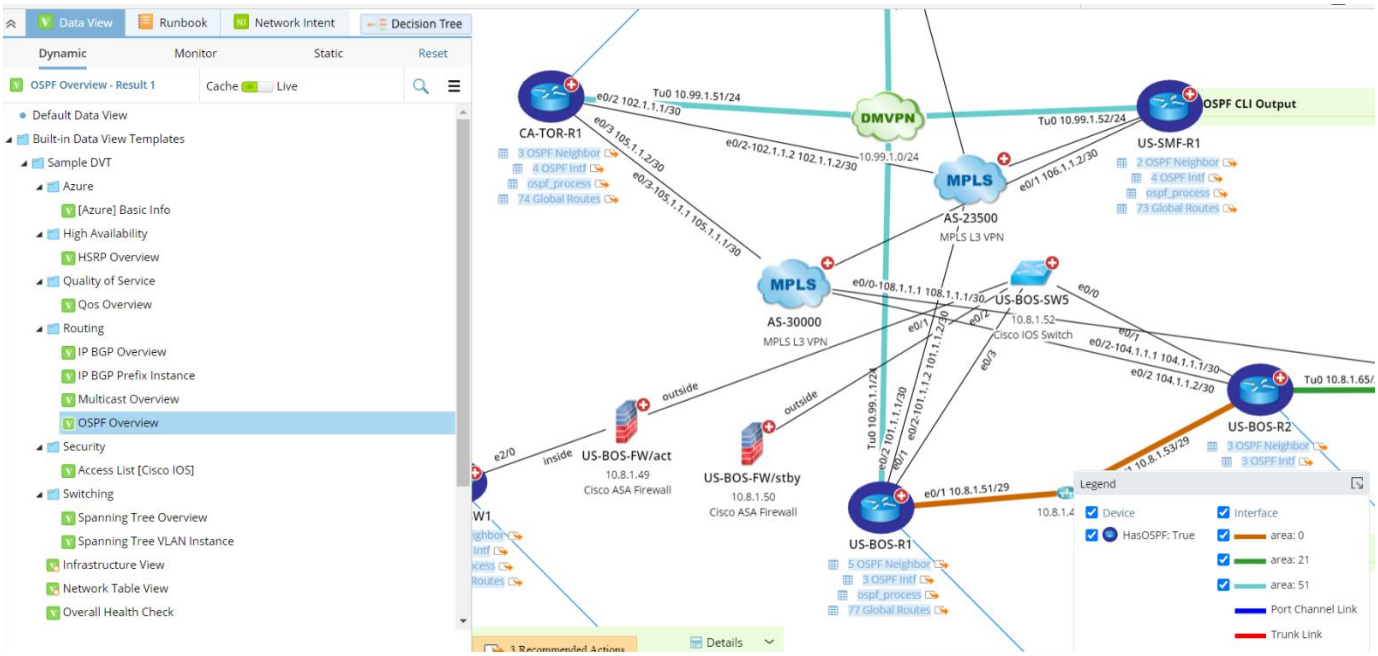
To view the results of automation task themselves instead of FI template you can use the execution tree to view the results per device.


Execution Tree
BST_POP2
Incident: Select
07/05/21 11:00 AM - 07/12/21 11:33 AM
All
Cli command (1)
EIGRP (1)
OSPF (1)
Show:
Alert Only
Triggered by Current Device
EIGRP Config Change
07/07/21 04:37 PM
Check EIGRP Design
EIGRP Design Check
OSPF Config Change
07/07/21 04:51 PM
Check OSPF Design
OSPF Design Check
CPU High
07/07/21 02:56 PM
Check Uptime
CheckUptime
CRC Error Increase
07/07/21 04:56 PM
uptimes
07/12/21 11:26 AM
Configuration Change
07/07/21 04:43 PM

7. Create/Update Data View Templates

Data View Templates can be used to understand the network designs and running status. You can create data view templates manually with the pre-defined qualifications so the data view templates can only be applied to specific devices. The current qualification supports basic GDR properties which might limit the accuracy of data view templates in some use cases.

With the Feature Intent qualification, you can match devices with the exact device feature and add these devices to the schedule Data View Template tasks.



One of the main problems resolved by Feature Intent template is how NetBrain service engineer can create data view templates and apply them to different customers. Feature Intent Templates indeed provides the natural way within NetBrain to define data view templates and adjust according to Network Change.

The following example shows how to define data view template and what functions of date view template are supported:

- **Define Qualifications**: Define the qualification for this data view templates.
- **Reference existing parser variables**: You will need to reference existing parser variables already in the system.
- **Define recommended automations**: You can define recommended automations for the current data view templates.

```yaml
dataview_templates: # DVTs that need to create
  - path: ">>Data View Templates>Public Data View Templates>xxxx"
    drill_down_actions: # dvt level drill down action
      - name: Basic Commands
        type: CLI # support types: CLI, RunbookTemplate
        commands: # cli commands
          - "show verson"
          - "show standby"
        description: xxxx
      - name: Multicast Source Tree Health Check
        type: RunbookTemplate # support types: CLI, RunbookTemplate
        path: "Built-in Runbook Templates/Troubleshooting/Multicast Source Tree Health Check" # runbook template path
        description: xxxx
    filter_criteria:
      device_types:
        - Cisco IOS Switch
        - Cisco Router
      feature_names: #FID  name
        - xxxx
    device_positions:
      - parser: "Built-in Files/CLI Command/Cisco IOS/Version [Cisco IOS]"
        variable: cpu
      - parser: "Built-in Files/CLI Command/Cisco IOS/Interface [Cisco IOS]"
        variable: intfs_table
        index: 5 # the dataview position index, if not set, the positions are set in array order by default.
    interface_positions:
      - interface_type: IPv4 Interface
        positions:
          - parser: "Built-in Files/CLI Command/Cisco IOS/Interface [Cisco IOS]"
            variable: intfs_table.status
            index: 5
```

Refer to the table below for a full list of supported properties and definitions:

Properties	Type	Must	Definition
path	string	Y	full path of the Data View Template
node_type	enum	N	In IE 10.0, node_type only supports Legacy. In the future, we will consider more node types including: • Fabric Node. • EPG • Bridge Domain • VRF • L2Out • L3Out • Contract • Cluster Virtual Standard Switch • Virtual Standard Switch • Vmware Distributed Virtual Switch • etc.
drill_down_actions	object	N	In IE 10.0, we only support • Execute CLI Commands. • Runbook Template.

drill_down_actions.commands	list - string	N	Show command.
drill_down_actions.runbook_templates	list - string	N	Full path of runbook template.
filter_criteria	object	N	Filter Criteria, see https://www.netbraintech.com/docs/ie100a/help/index.html?advanced-search.htm#dynamic-search for more details
filter_criteria.device_types	list - string	N	Device Type name list: • Cisco IOS Switch • Cisco Router • All device type name, please see 10. Device Type Name
filter_criteria.feature_names	string	N	Match GDR property: "_nb_features"
device_positions	object	N	
device_positions.parser	string	N	Parser path
device_positions.variable	string	N	Variable name
device_positions.index	enum	N	Data view position: 0, 1, 2 or 3, ...19.
interface_positions	object	N	
interface_positions.interface_type	enum	N	• IPv4 Interface • Interface • IPv6 Interface • IPsec VPN Interface • GRE VPN Interface
interface_positions.positions	object	N	
interface_positions.positions.parser	string	N	Parser path
interface_positions.positions.variable	string	N	Variable name

interface_positions.positions.index	enum	N	Data view position: 0, 1, 2 or 3, ...23.

7.1. Creating Data View Template based on FIG

Besides creating the FIT level Data View Template, you can create more specific Data View Templates based on FIG level resources and also use the parameters decoded from that FIG to generate more specific network designs.

The sample YAML file shown below can be used to create FIG level Data View Template:

```
dataview_templates: # DVTs that need to create

  - path: ">>Data View Templates>Public Data View Templates>xxxx_{$fi.eigen1}"

    create_for_level: FIG # FIG or Global, the default is Global

    ref_label: "refLabel1"

    tags: [BGP, HSRP]

    default_data_source:

      type: LiveRegularly # LiveOnce, LiveRegularly, CurrentBaseline, the default is
CurrentBaseline

      frequency: # for LiveRegularly

        every: 2 # unit is minute

        repeat_times: 3 # repeat run times, if null means no repeat
```

You will need to specify `create_for_level` to create resources at FIG level. You can create multiple data view templates based on multiple FIGs you have.

As these Data View Templates need to have different names, you will need to use the sub FI variable to differentiate them. The recommended method is to use the eigen variable for `cross_relationship`, as it is the unique key for each FIG.

```
- path: ">>Data View Templates>Public Data View Templates>xxxx_{$fi.eigen1}"
```

The benefits of creating FIG level Data View Template as well as other resources is that you can easily leverage the SubFI variables in these automation assets.

For the data view template, you can use the SubFI variables in multiple areas, but please make sure the value of SubFI variable for each FIG needs to be unique and there's no conflicting value within a single FIG. If the value of sub FI is not unique within a FIG, NetBrain will only be able choose the first one, which may not be your intended one.

- Input_Variables：The input variables (if required by the selected parser). The FI variable can be used to set the value for the input variable.

```
input_variables:

     - type: ParserParameter

       parser: "Built-in Files/CLI Command/Aruba WLC/IP Interface Brief [Aruba WLC]"

       parameters:

         - name: para1

           value: ($fi.eigen1+2)

           allow_manual_input: false # the default is false

         - name: para2

           value: $fi.eigen2

         - name: para3

           value: "\"GDR:$vlan\""
```

- Device Position: the FI variable can be used in device position to show sub FI level specific value in different Data View Templates generated by respective FIG.

```
device_positions:

     - parser: "Built-in Files/CLI Command/HP ProCurve Switch/NTP Status [HP ProCurve Switch]"

       variable: sntp_mode

     - type: Text # Text or ParserVariable, the default is ParserVariable

       title: "ospf: {$fi.var1}"  # GDR property name, not display name

       content: xxxxxxxxxxxxxx{$fi.var1}xx

Drill Down CLI Command: You can use FI variable in drill down command.

    - name: Basic Commands

       type: CLI # support types: CLI, RunbookTemplate

       commands: # cli commands

         - "show verson"

         - "show standby {$fi.xxxx}"
```

Please also note that you can use the FIG level Network Intent by creating FIG level Data View Template. To attach only FIG level Network Intent as the drill down actions of Data View Template, set the following parameter as true.

```
drill_down_actions: # dvt level drill down action

     - auto_append_created_NI_for_same_FIG: true # the default is false
```

8. Create/Update Runbook Template

Runbook Template can be used to organize the troubleshooting steps based on certain troubleshooting scenarios. Currently you can use the Runbook Template to create the following node types:

- Data View Template
- Qapp
- Overall Health Monitor Qapp: The Overall Health Monitor Qapp which can help you understand the running status and alert you if certain threshold is reached.
- Overall Health Monitor DVT: The Data View Template which can help you understand the running status for all technologies, across traditional devices, SDN and Cloud network.
- CLI

To create the Runbook Template, first you will need to specify the basic parameters as shown in the following YAML file:

```
runbook_templates:

  - path: "Shared Runbook Templates/{$fit.currentDomain}/xxx_{$fi.eigen1}"

    create_for_level: FIG # FIG or Global, the default is Global

    ref_label: "refLabel1"

    tags: [BGP, HSRP]

    qualified_devices: # support all the GDR properties that ES supports

      dynamic_search:

        conditions:

          - property: subTypeName

            values: [Cisco IOS Switch]

          - property: name

            operator: Contains # support types: Contains, Match, NotCotrains, NotMatch

            values: "BJ"

        boolean_expression: A and B

      static_include:

        auto_append_matched_devices: false # the default is false

        devices: [R1, R2]

        # qualification: $fi.xxx !="xxxxx"

      static_exclude:

        auto_append_matched_devices: false # the default is false

        devices: [R3, R4]
```

```
        # qualification: $fi.xxx =="xxxxx1"
```

The path represents the directory where you want to put these runbook templates. The resource level you are creating may vary according to different use cases (similar to Data View Template):

- Global: Global means within current FIT, we only create one single Runbook Template. In this case, you will need to specify the Runbook Template name without any FI variables.
- FIG Level: This means for each FIG, we will create corresponding Runbook Template, in this case, you will need to specify the Runbook Template name with the FI variable name to create different RBTs. Since these resources are related to these specific devices which reside in the current domain, you may want to create a folder (so these resources can remain in the current domain name) by specifying the following path to create resources:
  `-path:"Shared Runbook Templates/{$fit.currentDomain}/xxx_{$fi.eigen1}"`

You can define the pre-qualification for the current Runbook Template with the `dynamic_search` and also the static include/exclude statements.

If you are creating Global level resources, you can usually specify the pre-qualification with the dynamic search criteria by using GDR properties.

If you are creating FIG level resource, you should specify the pre-qualification with the static devices that has been matched to achieve decisive match. This can be achieved by setting the `auto_append_matched_devices` field to true.

```
qualified_devices: # support all the GDR properties that ES supports

      dynamic_search:

      static_include:

        auto_append_matched_devices: true # the default is false

        devices: [R1, R2]

        # qualification: $fi.xxx !="xxxxx"
```

Let's further take a look at how you can create different node types in the next three sections.

8.1. Adding DVT Node into RBT

Based on whether you are creating global level or FIT level Runbook Template, you have the option to attach the Data View Templates.

- Global Level: If you are creating global level Runbook Templates, you have the option to attach all DVTs created within the current FIT by using the following statement:
  `-auto_append_created_DVT:true# Whether to automatically add the created RBT by this FIT, the default is false`
  And also you can reference different Data View Templates by using their paths:
  `path:">>Data View Templates>Built-in Data View Templates>_Built-in Data for LWAP"`
- FIG Level: If you are creating FIG level Runbook Templates, you can attach the DVTs that are also created by the same FIG by using the following statement:
  `-auto_append_created_DVT_for_same_FIG:true`

8.2. Adding Qapp Node into RBT

Adding Qapp into the RBT is quite straightforward. As currently you cannot create Qapp within the FIT directly, you can reference an existing Qapp with its path.

```
    - type: Qapp # support types: DataviewTemplate, CLI, Qapp, OverallHealthMonitor,
OverallHealthView

        path: "Built-in Qapp/Qapp1"

        name: Qapp1

        qualified_devices: # optional

          dynamic_search: ~

          static_include: ~

          static_exclude: ~
```

8.3. Adding CLI Node into RBT

If the current RBT is created based on FIG, you can reference FI variables and use it in the CLI Command as shown in the sample below:

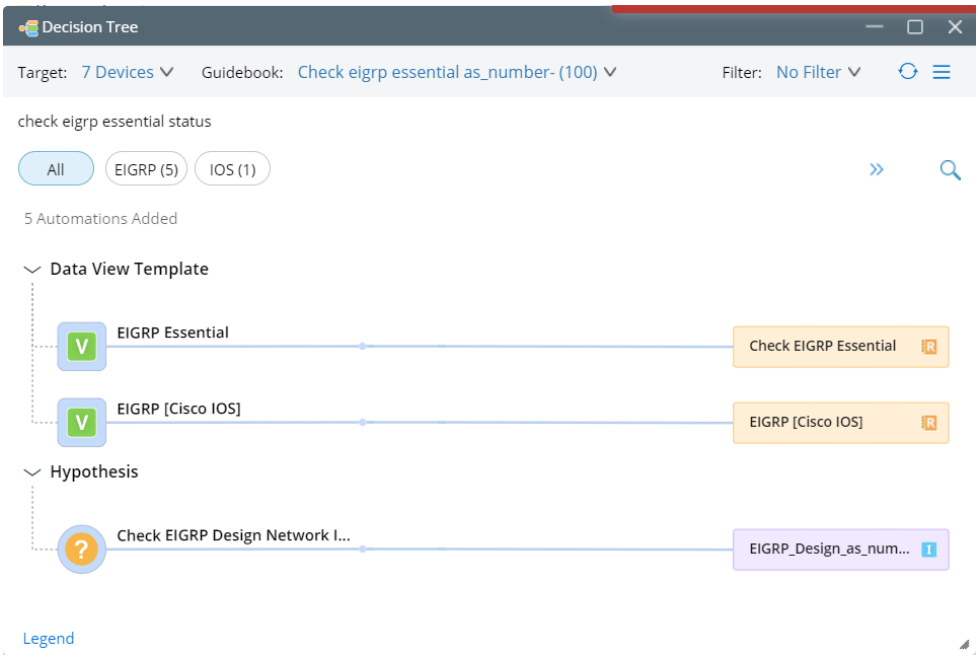
```
  - type: CLI

        name: Command List

        auto_append_feature_commands: true #

        commands:

          - show version

          - show interface

          - "show standby {$fi.xxxx}"

        qualified_devices: # optional

          dynamic_search: # support all the GDR properties that ES supports

            conditions:

              - property: subTypeName

                values: [Cisco IOS Switch]

              - property: name

                operator: Contains #
```

```
            values: "BJ"

      boolean_expression: A and B

    static_include:

      auto_append_matched_devices: false # the default is false

      devices: [R1, R2]

      # qualification: $fi.xxx !="xxxxx"

    static_exclude:

      auto_append_matched_devices: false # the default is false


      devices: [R3, R4]

      # qualification: $fi.xxx =="xxxxx1"
```

> **Tip:** You can also define the pre-qualification for this CLI node.

9. Create/Update Guidebook

Decision tree can help you create different guidebooks based on your troubleshooting scenarios. For more information about the decision tree and how to use guidebooks via UI, please refer to the document: Guidebook Feature Summary



You can use YAML to define Guidebook and organize the related automation assets. Similar to Data View Template and the Runbook Template, there are 2 methods to create Guidebook:

- Global: create guidebook globally based on current FIT.
- FIG: create guidebook based on the FIG matched.

You can use the `create_for_level` parameter to create resources based on your needs.

```
  - name: guidebook1 under ({ $fig.GetDeviceNamesStr() })

    create_for_level: FIG # FIG or Global, the default is Global
```

You can also define the pre-qualification for the matched devices, the logic here is similar to what's shown in Data View Template and Runbook Template section earlier:

```
qualified_devices: # support all the GDR properties that ES supports

      dynamic_search:

        conditions:

          - property: subTypeName

            values: [Cisco IOS Switch]

          - property: name

            operator: Contains # support types: Contains, Match, NotCotrains, NotMatch
```

```
        values: "BJ"

    boolean_expression: A and B

  static_include:

    auto_append_matched_devices: false # the default is false

    devices: [R1, R2]

    # qualification: $fi.xxx !="xxxxx"

  static_exclude:

    auto_append_matched_devices: false # the default is false

    devices: [R3, R4]
```

If you create the Guidebook globally, you will need to define the qualification based on the GDR or static devices that have been matched. If you create the Guidebook based on FIG, it's recommended to define the pre-qualification with `static_include` statement to achieve precise match. So, when you open a match with devices, only the qualified guidebooks will be shown for the easy troubleshooting purpose.

The following two types of automations can be added to guidebooks：

- Data View Template
- Hypothesis

9.1. Adding DVT Node into Guidebook

Based on whether you are creating global level or FIT level Runbook Template, you have the option to attach the Data View Templates.

- Global Level: If you are creating global level Runbook Templates, you have the option to attach all DVTs created within the current FIT by using the following statement:
  `-auto_append_created_DVT:true# Whether to automatically add the created RBT by this FIT, the default is false`
  And also you can reference different Data View Templates by using their paths:
  `path:">>Data View Templates>Built-in Data View Templates>_Built-in Data for LWAP"`
- FIG Level: If you are creating FIG level Guidebooks, you can attach the DVTs that are also created by the same FIG by using the following statement:
  `-auto_append_created_DVT_for_same_FIG:true`

9.2. Adding Hypothesis into Guidebook

When you add hypothesis into Guidebook, you will also need to specify what automation resources are to be added based on the hypothesis.

- RBT: add RBT to be associated with the defined hypothesis.

- NI: add NI to be associated with the defined hypothesis.

Similar to the way you are adding DVT into the Guidebook, you have the option to attach FIG or FIT level resources to the Hypothesis:

- Global Level: If you want to add these RBTs and NIs into the Guidebook, you can use the following methods:

```
auto_append_created_NI:True# Whether to add all the created NI by this FIT, the default is false
auto_append_created_RBT: True# Whether to add all the created RBT by this FIT, the default is false
```

  Also, you can add static Network Intents and Runbook Templates with their paths.

```
network_intents:
  - "HSRP/General HSRP1" # static NI
runbook_templates:
  - "Built-in RBTs/BGP/General HSRP" # static RBT
```

- FIG Level: If it's FIG Level Guidebooks, you can attach the FIG level Network Intents and Runbook Templates.

```
auto_append_created_NI_for_same_FIG:true# Whether to add the created NI for same FIG, only for create_for_level=FIG, the default is false
auto_append_created_RBT_for_same_FIG:true# the default is false
```

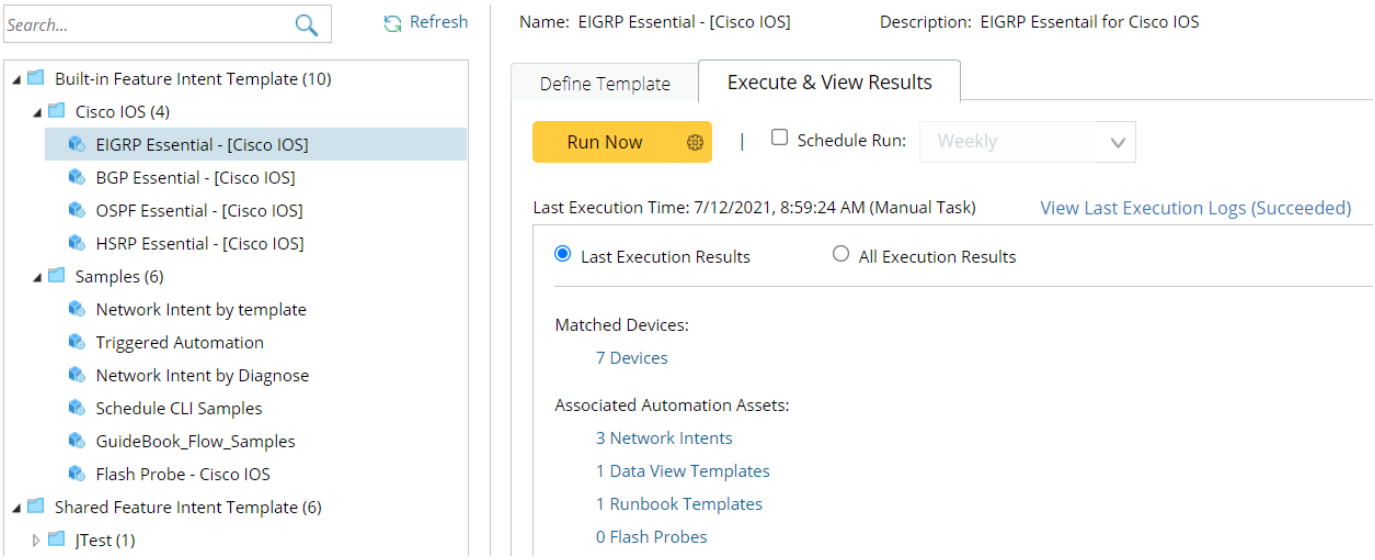
10. Scheduling Feature Intent Template

Network is dynamic as network changes constantly occur. To make sure the automation resources and tasks created by feature intent template are up to date, you can schedule the feature intent template to run periodically.

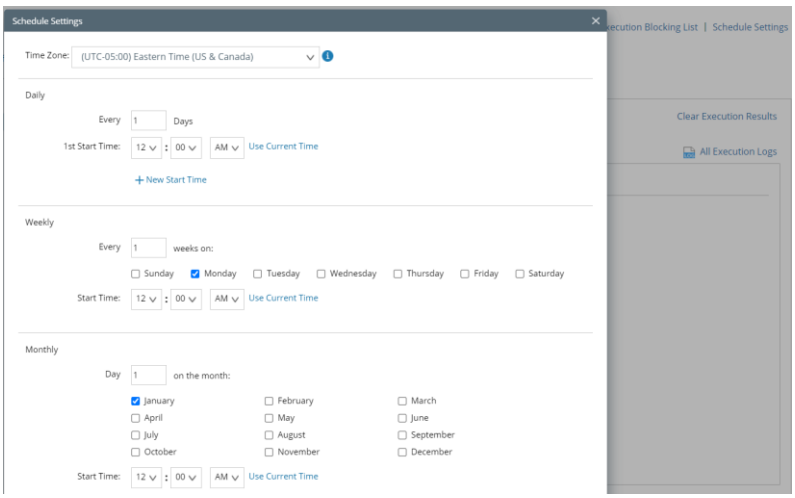
The system provides the option to run feature intent template daily/weekly/monthly, and you can define the proper frequency according to the network feature change frequency.

During the execution of the schedule task, the system will check the latest data and update the resources accordingly, and it will also remove any obsolete resources.

Feature Intent Template Manager



The execution time of schedule task can be found from the top right corner. All the feature intent templates defined with the same frequency will run at the same time to ensure all resources are updated.



11. More Functions Provided with Feature Intent Template

In this tutorial, we have covered the main functions of feature intent template and how it can help to scale the entire reference workflow. The list below enumerates all the other resources that can be created by feature intent template.

1. Automation assets can be created:
    - Flash Probe
    - Network Intent
    - Data View Template
    - Runbook
    - Golden Baseline
    - Guidebook
2. Automation assets can be installed to run:
    - Network Intent triggered by Flash Probe
    - Schedule Network Intent (triggered by timer-based Flash Probe)
    - Schedule Data View Template
    - Schedule Parser
    - Schedule CLI

For complete guide on how to create all these resources, please refer to the Feature Intent Template Online Help.