

TECOPS-2665

Reaching Network Automation Level 5 Principles and Practice

Per Andersson Ulrik Stridsman
Viktoria Fordos
NSO Engineering Architects

Learning Objectives

Once the Adaptive Service Activation Scripts (Network Automation Level 2) are left behind, a world of life-cycle flexibility and stylistic freedom of the services running in our network. What can we use this flexibility for?

In this lab, the participant will upgrade a pre-existing Network Services Orchestrator (NSO) service from Network Automation Level 3 to Level 5 by adding service health measurement and make it adapt to changes in the environment.

The vision of the Internet Engineering Task Force (IETF) Traffic Optimization (ALTO) Working Group (WG) is to optimize the application and network together, so that the delivered service is delivered as well as possible to the collection of users. This might mean delivering this user's Application Level service from a data center that is not the closest one, but from another with less load or lower energy prices, and therefore less expensive to the customer. Moving service instances around becomes a natural part of the life-cycle.

Prerequisites:

- Basic programming skills in Python
- Some familiarity with NSO services and YANG modules

Key takeaways:

- Hands-on experience with Network Automation Levels 3-5
- Giddy feeling of power arising from services that adapt to their environment

Disclaimer

This training document is to familiarize with Network Services Orchestrator (NSO) and Network Automation Levels. Although the lab design and configuration examples could be used as a reference, it's not a real design, thus not all recommended features are used, or enabled optimally. For the design related questions please contact your representative at Cisco, or a Cisco partner.

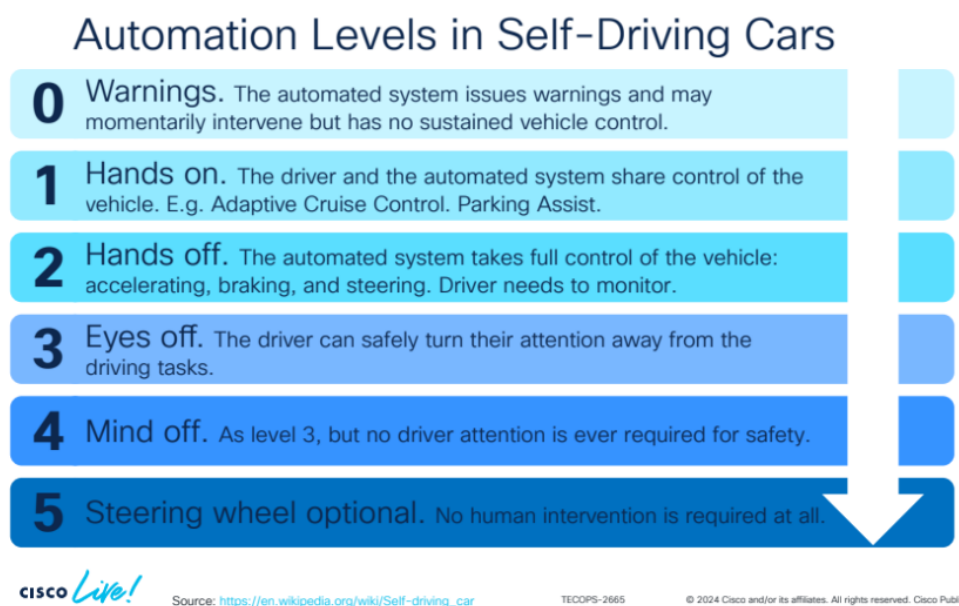
Learning Objectives.....	- 2 -
Disclaimer.....	- 2 -
Prep 1: The Scenario.....	- 5 -
Background	- 5 -
Video Delivery Service	- 6 -
Lab Starting Point.....	- 7 -
Prep 2: Getting into to the Lab Environment	- 9 -
Cloning and Running Locally	- 9 -
Using the Cisco DevNet Sandbox.....	- 9 -
Using the Cisco Cloud IDE	- 10 -
Using the Cisco Cloud IDE development environment.....	- 10 -
Prep 3: Building and Running the Level 3 System	- 14 -
Step P.1: Working in VS Code	- 14 -
Step P.3: Finding the Linux Command Prompt	- 15 -
Step P.4: Selecting Implementation and Building the System	- 15 -
Step P.5: Getting to the NSO Command Prompt.....	- 17 -
Step P.6: Updating the Project.....	- 17 -
Step P.7: Configuring a Service Instance.....	- 18 -
Step P.8: Looking at Operational Data and the Plan	- 21 -
Step P.9: Turning off Commit Messages	- 22 -
Task 1 Overview: Go to Level 4, Add Monitoring.....	- 24 -
Overview 1.1 Replace dc/is-active with dc/oper-status/jitter	- 24 -
Overview 1.2 Replace edge/dc with edge/oper-status/chosen-dc.....	- 25 -
Overview 1.3 Write a DC Selection Function in Python	- 25 -
Overview 1.4 Configure Skylight.....	- 26 -
Overview 1.5 React to Skylight Notifications	- 26 -
Overview 1.6 Build and Deploy	- 27 -
Overview 1.7 Test.....	- 27 -
Task 1 Detailed Instructions.....	- 32 -
Step 1.1 Replace dc/is-active with dc/oper-status/jitter	- 32 -
Step 1.2 Replace edge/dc with edge/oper-status/chosen-dc.....	- 33 -
Step 1.3 Write a DC Selection Function in Python	- 34 -
Step 1.4 Configure Skylight.....	- 36 -
Step 1.5 React to Skylight Notifications	- 38 -
Step 1.6 Build and Deploy	- 39 -
Step 1.7 Test.....	- 40 -
Task 2 Overview: Go to Level 5, Add Ongoing Optimization	- 41 -

Overview 2.1 Add edge-capacity and energy-price	- 41 -
Overview 2.2 Add edge-clients (optional)	- 41 -
Overview 2.3 Better DC Selection Function	- 42 -
Overview 2.4 Optimization doesn't have to be Notification Based	- 42 -
Overview 2.5 Keep Optimizing	- 43 -
Overview 2.6 Build and Deploy	- 43 -
Overview 2.7 Test	- 44 -
Task 2 Detailed Instructions	- 51 -
Step 2.1 Add edge-capacity and energy-price	- 51 -
Step 2.2 Add edge-clients (optional)	- 51 -
Step 2.3 Better DC Selection Function	- 52 -
Step 2.4 Optimization doesn't have to be Notification Based	- 53 -
Step 2.5 Keep Optimizing	- 54 -
Step 2.6 Build and Deploy	- 55 -
Step 2.7 Test	- 55 -
Summary	- 56 -
Related Sessions at Cisco Live	- 57 -
Beyond Cisco Live	- 58 -

Prep 1: The Scenario

Background

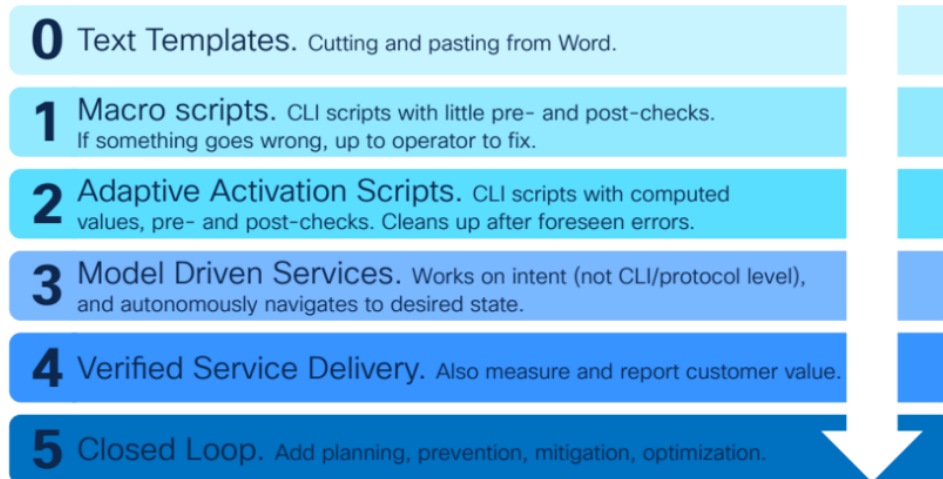
As an industry, we have been talking for years about “network automation”. The exact meaning of that concept has often been rather nebulous. Attempts at defining more specific levels, similarly to how the auto industry defined specified levels of “self-driving cars”.



Even with the examples showing what the network automation levels mean, however, people did not get a very clear and intuitive understanding of what the levels mean.

The purpose of this lab is to give the participants a concrete, hands-on, intuitive feel for the meaning of level 3, level 4 and level 5 network automation.

Automation Levels in Self-Driving Networks



cisco *Live!*

TECOPS-2665

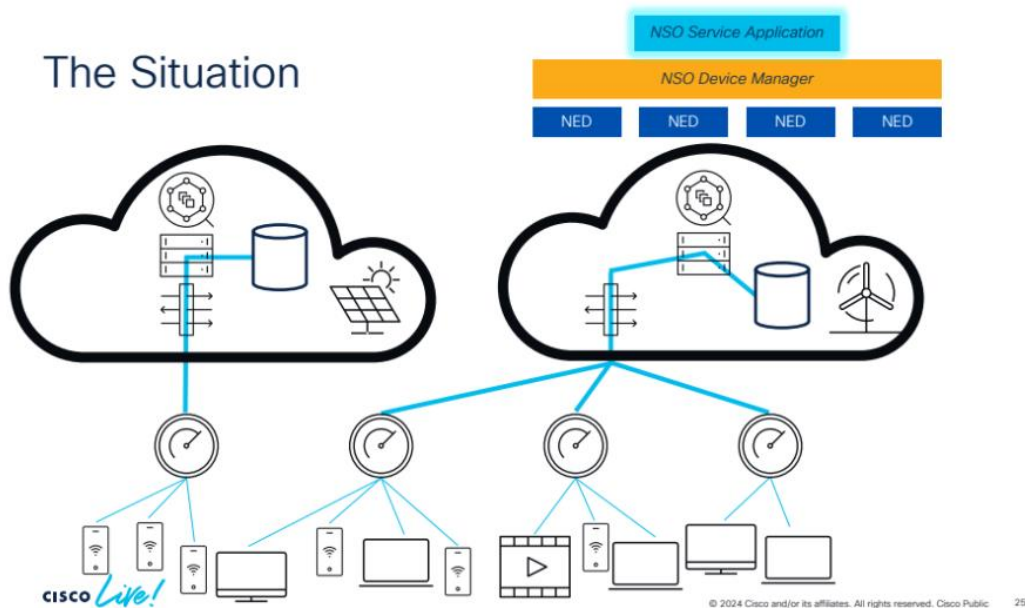
© 2024 Cisco and/or its affiliates. All rights reserved. Cisco Public

In this lab we are going on a journey, starting at a pre-existing NSO level 3 service. The first stage of the journey will be to upgrade it to a level 4 implementation by adding service monitoring functionality into the service definition. The second stage takes the service to level 5 by adding a mechanism for constant, on-going optimization of all service instances.

You can read more about the network automation levels in this blog post <https://community.cisco.com/t5/nso-developer-hub-blogs/network-automation-levels/ba-p/4742665>

Video Delivery Service

In this example, we are looking at a video delivery service. The service is implemented by a network of data centers (DCs) and edge devices. Each DC contains an Origin Video Server (this is something we made up for this lab), and a generic firewall. The edge devices are Edge Video Caches (again, this is something we made up for this lab), that are built to work together with the Origin Video Servers. Imagined end-users can then consume their video feeds through the Edge Video Caches.

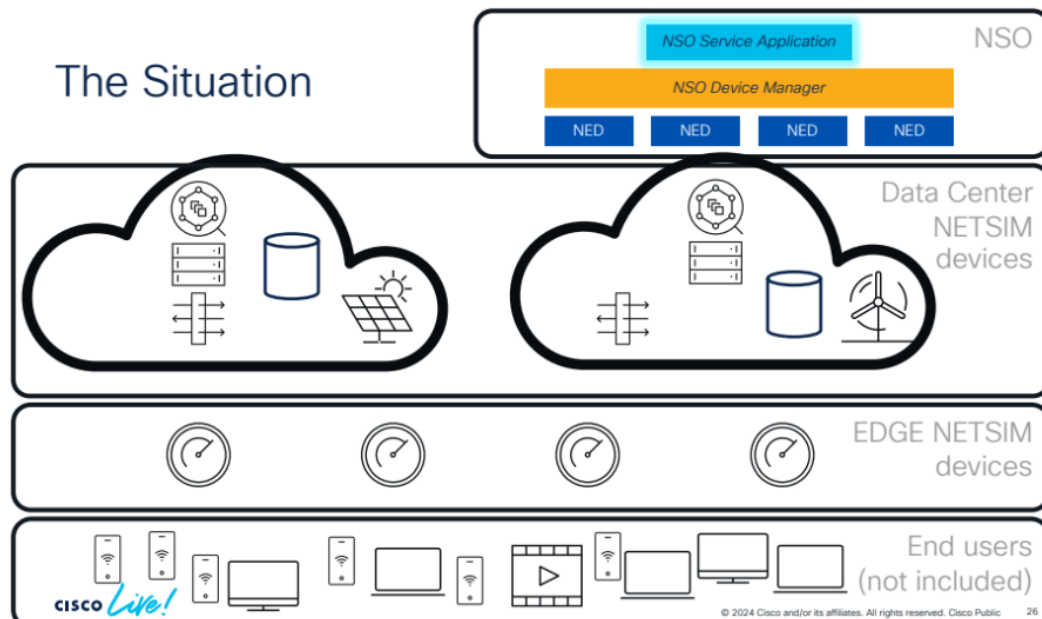


The NSO service we are talking about here controls how the Edge Video Caches are connected to the DCs. Each Edge Video Cache must be connected to exactly one DC, but which one can vary over time, depending on the DC availability, delivered quality, load and the current energy price at the DC location.

At the starting point of this lab, it is the NSO operators that manually decide through configuration which DCs are considered available and which Edge Video Caches are connected to which DCs. While the service at this level automates the device configuration work for the Origin Video Servers and firewalls, the manual decisions/configurations about DC availability and which Edge device should connect to which DC obviously is not automated to a high level. We call this level 3.

Lab Starting Point

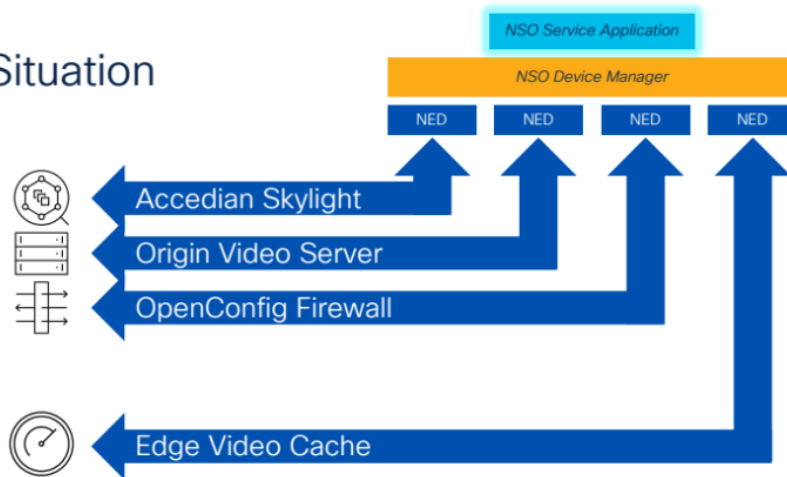
In the lab setup, we have a single NSO instance that controls all the DCs, all the relevant devices in them, and all the Edge devices. All the devices in the DC and the Edge devices are really NSO NETSIM devices. That means they basically only implement the management interfaces as described by their YANG models, but otherwise don't really do anything. We have endowed some of these NETSIMs with a little bit of additional behavior, though, so that they react and respond to a few things. Such functionality will be described later, as needed.



Also, logically part of each DC is a Provider Connectivity Assurance (PCA) monitoring system or Accedian Skylight when referred to as its older name. In our setup, there is only one instance of a skylight system, or “device” as NSO tends to think of all managed systems as devices. A single skylight is enough since we think of it as running “in the cloud” outside our DCs.

When we start, the NSO system is already up and running with the level 3 service implementation, and there are Network Element Drivers (NEDs) for all device types installed, and the devices are already listed in the NSO device list.

The Situation



TECOPS-2665

© 2024 Cisco and/or its affiliates. All rights reserved. Cisco Public 24

Prep 2: Getting into to the Lab Environment

There are multiple ways to access this lab. If you have access to an NSO development environment of recent date, you could clone the example code's git repository and run there. This lab is not very resource demanding, so an average laptop will suffice just fine.

The lab source code is available on GitHub

<https://github.com/nso-developer/nso-automation-levels-example>

Another option is to use the Cisco Cloud IDE where you can log in and connect to a VS Code development environment and NSO server "in the cloud". Simply press the "Cisco Cloud IDE Run It!" button in the GitHub repository README.md.

You can also find this lab by browsing Cisco's Code Exchange catalog. Search for "*nso automation levels*".

<https://devnet.cisco.com/codeexchange>

Each method is described in some more detail below.

Cloning and Running Locally

This lab has been created to run in the Cisco Cloud IDE, but if you have access to an NSO 6.4.1 local installation with relevant development tools, you could also run the lab locally on your machine (MacOS or Linux). You would just need to clone the public git repository.

```
$ git clone https://gitlab.com/nso-developer/nso-automation-levels-example.git
```

Using the Cisco Cloud IDE

Note that there is a two-hour limit on using the Cisco Cloud IDE, when the limit is reached the session resets itself!

Go to the lab's GitHub repository and press the button at the top labeled "Cisco Cloud IDE Run It!"

<https://github.com/nso-developer/nso-automation-levels-example>



Using the Cisco Cloud IDE development environment

Simply login to the Cisco Cloud IDE and a VS Code instance is brought up in the browser. A terminal is available in the lower pane, this is where we will run our commands.

How to persist your changes in Cisco Cloud IDE development environment

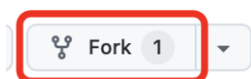
If you want to save your changes you have two options, otherwise please go on to Prep 3.

The first option is to simply copy the contents of the file you want persist and paste it into an editor running locally in your laptop.

The second is to make a fork of the lab's GITHUB repository and use that as the remote URL when pushing your changes. *NOTE!* This requires you to have a user account on GITHUB.

Open <https://github.com/nso-developer/nso-automation-levels-example> in a separate tab in your browser and login to GITHUB.

Click on fork in the upper right corner of the screen:



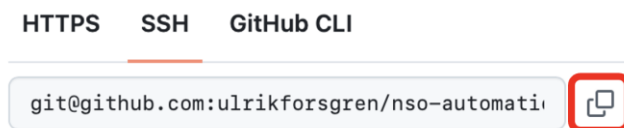
Click on Create Fork:



When in you fork click on Code:



Copy the URL:



Open a terminal in the Cisco Cloud IDE and go to the repository:

```
developer:src > cd nso-automation-levels-example  
developer:nso-automation-levels-example >
```

Change the remote URL with the command below and paste in the copied URL:

```
developer:src > cd nso-automation-levels-example  
developer:nso-automation-levels-example > git remote set-url origin  
https://github.com/ulrikforsgren/nso-automation-levels-example.git  
developer:nso-automation-levels-example >
```

Update the username and email:

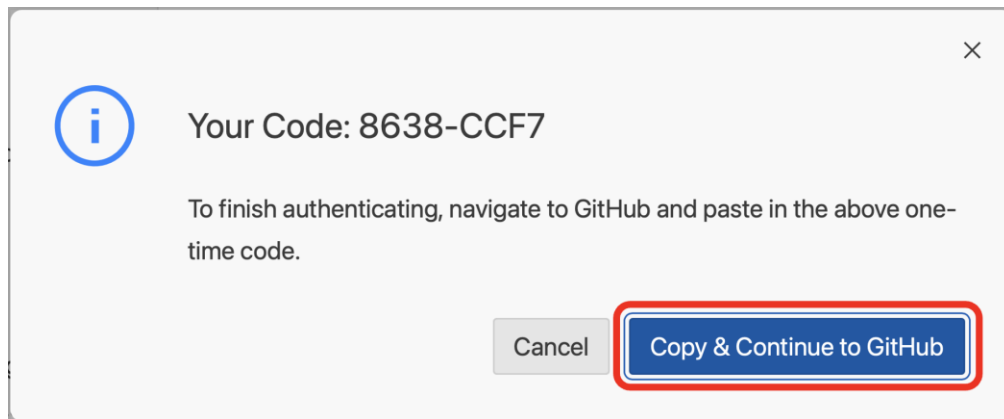
```
developer:nso-automation-levels-example > git config --global user.email "uforsgre@cisco.com"  
developer:nso-automation-levels-example > git config --global user.name "Ulrik Stridsman"  
developer:nso-automation-levels-example > git pull
```

To persist your changes you need to push them. The first time this is done, you'll need to answer a few questions.

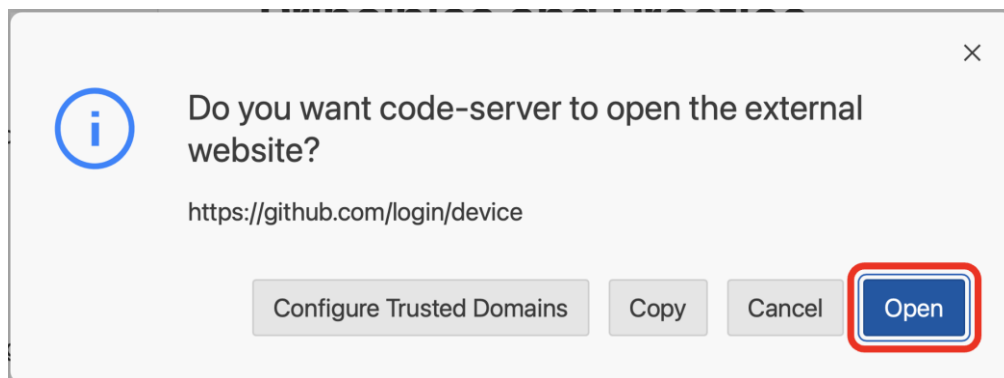
Sign-in to GITHUB



Copy the code and continue



Click Open




Sign to GITHUB when asked:



Paste in the previously copied code and click Continue.

Device Activation

 Signed in as **ulrikforsgren**

Enter the code displayed on your device

F

9

3

0

-

3

7

0

2

Continue

GitHub staff will never ask you to enter your code on this page.


And finally click authorize.

Cancel

Authorize Visual-Studio-Code

Requested from San Jose 3.101.70.146 on February 2nd, 2025 at 21:01 (CET)

Done.



Congratulations, you're all set!
Your device is now connected.

Prep 3: Building and Running the Level 3 System

Step P.1: Working in VS Code

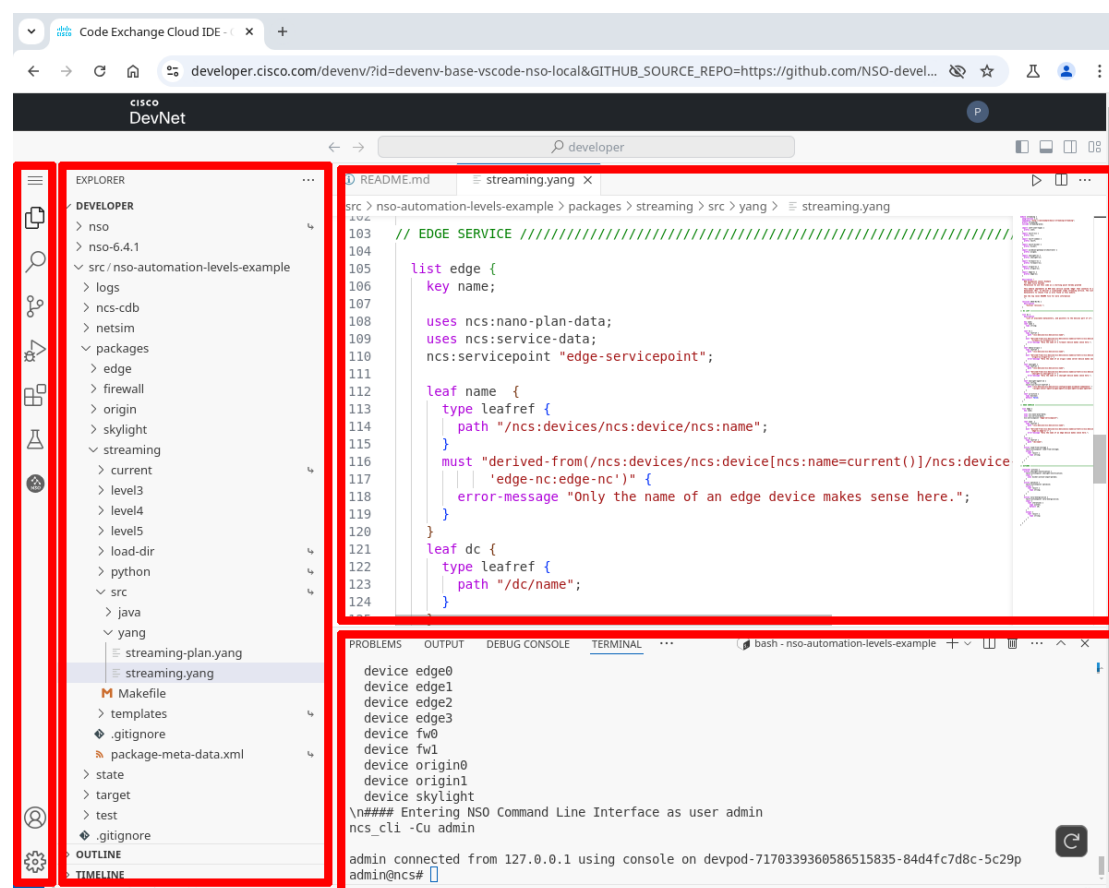
Our project is already open in VS Code, when you launch it.

Once inside the VS Code, you will find the screen divided into four main areas. On the very left we have a view selection bar. The top icon brings you to the project explorer. That is probably the most useful view most of the time. Under it, the search view is useful to search for (and possibly replace) strings across the entire project.

Next to the view selector, the explorer view shows all the files in the project and makes it easy to jump between them.

On the top right, we have the editor, where you can do your development work.

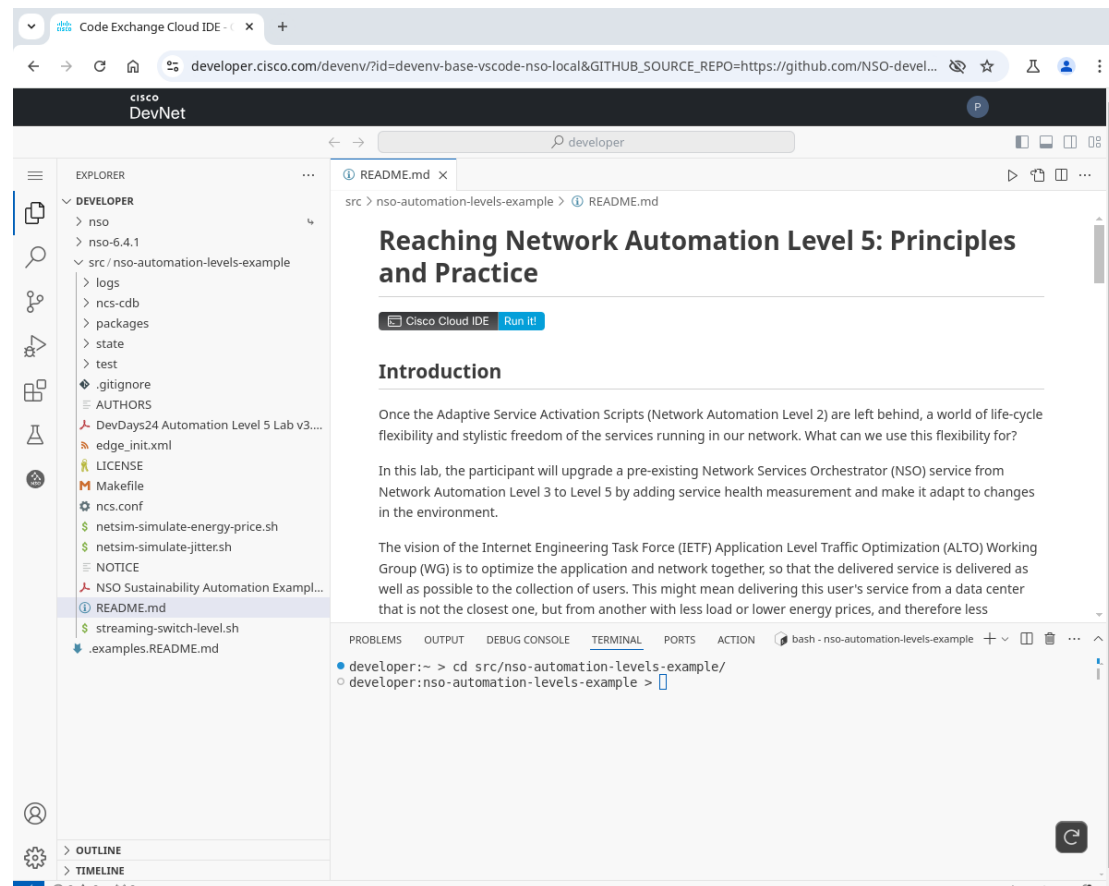
On the lower right you have the terminal windows. The drop down allows you to select between multiple terminals.



Step P.3: Finding the Linux Command Prompt

You can create as many bash terminal windows as you'd like using the plus icon, and then jump between them on the right side of the terminal window.

The lab resides in the `src/nso-automation-levels-example` directory.



Step P.4: Selecting Implementation and Building the System

This lab has a top level Makefile that takes care of building and sets up pretty much everything required to run this project. It even goes so far as to start up the NSO Command Line Interface (CLI) if all the build steps are successful.

At the core of the project, you will find the “streaming” NSO service. This project contains three different implementations of the streaming service, referred to as level3, level4, and level5. The expected starting point for the lab is level3, but you can switch back and forth between levels any time.

Before your first build, you will need to set the project to use the level3 implementation.

```
$ ./streaming-switch-level.sh

**** No streaming service implementation is currently selected. Build will fail.
Please select an implementation, e.g.:
./streaming-switch-level.sh level3

Available streaming service implementations are:
level3
level4
level5
$ ./streaming-switch-level.sh level3
Set streaming service implementation to:
level3
```

You may edit the code in any of these level<X>-directories directly. You may also copy any of the levels into your own directory (but you need to keep that copy in the same package directory). This may be useful if you want to keep the original implementations unchanged, so that you can easily compare your solution with any of them.

OPTIONAL:

```
$ cp -r packages/streaming/level3 packages/streaming/myway
$ ./streaming-switch-level.sh myway
```

Once you have selected which streaming service implementation to use, you can build the project using make.

```
$ make clean all
...
```

If everything goes well, the build ends with a prompt inside the NSO Command Line Interface (CLI). In this case, all NSO NED packages and the streaming service package have been built. An NSO NETSIM network created and started. All devices synced.

```
...
#### Entering NSO Command Line Interface as user admin
nsc_cli -Cu admin

admin connected from 127.0.0.1 using console on ...
admin@nsc#
```


You can now work with the devices and services in the NSO system.

Step P.5: Getting to the NSO Command Prompt

If you dropped out of the NSO CLI and want to get back in from the Linux command line, there are a couple of ways. If your build is working fine, just running make will drop you in there soon enough:

```
$ make all
...
admin connected from 127.0.0.1 using console on ...
admin@ncs#
```

If the build is stopping mid-way, and you still want to get into the NSO CLI, you can also just launch the NSO CLI binary.

```
$ ncs_cli -Cu admin

User admin last logged in ... using cli-console
admin connected from 127.0.0.1 using console on ...
admin@ncs#
```

To drop back out to the Linux command line (or leave any NSO CLI sub mode), use the exit command (or press CTRL+D).

```
admin@ncs# exit
$
```

Step P.6: Updating the Project

As you are making changes to the source packages in the project, NSO will **not** automatically pick them up. That's a feature, not a bug! Updating NSO packages' behavior should only happen when the operator explicitly requests so, and not because some file happened to change.

When using the `./streaming-switch-level.sh` command to change which packages are being used, it may be good to do a full clean rebuild of the project. The way the Makefile has been set up, this will also remove all data from the NSO database etc. That would not be normal in a production environment, but makes sense in this lab.

```
$ make clean all
```

If you have not changed the set of packages, but did change the contents of one or more YANG file, an incremental rebuild is what you want. This will not change

the set of packages, and the NSO database contents will be kept. Once the build completes, you will need to perform a *packages-reload* command to make the change take effect, and it's a good habit to always check the packages' operational status after a reload. If there isn't an *oper-status up* after the package name, you have some debugging to do.

```
$ make all
...
User admin last logged in ... using cli-console
admin connected from 127.0.0.1 using console on ...
admin@ncs# packages reload
...
reload-result {
  package streaming
  result true
}
admin@ncs# show packages package oper-status
packages package edge-nc-1.0
  oper-status up
packages package firewall-nc-1.0
  oper-status up
packages package origin-nc-1.0
  oper-status up
packages package edge-nc-1.0
  oper-status up
packages package streaming
  oper-status up
admin@ncs#
```

For changes to files that do not require compilation, that is python and template files, all you need to do in order for the change to take effect is to *redeploy* the updated package.

```
admin@ncs# packages packages streaming redeploy
redeploy-result {
  result true
}
admin@ncs#
```

Step P.7: Configuring a Service Instance

In NSO our service streaming service is called “edge” in the YANG tree. To add edge service instances, you first must go to configuration mode.

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)#
```

To then create an edge service instance, you type edge and a name for the service instance. Since the streaming service instances are tied 1:1 to the edge video caching devices, the name of the service instance has been locked down to only allow names of edge devices. As the system comes prepared, there are four edge devices, edge0, edge1, edge2, and edge3.

Pressing the TAB-key (→) at any point completes the command or provides possible continuations.

```

admin@ncs(config)# edge edge0 →
Possible completions:
  check-sync      Check if device configuration is according to the service
  dc
  deep-check-sync  Check if device configuration is according to the service
  get-modifications Get the data this service created
  load-from-storage
  log
  plan
  re-deploy        Run/Dry-run the service logic again
  reactive-re-deploy Reactive re-deploy of service logic
  touch            Mark the service as changed
  un-deploy        Undo the effects of the service
  <cr>
admin@ncs(config)# edge edge0

```

In the initial (level3) version of the streaming service, you have to manually specify which datacenter the edge service should be connected to. Out of the box this lab has two datacenters, called dc0 and dc1.

```

admin@ncs(config)# edge edge0 dc →→
Possible completions:
  dc0 dc1
admin@ncs(config)# edge edge0 dc dc0
admin@ncs(config-edge-edge0)#

```

To review changes, you can use the show config command (at the top level to see all config changes).

```

admin@ncs(config-edge-edge0)# top
admin@ncs(config)# show config
edge edge0
dc dc0
!
admin@ncs(config)#

```

To make these pending changes take effect, they need to be committed.

Before we do that, let's take a quick look at the current configuration on the origin0 video server. This is just to show that currently the content catalogue is empty.

```

admin@ncs(config)# show full-configuration devices device origin0 config origin content
% No entries found.
admin@ncs(config)#

```

Now we can commit our edge service instance. We will look at the content catalogue again shortly.

```

admin@ncs(config)# commit
Commit complete.

```


Step P.8: Looking at Operational Data and the Plan

Operational data is the data you can look at, but not change. At least not change directly. The streaming service has some interesting operational data you may want to observe. You can observe this data while you are still in configuration mode, but then you have to put `do` in front of your show command, to indicate that you want to run an operational mode command.

```
admin@ncs(config)# do show edge edge0 plan | tab
```

TYPE	NAME	BACK TRACK	GOAL	STATE	STATUS	WHEN	ref	POST ACTION STATUS
self	self	false	-	init	reached	2025-01-21T16:30:13	-	-
				ready	not-reached	-	-	-
dc	dc	false	-	init	reached	2025-01-21T16:30:13	-	-
				skylight-configured	reached	2025-01-21T16:30:13	-	-
				fw-configured	not-reached	-	-	-
				ready	not-reached	-	-	-

```
admin@ncs(config)#
```

At this point in the lab, pretty much all you see is the service's plan. Each NSO nano-service has an associated plan that shows how far the service instance has come in its realization.

As you can see here, this service instance has reached its init state, but is not ready. The datacenter component seems to be waiting on the state `fw-configured`. Looking in `streaming-plan.yang` reveals that this stage has a pre-condition associated. It is waiting for the operator to configure the datacenter leaf `is-active` with value `true`.

```
admin@ncs(config)# dc dc0 is-active true
admin@ncs(config-dc-dc0)# commit
Commit complete.
admin@ncs(config-dc-dc0)#
System message at 2025-01-21 17:43:04...
Commit performed by admin via console using cli.
admin@ncs(config-dc-dc0)#
System message at 2025-01-21 17:43:12...
Commit performed by admin via system using cli.
admin@ncs(config-dc-dc0)#
System message at 2025-01-21 17:43:12...
Commit performed by admin via system using cli.
admin@ncs(config-dc-dc0)#
System message at 2025-01-21 17:43:12...
Commit performed by admin via console using cli.
admin@ncs(config-dc-dc0)#
```

The commit command changed the datacenter `dc0` to `is-active`. Moments later, there was a second commit performed by the system itself. This is called a re-deploy. It was triggered by the pre-condition for state `fw-configured` now being true. Seconds later a few more commits happened spontaneously. Other stages of the service plan got into action. Dropping out to operational mode to have a look.

```

admin@ncs(config-dc-dc0)# exit
admin@ncs(config)# exit
admin@ncs# show edge edge0 plan | tab

```

TYPE	NAME	TRACK	GOAL	STATE	STATUS	WHEN	ref	POST ACTION STATUS
self	self	false	-	init	reached	2025-01-21T16:30:13	-	-
				ready	reached	2025-01-21T16:43:04	-	-
dc	dc	false	-	init	reached	2025-01-21T16:30:13	-	-
				skylight-configured	reached	2025-01-21T16:30:13	-	-
				fw-configured	reached	2025-01-21T16:43:04	-	-
				ready	reached	2025-01-21T16:43:04	-	-
edge	edge	false	-	init	reached	2025-01-21T16:43:04	-	-
				connected-to-dc	reached	2025-01-21T16:43:04	-	create-reached
				connected-to-skylight	reached	2025-01-21T16:43:04	-	-
				ready	reached	2025-01-21T16:43:04	-	-

```

admin@ncs#

```

All the service plan states have now been reached. Let's check the Origin Video Server catalogue again.

```

admin@ncs# show running-config devices device origin0 config origin content
devices device origin0
config
  origin content "Blade Runner"
  !
  origin content "Die Hard"
  !
  origin content "Pulp Fiction"
  !
  origin content "The Dark Knight"
  !
  !
!
admin@ncs#

```

The service creation has apparently resulted in some device-level configuration changes.

Step P.9: Turning off Commit Messages

If at any point in this lab, you grow tired of the automatic commit messages popping up as soon as something is being committed by the system (or a fellow operator, if you have any), you can turn them off.

```

System message at 2024-01-23 21:46:57...
Commit performed by admin via system using cli.
admin@ncs(config)#
System message at 2024-01-23 21:46:57...
Commit performed by admin via system using cli.
admin@ncs(config)#

```

Open up the ncs.conf NSO configuration file in the project root directory, and look up the commit-message setting.

```
<commit-message>true</commit-message>
```

Change it to false, then save the file.

OPTIONAL:

```
<commit-message>false</commit-message>
```

In order for the change to take effect, you also need to give NSO a reload command (or simply restart it with make clean all) on the Linux command line.

```
$ ncs --reload  
$
```

The commit messages were left on by default so that you would see when things are happening.

Task 1 Overview: Go to Level 4, Add Monitoring

The first task in this lab is to extend the existing level3 streaming service so that every service instance created also is monitored, rendering it a level4 service. For monitoring, we are using Accedian Skylight (Provider Connectivity Assurance, PCA). To do that we have a number of changes in mind.

Here follows a number of steps to implement the level4 service. First each step is described on an overview level. After that follows the same steps one more time, but now on a very detailed level with specific implementation proposals. If you already have a good idea about how to implement the step as described in the overview, you don't need to read the detailed description for that step. If you aren't sure what the overview instructions are trying to say, or would like to verify that your solution is on the right track, you will find one of many possible implementations in the detailed section.

Overview 1.1 Replace `dc/is-active` with `dc/oper-status/jitter`

The `is-active` leaf allows an operator to declare by configuration when s/he judges that a particular DC is ready to be used. This sort of manual health judgement does not feel very automated at all, and needs to go. Instead we would like Skylight to monitor and report the health of each DC, and only advice services to deploy on a particular DC if its health scores are good. In this lab, we're letting jitter be a proxy for health score, in order not to overcomplicate matters.

Note that the `streaming-plan.yang` currently refers to `dc/is-active`, so you will need to update the plan too.

To allow us to track the jitter values, we need to add an operational leaf inside the list of DCs. Operational data is often kept in a container called `oper-status` to make it super obvious that it's not configuration. Here is a suggested formulation for the jitter leaf.

```
container oper-status {
  config false;
  tailf:info "The actual operational state of the DC.";
  tailf:cdb-oper {
    tailf:persistent true;
  }
  leaf jitter {
    description "The latest reported jitter value for this DC.";
    type decimal64 {
      fraction-digits 3;
    }
    units "ms";
  }
}
```



```
}  
}
```

Overview 1.2 Replace edge/dc with edge/oper-status/chosen-dc

Level3 edge service instances are configured to use a particular dc. This does not feel very automated at all, and on level4, this needs to go. Instead we would like each edge service instance to automatically pick the DC with the best jitter (health score).

Since operators and code in the later stages in the service deployment process will want to know which DC was selected, an operational leaf (a leaf that is not configurable by the operator) in the YANG module should be introduced instead. We suggest also changing the name of the leaf, so that the change is more visible and clear. We recommend placing the operational data in the service under a container called oper-status. Since the data should survive a restart of NSO, it makes sense to mark it as persistent. You could use a YANG structure like this.

```
container oper-status {  
  config false;  
  tailf:info "The actual operational state of the service.";  
  tailf:cdb-oper {  
    tailf:persistent true;  
  }  
  leaf chosen-dc {  
    ...  
  }  
}
```

There are a few templates and one reference in Python to leaf dc that need to be updated to reflect the new location and name.

- python/streaming/main.py (in class LoadFromStoragePostAction)
- templates/edge-servicepoint-dc-fw-configured.xml
- templates/edge-servicepoint-edge-connected-to-dc.xml
- templates/edge-servicepoint-edge-connected-to-skylight.xml

Overview 1.3 Write a DC Selection Function in Python

Since the edge service can no longer rely on operator input when figuring out which DC to connect to, we need to add Python code that makes this choice. We also need to ensure it is invoked at the right time in the nano-service plan, so that coming plan stages that depend on this decision have access to the result of the decision. The decision also needs to be published in such a way that later stages (and operators) can see which DC the service instance decided to use.

The boiler-plate code to insert in main.py in order to handle a particular state might look something like the below. Then add your own decision logic, and

update chosen-dc with the result. Replace ComponentStateName with something that reflects which component and state this code pertains to, for your own code navigation.

```
class ComponentStateName(NanoService):
    @NanoService.create
    def cb_nano_create(self, tctx, root, service, plan, component, state, proplist,
                      compproplist):
        self.log.info(f'cb_nano_create: ComponentStateName for {service.name}')
```

In order for the above state handling code to be called, it also need to be registered in the service's setup method. You do it like this, but remember to replace ComponentName, StateName with the component and states you have chosen that this code should be invoked in, and replace ComponentStateName with the same name you picked for your Python class above.

```
def setup(self):
    ...

    self.register_nano_service(servicepoint='edge-servicepoint',
                              componenttype="streaming:ComponentName",
                              state="ncs:StateName",
                              nano_service_cls=ComponentStateName)
```

Overview 1.4 Configure Skylight

The Skylight device needs to be updated with a monitoring session for each edge service instance. Luckily for us, Skylight experts have provided a template with some variables for us to use. You will find this template in `edge-servicepoint-edge-connected-to-skylight.xml`

This Skylight configuration template sets up a Two-Way Active Measurement Protocol (TWAMP) session between a DC and an Edge device. In addition, the edge device needs to have its TWAMP-reflector function enabled. This is done by adding the following piece of configuration template to the edge device:

```
<devices xmlns="http://tail-f.com/ns/ncs">
  <device>
    <name>{$EDGE}</name>
    <config>
      <twamp xmlns="urn:ietf:params:xml:ns:yang:ietf-twamp">
        <session-reflector>
          <admin-state>true</admin-state>
        </session-reflector>
      </twamp>
    </config>
  </device>
</devices>
```

Overview 1.5 React to Skylight Notifications

Finally, we need to do something to remedy the situation when Skylight reports changes in jitter (health score). This certainly applies if the jitter (health score) would sky-rocket, but may be equally relevant if the jitter improves markedly.

Overview 1.6 Build and Deploy

As your service changes develop towards a fully-fledged level4 solution, you will likely need to rebuild and re-deploy the service multiple times. If you have changed any YANG module contents, you will need to rebuild on the Linux command line as described previously.

If you only changed Python or template files, packages reload is sufficient to bring in the changes.

Overview 1.7 Test

Something like following behavior is expected once the solution is complete, but it is certainly up to you to decide how closely to recreate this behavior.

First, let's see that we have a good initial state. Packages loaded without error, there is a jitter value per dc, there are no edge services at this point.

```
admin@ncs# show packages package oper-status | tab
```

NAME	UP	PROGRAM CODE ERROR	JAVA UNINITIALIZED	PYTHON UNINITIALIZED	...	PACKAGE META DATA ERROR	FILE LOAD ERROR	ERROR INFO	WARNINGS
edge-nc-1.0	X	-	-	-	...	-	-	-	-
firewall-nc-1.0	X	-	-	-	...	-	-	-	-
origin-nc-1.0	X	-	-	-	...	-	-	-	-
skylight-nc-3.0	X	-	-	-	...	-	-	-	-
streaming	X	-	-	-	...	-	-	-	-

```
admin@ncs# show dc | tab
```

NAME	JITTER
dc0	20.0
dc1	25.0

```
admin@ncs# show edge
% No entries found.
admin@ncs#
```

Then let's create an edge service instance, and *dry-run* it to see what it would do with the devices in the network. The output should certainly mention all the devices, edge0, fw0, origin0 and skylight. Note that we abbreviated the fw0 output below with an ellipsis (...), since it is quite long.

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# edge edge0
admin@ncs(config-edge-edge0)# commit dry-run
cli {
  local-node {
    data devices {
      device edge0 {
        config {
          edge {
            + origin-server 127.0.0.1;
          }
        }
      }
    }
  }
}
```

```

        twamp {
            session-reflector {
                + admin-state true;
            }
        }
    }
}
device fw0 {
    config {
        acl {
            acl-sets {
                + acl-set edge0 ACL_IPV4 {
                    + config {
                        + name edge0;
                        + type ACL_IPV4;
                    }
                    + acl-entries {
                        ...
                    }
                }
            }
        }
    }
}
device origin0 {
    config {
        origin {
            + edge 127.0.0.1 {
            }
            + content "Blade Runner" {
            }
            + content "Die Hard" {
            }
            + content "Pulp Fiction" {
            }
            + content "The Dark Knight" {
            }
        }
    }
}
device skylight {
    config {
        sessions {
            + session bd05cfaa-d30d-5560-8f4b-5b88f1e9a0fc agent {
                + sessiontype twamp;
                + ifStateful true;
                + agent {
                    + agentId a31e3e45-ec69-4f4a-a479-b2273f8077be;
                    + agentSessionName dc0-agent-to-edge0-twamp;
                    + enable true;
                    + period continuous;
                }
                + twamp {
                    + senderDscp 0;
                    + reflectorAddr 127.0.0.1;
                    + reflectorPort 4000;
                    + reportInterval 31;
                }
            }
        }
    }
}
+edge edge0 {
+}
}
}
admin@ncs(config-edge-edge0)# commit
Commit complete.
admin@ncs(config-edge-edge0)#

```

There are debug options possible for templates and xpath, either one at a time or both simultaneously

```
admin@ncs(config-edge-edge0)# commit dry-run | debug template
...
admin@ncs(config-edge-edge0)# commit dry-run | debug xpath
...
admin@ncs(config-edge-edge0)# commit dry-run | debug template | debug xpath
...
```

At this point, the service creation happens, and the configuration changes are sent to all the devices. Since this is a nano-service with multiple stages, the services will re-deploy themselves when a condition they have been waiting for has been fulfilled. We can then show the nano-service plan state for the edge0 service instance.

```
admin@ncs(config-edge-edge0)#
System message at 2024-01-29 16:37:58...
Commit performed by admin via system using cli.
admin@ncs(config-edge-edge0)#
System message at 2024-01-29 16:37:58...
Commit performed by admin via system using cli.
admin@ncs(config-edge-edge0)#
System message at 2024-01-29 16:37:58...
Commit performed by admin via console using cli.
admin@ncs(config-edge-edge0)# exit
admin@ncs(config)# exit
admin@ncs# show edge edge0 plan | tab
```

TYPE	NAME	TRACK	GOAL	STATE	STATUS	WHEN	ref	POST ACTION
self	self	false	-	init	reached	2025-01-23T15:30:59	-	-
				ready	reached	2025-01-23T15:30:59	-	-
dc	dc	false	-	init	reached	2025-01-23T15:30:59	-	-
				skylight-configured	reached	2025-01-23T15:30:59	-	-
				fw-configured	reached	2025-01-23T15:30:59	-	-
				ready	reached	2025-01-23T15:30:59	-	-
edge	edge	false	-	init	reached	2025-01-23T15:30:59	-	-
				connected-to-dc	reached	2025-01-23T15:30:59	-	create-reached
				connected-to-skylight	reached	2025-01-23T15:30:59	-	-
				ready	reached	2025-01-23T15:30:59	-	-

```
admin@ncs#admin@ncs# show edge edge 0 oper-status
oper-status chosen-dc dc0
admin@ncs#admin@ncs#
```

Great, the service creation seems to work. Now, let's change the jitter value for dc1 and create a few more service instances. The long command under devices device skylight reaches out to the skylight device and requests that it sends a notification update about a measured jitter value. Since we have no real devices and no traffic in the network, we use this manual setup to simulate what a Skylight device might have sent.

```
admin@ncs# show dc | tab
NAME JITTER
```

```

-----
dc0 20.0
dc1 25.0

admin@ncs# devices device skylight rpc rpc-send-notification-low send-notification-low device
dc1 type jitter
admin@ncs# show dc | tab
NAME JITTER
-----
dc0 20.0
dc1 1.198

admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# edge edge1
admin@ncs(config-edge-edge1)# edge edge2
admin@ncs(config-edge-edge2)# edge edge3
admin@ncs(config-edge-edge3)# commit
Commit complete.
admin@ncs(config-edge-edge3)#

```

At this point, three additional service instances are created. They will naturally pick the dc with the best jitter value. As these service instances progress in their nano-service plan, there will be multiple system messages like this.

```

System message at 2024-01-29 16:40:43...
Commit performed by admin via system using cli.
admin@ncs(config-edge-edge3)#

```

When the service creations are completed, we can have a look at the edge services' operational status. All the new service instances have picked dc1. The previous service instance edge0 remains with dc0.

```

admin@ncs(config-edge-edge3)# exit
admin@ncs(config)# exit
admin@ncs# show edge oper-status | tab
CHOSEN
NAME DC
-----
edge0 dc0
edge1 dc1
edge2 dc1
edge3 dc1

admin@ncs#

```

If we simulate that dc1 suddenly gets a much greater jitter value by sending another Skylight notification, the service instances on dc1 will immediately switch to dc0.

```

admin@ncs# devices device skylight rpc rpc-send-notification-high send-notification-high
device dc1 jitter 45.2
admin@ncs#
System message at 2024-01-29 16:45:14...
Commit performed by admin via console using cli.
admin@ncs#
System message at 2024-01-29 16:45:14...
Commit performed by admin via console using cli.
admin@ncs#
System message at 2024-01-29 16:45:14...
Commit performed by admin via console using cli.
admin@ncs# show dc | tab
NAME JITTER
-----
dc0 20.0
dc1 45.2

admin@ncs# show edge oper-status | tab
CHOSEN
NAME DC
-----
edge0 dc0
edge1 dc0
edge2 dc0
edge3 dc0

admin@ncs#

```

Task 1 Detailed Instructions

The steps that follow here are detailed implementation proposals for each one of the overview points discussed in the previous chapter.

Step 1.1 Replace `dc/is-active` with `dc/oper-status/jitter`

In `streaming.yang`, look up leaf `is-active`. Remove it. Remember to remake the package, and issue a `packages-reload` command in NSO afterwards to make the changes take effect.

Remove this from `streaming.yang`:

```
leaf is-active {  
  type boolean;  
  default false;  
}
```

In `streaming-plan.yang`, there is a reference to `is-active`. This blocks edge service instances from using a DC that is not active. Since we will update the selection logic, we won't be needing this block any more. Remove it.

Remove this from `streaming-plan.yang`:

```
ncs:pre-condition {  
  ncs:monitor  
    "/streaming:dc[name = $SERVICE/dc]" {  
      ncs:trigger-expr "is-active = 'true'";  
    }  
}
```

To provide a leaf location for the jitter value for each DC, add this to `streaming.yang` inside, e.g. at the end of the list `dc { ... }`, where the leaf `is-active` was.

```
container oper-status {  
  config false;  
  tailf:info "The actual operational state of the DC.";
```



```

tailf:cdb-oper {
    tailf:persistent true;
}

leaf jitter {
    description "The latest reported jitter value for this DC.";
    type decimal64 {
        fraction-digits 3;
    }
    units "ms";
}
}

```

Step 1.2 Replace edge/dc with edge/oper-status/chosen-dc

Removing a configurable leaf and replacing with an operational leaf is pretty easy,

Replace this in streaming.yang:

```

leaf dc {
    type leafref {
        path "/dc/name";
    }
}

```

... with this:

```

container oper-status {
    config false;
    tailf:info "The actual operational state of the service.";
    tailf:cdb-oper {
        tailf:persistent true;
    }
    leaf chosen-dc {
        type leafref {
            path "/dc/name";
        }
    }
}

```

Because of this change, we have to update several references to edge/dc to go to edge/oper-status/chosen-dc instead. Three such references are in templates, and one in Python.

Replace this in `edge-servicepoint-edge-connected-to-skylight.xml`, in `edge-servicepoint-dc-fw-configured.xml` and in `edge-servicepoint-edge-connected-to-dc.xml`:

```
<?set DC = {./dc}?>
```

... with this:

```
<?set DC = {./oper-status/chosen-dc}?>
```

You will find the reference from Python in class `LoadFromStoragePostAction`. This action is being called in the service's nano-plan. This action tells the Origin video server to update its content so that it has all the content currently being cached in the Edge video caching devices. Obviously, that command needs to be sent to the right Origin video server; the one in the selected DC. The change is needed due to us renaming leaf `dc` to leaf `oper-status/chosen-dc`.

Replace this in `main.py`:

```
origin_name = root.dc[service.dc].media_origin
```

... with this:

```
origin_name = root.dc[service.oper_status.chosen_dc].media_origin
```

Step 1.3 Write a DC Selection Function in Python

The DC selection needs to happen very early in the deployment of the edge service, since most other things depend on it. This selection does not depend on anything else being decided in the service instance, so it can very well go first in the DC component nano-plan.

The dc component already has an “init” state that does nothing. We can change that so that it calls a Python method that makes the DC selection, and then writes the decision to the chosen-dc operational leaf.

Replace this in streaming-plan.yang:

```
ncs:component-type "dc" {  
    ncs:state "ncs:init";  
}
```

... with this:

```
ncs:component-type "dc" {  
    ncs:state "ncs:init" {  
        ncs:create {  
            ncs:nano-callback;  
        }  
    }  
}
```

Since we chose to implement the DC selection in the DC component’s init state, a reasonable class name might be DCInit. This class should have a cb_nano_create method, which will be invoked by NSO when getting to the state in the plan, which will be immediately at service creations with the nano-plan above.

This function should now figure out which DC to use, and write the name of the chosen one to service.oper_status.chosen_dc. As selection logic here, we are just looping over all the DCs and keeping track of which one has the lowest jitter. Yes, if you are a Python wizard you could do this in more or less a single line, but here we prefer to keep to the basics.

```
class DCInit(NanoService):  
    @NanoService.create  
    def cb_nano_create(self, tctx, root, service, plan, component, state, proplist,  
                        compproplist):  
        self.log.info(f'cb_nano_create: DCInit for {service.name}')  
  
        # Find the DC with the lowest jitter  
        best_jitter = 100000  
        best_dc = None  
  
        for dc in root.dc:  
            if dc.oper_status.jitter is None:  
                self.log.info(f'Checking DC {dc.name}: DC is not ready')  
                continue # Data not available yet, disregard this option  
  
            dc_jitter = float(dc.oper_status.jitter)  
            self.log.info(f'Checking DC {dc.name}: jitter {dc_jitter}')  
  
            if dc_jitter < best_jitter:  
                best_jitter = dc_jitter  
                best_dc = dc
```

```

if best_dc is None:
    raise Exception('No DC found')

self.log.info(f'Found DC {best_dc} with the lowest jitter {best_jitter}')
# Value goes into operational data, usable by templates applied at later stages
service.oper_status.chosen_dc = best_dc.name

```

Then, a bit lower down in main.py, make sure the setup method starts like this (but keep the rest of what is in the setup method too), to have the state code registered/associated with the right component and state:

```

def setup(self):
    # The application class sets up logging for us. It is accessible
    # through 'self.log' and is a ncs.log.Log instance.
    self.log.info('Main RUNNING')

    # Nano service callbacks require a registration for a service point,
    # component, and state, as specified in the corresponding data model
    # and plan outline.
    self.register_nano_service(servicepoint='edge-servicepoint',
                              componenttype="streaming:dc",
                              state="ncs:init",
                              nano_service_cls=DCInit)

```

Step 1.4 Configure Skylight

Services usually have to configure devices, or sometimes lower-level services. This generally happens through templates. The contents of these templates are typically received from experts on the particular devices or services being used. The templates use XML format in order to resolve ambiguity that often arise with other formats. This XML content is usually generated by configuring the device (or lower-level service) manually using CLI or any convenient method, then grabbing the result as NETCONF XML.

The edge-servicepoint-edge-connected-to-skylight.xml template is great when it comes to what needs to be setup on the Skylight device. There are two main problems with it, however, the way it is created. First, the config templates declares a servicepoint, componenttype and state. This means the template will override any Python (or Java) code that implements this state. The code we have in ConnectedToSkylight in main.py isn't getting called, despite being properly registered. The purpose with that code is to create a unique session id, needed by the Skylight configuration. This leads to the second problem, which is that the SESSION_ID variable in the template is hard-coded. The session id would not be unique at all, making all edge service instances overwrite each other's Skylight config.

Fixing these things in the template is easy, just remove a few of the headers, and the Python code will be invoked. The Python code already computes a value for

the SESSION_ID, which will then be picked up as the Python code applies the template.

One more thing, leaf dc changed name to oper-status/chosen-dc. This needs to be updated in the template DC variable as well.

Replace this in edge-servicepoint-edge-connected-to-skylight.xml:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0"
  servicepoint="edge-servicepoint"
  componenttype="streaming:edge"
  state="streaming:connected-to-skylight">
```

... with this:

```
<config-template xmlns="http://tail-f.com/ns/config/1.0">
```

Besides configuring the Skylight device, we also need to configure the Edge device to enable its TWAMP reflector. Maybe this should have been a separate step in the nano-plan for super clarity, but here we chose the slightly more lazy option to just piggy back this in the same template as the skylight setup.

So we are back in edge-servicepoint-edge-connected-to-skylight.xml again, and right after the existing configuration for the skylight device:

```
<devices xmlns="http://tail-f.com/ns/ncs">
  <device>
    <name>skylight</name>
    ...
  </device>
</devices>
```

Insert this:

```
<devices xmlns="http://tail-f.com/ns/ncs">
  <device>
```

```

    <name>{$EDGE}</name>
    <config>
      <twamp xmlns="urn:ietf:params:xml:ns:yang:ietf-twamp">
        <session-reflector>
          <admin-state>true</admin-state>
        </session-reflector>
      </twamp>
    </config>
  </device>
</devices>

```

This will be near the very end of the template, but make sure the `</config-template>` line still goes last.

Step 1.5 React to Skylight Notifications

Monitoring is great, but if we don't handle most situations automatically, we can't say the network is very automated, can we? Of course, we don't want to raise any alarms for situations that the system is resolving on its own, right? Alarms are for getting the attention of operators, and that's not what we are looking for here. So what do we do?

There will obviously be situations that the automated system can't handle well, but if the Skylight indicates that one of our DCs is giving is trouble, such as high jitter, an obvious solution would be to switch to another. Which one? Well, there is already logic in the service for picking the best DC at the time of creation. So how about simply reusing that? In NSO terms, recomputing the best way to express an existing service in light of new operating conditions, but without changing the intent (i.e. service configuration), is called a re-deploy.

So what we want is to automatically re-deploy services when we get notifications about jitter changes from Skylight. Which services? All of them? While re-deploying all services every time a notification about a change comes in might be optimal in one sense, it does not scale all that well. Doing so would make all service instances always and immediately go to the DC with the best jitter. Also, if you have a lot of service instances (e.g. tens of thousands or millions), the sheer re-computation work could be large.

In the design here, we have chosen to re-deploy all services on the particular DC that the notification concerns. This means the services on this DC will migrate if the jitter values get high, but all edge service instances will not always immediately flock to the best DC.

To implement this automatic re-deploy of services, we are using an NSO notification kicker, i.e. a function that will invoke an action when a particular

kind of notification is received. Kickers are easily set up with a bit of configuration. We have chosen to, yet again, place this kicker config snippet in the edge-servicepoint-edge-connected-to-skylight.xml since it is applied at the right time.

Add this to edge-servicepoint-edge-connected-to-skylight.xml . You can add it “anywhere” in the template (order between blocks does not matter), but of course not within the existing blocks. How about putting it after the <devices>...</devices> blocks in there?

```
<kickers xmlns="http://tail-f.com/ns/kicker">
  <notification-kicker xmlns="http://tail-f.com/ns/ncs-kicker">
    <id>skylight-notification-kicker</id>
    <serializer>111</serializer>
    <selector-expr>$SUBSCRIPTION_NAME='skylight-events'</selector-expr>
    <kick-node xmlns:streaming="http://com/example/basic-streaming/streaming">
      /streaming:actions
    </kick-node>
    <action-name>skylight-notification</action-name>
  </notification-kicker>
</kickers>
```

This will be near the very end of the template, but make sure the </config-template> line still goes last.

Step 1.6 Build and Deploy

How to build and deploy has been described earlier in this document, but here follows a brief summary of what you need to do after making the changes above.

Every time you make a change to a YANG module (streaming.yang, streaming-plan.yang) or Java module (none in this project) you need to get to the Linux command line and run make like this:

```
$ make all
```

If all goes well, you will end up on the NSO command line again. If there are compilation or build errors, you will need to correct them, then rerun make as above again.

NSO will not automatically use the new versions of your packages. Upgrading your system is a potentially disruptive operation and should always be done very consciously, using `packages reload`. For changes to templates and Python code, this is sufficient to get those changes into the system. We recommend to always follow up with showing the package status, in order to see more clearly that all packages are running fine.

```
admin@ncs# packages reload
...
admin@ncs# show packages package oper-status
```

Step 1.7 Test

If everything is running fine now, the command sequence described earlier in this chapter under point (see page) should work on your system and produce something similar.

During the development work, there are a couple of log files that may come in handy, to see what is going on in the system.

The most useful one is probably going to be the streaming service log file.

```
$ less logs/ncs-python-vm-streaming.log
```

The development log, and XPath trace log may occasionally also be useful.

```
$ less logs/devel.log
$ less logs/xpath.trace
```


Task 2 Overview: Go to Level 5, Add Ongoing Optimization

In the second task in this lab, we are taking the service to level 5. This means we are adding a more advanced optimization, and in particular, ensuring that optimization is always on. We will even go so far as to stop reacting immediately on incoming notifications. In the real world, you would probably keep reacting immediately when you are notified about really bad conditions, but here the point we are trying to make stands out clearer if we don't.

As with the task 1 work, each step is described first on an overview level, then once more in full detail a bit further down.

Overview 2.1 Add edge-capacity and energy-price

In order to have a more complex (more interesting) decision function that takes into account several different parameters when deciding which DC an edge service should connect to, we'll start by adding two leafs to the DC model. Edge-capacity is a configurable leaf where the operator sets the max number of edge clients that the DC can support. Let's give this leaf a default value of 3.

Energy-price is an operational value in each DC that you can set manually to simulate changes. Want to play things scientifically and specify a unit for the leaf? You can use USD/MWh as the unit. We have also provided an action called `action/vary-energy-price`. If you invoke this action, a background Python thread will go around and adjust the energy-price for a DC every 15 seconds or so. You can turn off the price variations by running the same action again.

```
admin@ncs# actions vary-energy-price
result Started Varying Energy Price
admin@ncs# actions vary-energy-price
result Stopped Varying Energy Price
admin@ncs#
```

The code for this thing is found in `skylight_notification_action.py`, if you are curious, but we don't expect you to make any changes here.

Overview 2.2 Add edge-clients (optional)

Each edge service instance already has a leaf called `oper-status/chosen-dc` that operators can have a look at if they are interested to see which DC an edge service instance is connected to.

As it turns out, it will be interesting to have a look at the DC operational status as price and jitter varies. If the list of edge service instances attached to each DC was listed in the same operational status view, that looks cool. This step is entirely optional and only provides information that is already available in one more view.

If you want to do this, add a new leaf-list in the DC list as oper-status/edge-clients. Add and remove the name of the edge client to this list as it moves around.

You may be interested to know that the Python MAAGIC methods to remove and add elements from a leaf-list are

```
myleaflist.remove(somename)
myleaflist.create(somename)
```

Overview 2.3 Better DC Selection Function

Now that we have a few more parameters to look at besides jitter, namely edge-capacity and energy-price, you should devise an optimization function that selects the best DC by taking all of these into account in some way. We will leave exactly how up to you, but we are very interested in hearing your thoughts about what you chose to do, and also what you would have liked to do if you had more time to implement something grander.

In our opinion, the optimization function should not allow more edge service instances to join a particular DC than specified in edge-capacity for that DC.

Overview 2.4 Optimization doesn't have to be Notification Based

You probably remember from your work on task 1 that when a jitter change notification is received from Skylight, that triggers an invocation of the action class SkylightNotificationAction. This callback updates the DC jitter operational data value, and also calls `reactive_re_deploy()` on the service instances that are connected to the particular DC. It's time to remove the re-deploy part of this code now.

Not that this behavior is bad, but it makes it harder to see what's going on once we get to level5, so for now, let's take it out. In a real system, keeping something that re-deploys services that got into trouble is a good thing, but typically some more thought needs to go into the solution than just immediately re-deploying large number of service instances.

Overview 2.5 Keep Optimizing

Remember that vary-energy-prices action mentioned earlier? We have one more Python background thread that can be started and stopped at will with a similar command.

```
admin@ncs# actions optimize
result Stopped Optimizer
admin@ncs# actions optimize
result Stopped Optimizer
admin@ncs#
```

The idea with the optimizer thread is that it picks one of the edge service instances every 15 seconds in a round-robin fashion and optimizes it with respect to the current DC capacity, jitter and energy price.

This action is already implemented in `keep_optimizing_action.py`, but there is one problem. The code that actually does the optimization of the edge service instance is missing, and you have to provide it.

Overview 2.6 Build and Deploy

The commands to rebuild and deploy your solution are exactly the same as before. For YANG changes you first need to run `make` on the Linux command line.

```
$ make all
```

After the `make` went through fine, or if you only did Python or template changes, you need to load the new package versions with `packages reload`. We recommend to also taking a look at the package oper-status afterwards.

```
admin@ncs# packages reload
...
admin@ncs# show packages package oper-status
```

Overview 2.7 Test

Something like following behavior is expected once the solution is complete, but it is certainly up to you to decide how closely to recreate this behavior.

First, let's see that we have a good initial state. Packages loaded without error, there is a jitter value per dc as before, but now also energy-price and edge-clients. There are no edge services at this point.

```
admin@ncs# show packages package oper-status | tab
```

NAME	UP	PROGRAM		PYTHON	UNINITIALIZED	PACKAGE			
		CODE	JAVA			META	FILE	ERROR	WARNINGS
		ERROR	UNINITIALIZED			DATA	LOAD	INFO	
edge-nc-1.0	X	-	-	-	-	-	-	-	-
firewall-nc-1.0	X	-	-	-	-	-	-	-	-
origin-nc-1.0	X	-	-	-	-	-	-	-	-
skylight-nc-3.0	X	-	-	-	-	-	-	-	-
streaming	X	-	-	-	-	-	-	-	-

```
admin@ncs# show dc
```

NAME	JITTER	PRICE	CLIENTS
dc0	20.0	100	-
dc1	25.0	75	-

```
admin@ncs# show edge
```

% No entries found.

```
admin@ncs#
```

Let's go and create all four edge service instances right away. A commit dry-run shows what configuration would be sent to the various devices. The output is actually quite substantive now, so we have shortened it a bit, in order to keep you from flipping too many pages.

Worth noting is that there is some twamp reflector config for all edge devices. A bunch of (different) firewall rules are provided for both fw0 and fw1. Both Origin video servers get their content catalogs updated with (different) content, depending on what titles are needed for each Edge video cache device. The skylight configuration gets a bunch of twamp sessions to set up. Finally the edge services themselves are also mentioned, but have no configuration data.

```
admin@ncs# config
Entering configuration mode terminal
admin@ncs(config)# edge edge0
admin@ncs(config-edge-edge0)# edge edge1
```

```

admin@ncs(config-edge-edge1)# edge edge2
admin@ncs(config-edge-edge2)# edge edge3
admin@ncs(config-edge-edge3)# commit dry-run
cli {
  local-node {
    data devices {
      device edge0 {
        config {
          edge {
            + streaming-port 554;
            + origin-server 127.0.0.1;
          }
          twamp {
            session-reflector {
              + admin-state true;
            }
          }
        }
      }
    }
  }
  device edge1 {
    ...
  }
  device edge2 {
    ...
  }
  device edge3 {
    ...
  }
  device fw0 {
    config {
      acl {
        acl-sets {
          + acl-set edge3 ACL_IPV4 {
            + config {
              + name edge3;
              + type ACL_IPV4;
            }
            + acl-entries {
              + acl-entry 10 {
                ...
              }
            }
          }
        }
      }
    }
  }
  device fw1 {
    ...
  }
  device origin0 {
    config {
      origin {
        + streaming-port 554;
        + edge 127.0.0.1 {
          + streaming-port 554;
        }
        + content "Apollo 13" {
          + }
        + content "Some Like it Hot" {
          + }
        + content "Terminator 2: Judgement Day" {
          + }
        + content "The Matrix" {
          + }
      }
    }
  }
  device origin1 {
    config {
      origin {
        + streaming-port 554;
        + edge 127.0.0.1 {
          + streaming-port 554;
        }
      }
    }
  }
}

```

```

+ }
+ content Alien {
+ }
+ content "Blade Runner" {
+ }
+ content "Die Hard" {
+ }
+ content "Don't Look Up" {
+ }
+ content Ghostbusters {
+ }
+ content "Pulp Fiction" {
+ }
+ content "Terminator 2: Judgement Day" {
+ }
+ content "The Dark Knight" {
+ }
+ content "The Matrix" {
+ }
+ }
}
}
device skylight {
  config {
    sessions {
      + session 2848df3f-f27c-53ce-9373-ab64036d1f0b agent {
        + sessiontype twamp;
        + ifStateful true;
        ...
      + }
    }
  }
}
+edge edge0 {
+}
+edge edge1 {
+}
+edge edge2 {
+}
+edge edge3 {
+}
}
}
admin@ncs(config-edge-edge3)#

```

Ok, let's commit that.

```

admin@ncs(config-edge-edge3)# commit
Commit complete.
admin@ncs(config-edge-edge3)# exit
admin@ncs(config)# exit
admin@ncs#
System message at ...

```

At this point a whole bunch of system messages drop into the CLI. After some seconds a few more land there. This is because the Origin video server requires some time to load the titles needed by the Edge video caches, and the DC switch won't complete until they are available at the new DC.

Now, let's observe the edge plan state.

admin@ncs# show edge plan | tab

LOG				BACK				POST ACTION					
LOG NAME	FAILED	MESSAGE	ENTRY	TYPE	NAME	TRACK	GOAL	STATE	STATUS	WHEN	ref	STATUS	ID
CREATED	FROM	ENTRY	MESSAGE										

edge0	-	-	-	self	self	false	-	init	reached	2025-01-23T17:03:21	-	-	
								ready	reached	2025-01-23T17:03:21	-	-	
				dc	dc	false	-	init	reached	2025-01-23T17:03:21	-	-	
								skylight-configured	reached	2025-01-23T17:03:21	-	-	
								fw-configured	reached	2025-01-23T17:03:21	-	-	
								ready	reached	2025-01-23T17:03:21	-	-	
				edge	edge	false	-	init	reached	2025-01-23T17:03:21	-	-	
								connected-to-dc	reached	2025-01-23T17:03:21	-	-	create-reached
								connected-to-skylight	reached	2025-01-23T17:03:21	-	-	
								ready	reached	2025-01-23T17:03:21	-	-	
edge1	-	-	-	self	self	false	-	init	reached	2025-01-23T17:04:21	-	-	
								ready	reached	2025-01-23T17:04:21	-	-	
				dc	dc	false	-	init	reached	2025-01-23T17:04:21	-	-	
								skylight-configured	reached	2025-01-23T17:04:21	-	-	
								fw-configured	reached	2025-01-23T17:04:21	-	-	
								ready	reached	2025-01-23T17:04:21	-	-	
				edge	edge	false	-	init	reached	2025-01-23T17:04:21	-	-	
								connected-to-dc	reached	2025-01-23T17:04:21	-	-	create-reached
								connected-to-skylight	reached	2025-01-23T17:04:21	-	-	
								ready	reached	2025-01-23T17:04:21	-	-	
edge2	-	-	-	self	self	false	-	init	reached	2025-01-23T17:04:21	-	-	
								ready	reached	2025-01-23T17:04:21	-	-	
				dc	dc	false	-	init	reached	2025-01-23T17:04:21	-	-	
								skylight-configured	reached	2025-01-23T17:04:21	-	-	
								fw-configured	reached	2025-01-23T17:04:21	-	-	
								ready	reached	2025-01-23T17:04:21	-	-	
				edge	edge	false	-	init	reached	2025-01-23T17:04:21	-	-	
								connected-to-dc	reached	2025-01-23T17:04:21	-	-	create-reached
								connected-to-skylight	reached	2025-01-23T17:04:21	-	-	
								ready	reached	2025-01-23T17:04:21	-	-	
edge3	-	-	-	self	self	false	-	init	reached	2025-01-23T17:04:21	-	-	
								ready	reached	2025-01-23T17:04:21	-	-	
				dc	dc	false	-	init	reached	2025-01-23T17:04:21	-	-	
								skylight-configured	reached	2025-01-23T17:04:21	-	-	
								fw-configured	reached	2025-01-23T17:04:21	-	-	
								ready	reached	2025-01-23T17:04:21	-	-	
				edge	edge	false	-	init	reached	2025-01-23T17:04:21	-	-	
								connected-to-dc	reached	2025-01-23T17:04:21	-	-	create-reached
								connected-to-skylight	reached	2025-01-23T17:04:21	-	-	
								ready	reached	2025-01-23T17:04:21	-	-	

admin@ncs# show edge oper-status | tab

NAME	CHOSEN
edge0	dc0
edge1	dc0
edge2	dc0
edge3	dc0

Similar information follows for the other edge service instances. We can see edge0 reached state ready and selected dc1 as its DC. At this point, the show dc view gives a pretty good overview.

```
admin@ncs# show dc | tab
```

NAME	JITTER	PRICE	EDGE	CLIENTS
dc0	20.0	100	[edge3]	
dc1	25.0	75	[edge0 edge1 edge2]	

```
admin@ncs#
```

Now we immediately see that three edge service instances (edge0 .. 2) selected dc1, and we can see why – the energy price is considerably lower, even if the jitter is a bit higher than dc0. The edge3 service instance had to pick dc0 because dc1 had already reached its edge-capacity of 3 edge clients.

We can now play around with the jitter, energy-price and edge-capacity values. As before, we can order the skylight device to “detect” a jitter change and send a notification to NSO.

```
admin@ncs# devices device skylight rpc rpc-send-notification-low send-notification-low device
dc0 type jitter
admin@ncs# show dc | tab
ENERGY
NAME JITTER PRICE EDGE CLIENTS
-----
dc0  1.951  100   [ edge3 ]
dc1  25.0   75    [ edge0 edge1 edge2 ]
admin@ncs#
```

This immediately updates the jitter value for dc0, but nothing happens when it comes to moving edge service instances around. To do that, we can turn on the optimizer.

```
admin@ncs# actions optimize
result Started Optimizer
admin@ncs#
System message at 2024-01-31 16:57:16...
Commit performed by admin via console using cli.
admin@ncs# show dc | tab
ENERGY
NAME JITTER PRICE EDGE CLIENTS
-----
dc0  1.951  100   [ edge0 edge3 ]
dc1  25.0   75    [ edge1 edge2 ]

admin@ncs#
System message at 2024-01-31 16:57:31...
Commit performed by admin via console using cli.
admin@ncs# show dc | tab
ENERGY
NAME JITTER PRICE EDGE CLIENTS
-----
dc0  1.951  100   [ edge0 edge1 edge3 ]
dc1  25.0   75    [ edge2 ]

admin@ncs#
```


The optimizer takes one edge service instance every 15s and re-deploys it. That makes the service instance implement itself on the best DC under the current circumstances. The first service instance to move from dc1 to dc0 was edge0. Apparently that very low jitter value is more attractive than the energy-price difference. Edge1 soon follows. Edge2 is stuck on dc1 because dc0 is now at capacity.

To make this a little more interesting, we can start to vary the energy prices in the background too, and display the dc status on repeat.

```
admin@ncs# actions vary-energy-price
result Stopped Varying Energy Price
admin@ncs# show dc | tab | repeat
```

The show on repeat gives a display like this.

```
ENERGY
NAME JITTER PRICE EDGE CLIENTS
-----
dc0  1.951  97    [ edge0 edge1 edge3 ]
dc1  25.0   75    [ edge2 ]
```

That high jitter on dc1 really keeps service instances away. Press Ctrl+C to interrupt and get back to the command prompt again. Let's give dc1 a little lower jitter. We can specify the exact value if we want (doing so on either rpc-send-notification-low or -high commands has the same effect).

```
admin@ncs# devices device skylight rpc rpc-send-notification-low send-notification-low device
dc1 jitter 3.5
admin@ncs# show dc | tab | repeat
```

After a little while, the DC status might look like this instead.

```
ENERGY
NAME JITTER PRICE EDGE CLIENTS
```

```
-----  
dc0  1.951  138  [ edge2 ]  
dc1  3.5    97   [ edge0 edge1 edge3 ]
```

Once you get to this point, you have completed the lab and implemented a level5 service. Congratulations! 🎉

Task 2 Detailed Instructions

The steps that follow here are detailed implementation proposals for each one of the overview points discussed in the previous chapter.

Step 2.1 Add edge-capacity and energy-price

Since edge-capacity is meant to be a configurable item the list dc, you can add this after leaf skylight-agent-id and before container oper-status. The exact placement is not particularly important, but it needs to be inside list dc and outside the oper-status container, and not within any of the other leaf declarations.

```
leaf edge-capacity {  
  description "The number of edge services this DC can handle.";  
  type uint32;  
  default 3;  
}
```

Leaf energy-price is meant to be an operational status value, so it should go inside container oper-status of the list dc.

```
leaf energy-price {  
  description "The latest reported energy-price value for this DC.";  
  type uint32;  
  units USD/MWh;  
}
```

Remember to run make when you have YANG module changes, before you do the packages reload.

Step 2.2 Add edge-clients (optional)

So you want to be a hero and do all the optional work as well?! ☑ Nice, don't worry, this step won't take long. There's really only a couple of small things here. Inside the container oper-status in list dc, add a leaf-list like this.

```
leaf-list edge-clients {  
  description "The edge service instances currently depending on this DC.";  
  type leafref {  
    path /edge/name;  
  }  
}
```

```
}
```

In the next step (the DC selection function), there are a couple of more Python lines to add and remove edge service instance names to this leaf-list.

Step 2.3 Better DC Selection Function

There really isn't any right answer for how to write this function, so I would be shocked if you came up with a very similar solution. I will definitely conclude you had a peek at this solution, if that happens. :)

So here is our thinking. We have the existing optimization function from level4, which talks about jitter. Since we'll be combining multiple variables here, let's not call it `best_jitter` anymore, but `best_score`, and base that score on jitter and `energy_price` in some way.

How to weigh jitter and `energy_price` against each other? Well, if you are more concerned about `energy_price` than jitter, giving it a higher weight might make sense (see `JITTER_WEIGHT` and `PRICE_WEIGHT`). How do you compare values that have different units of measurement and are of totally different magnitudes? As an engineer, let me propose using logarithms. This way a change of one value from 0.26 to 0.35 will show as a relatively larger change than a change in another from 1000 to 1200. But let's not relive algebra class here, I'm sure whatever you pick will do.

```
class DCInit(NanoService):
    JITTER_WEIGHT = 1
    PRICE_WEIGHT = 2

    @NanoService.create
    def cb_nano_create(self, tctx, root, service, plan, component, state, proplist, compproplist):
        self.log.info(f'cb_nano_create: DCInit for {service.name}')

        # Find the DC with the lowest score (based on jitter, energy-price and DC capacity)
        best_score = 100000
        best_dc = None

        for dc in root.dc:
            if dc.oper_status.jitter is None or dc.oper_status.energy_price is None:
                self.log.info(f'Checking DC {dc.name}: DC is not ready')
                continue # Data not available yet, disregard this option

            if (len(dc.oper_status.edge_clients) >= dc.edge_capacity and
                service.name not in dc.oper_status.edge_clients):
                self.log.info(f'Checking DC {dc.name}: DC is full')
                continue # Already full, not an option

            dc_jitter = float(dc.oper_status.jitter)
            dc_price = int(dc.oper_status.energy_price)
            dc_score = (DCInit.JITTER_WEIGHT * math.log10(dc_jitter) +
                       DCInit.PRICE_WEIGHT * math.log10(dc_price))
```

```

        self.log.info(f'Checking DC {dc.name}: jitter {dc_jitter} price {dc_price} -> '
                      f'score {dc_score}')

        if dc_score < best_score:
            best_score = dc_score
            best_dc = dc

    if best_dc is None:
        raise Exception('No DC found')

    # Value goes into operational data, usable by templates applied at later stages
    service.oper_status.chosen_dc = best_dc.name
    self.log.info(f'Using DC {root.dc[service.oper_status.chosen_dc].name}')

```

If you chose to do the optional step with edge-clients above, then revise the tail end of the code to the thing below instead. That will remove the edge service instance name from the previous DC's leaf-list edge-clients and add it to the newly selected one.

```

if best_dc is None:
    raise Exception('No DC found')

if service.oper_status.chosen_dc:
    self.log.info(f'Leaving DC {root.dc[service.oper_status.chosen_dc].name}')
    # This is just to keep operators informed, not affecting the service logic
    root.dc[service.oper_status.chosen_dc].oper_status.edge_clients.remove(service.name)

# Value goes into operational data, usable by templates applied at later stages
service.oper_status.chosen_dc = best_dc.name
self.log.info(f'Using DC {root.dc[service.oper_status.chosen_dc].name}')

# This is just to keep operators informed, not affecting the service logic
root.dc[service.oper_status.chosen_dc].oper_status.edge_clients.create(service.name)

```

Step 2.4 Optimization doesn't have to be Notification Based

In order not to re-deploy service instances as Skylight notifications are flowing in, just delete the following lines from `skylight_notification_action.py`.

```

# Automatically re-deploy services that are on the DC in the notification,
# but let all other services stay as they are
for edge in r.streaming_edge:
    if edge.oper_status.chosen_dc == notification.device:
        self.log.info(f'Re-deploying service {edge.name}')
        edge.reactive_re_deploy()
    else:
        self.log.info(f'Leaving service {edge.name} on {edge.oper_status.chosen_dc}'
                      f' as is')

```

Step 2.5 Keep Optimizing

What do you need to do to make a service instance optimize itself in the context of the current environment? We have said it, we have done it so many times already in this lab: re-deploy the service. The missing piece of code is really a one-liner.

```
edge.reactive_re_deploy()
```

We will show where to insert it below. To hide this simple fact a bit, we also changed the comments in the module to say <optimize> etc. when we actually mean re-deploy.

The comment at the top of the module should read

```
This module implements an NSO action callback that is used to optimize the running services.
When this action is invoked, it will start a background thread that reads the list of edge
services
and re-deploys them one by one at regular intervals (see INTERVAL_TIME below). Invoking the
same
action again will make the thread terminate.
```

The worker_thread comment should read

```
# worker_thread
# Reactively re-deploys edge service instances, one at a time.
# The thread runs every few seconds (INTERVAL_TIME) until requested to stop.
```

The log message and actual re-deploy function call should, near the bottom, just before the exception handling stuff, should read

```
edge = r.streaming__edge[edge_names.pop(0)]
self.log.info(f'Optimizer re-deploying service {edge.name}')
edge.reactive_re_deploy()
```

If you have eagle-eyes, you may have noticed that the code is calling `reactive_re_deploy()` rather than just `re_deploy()`. The difference is small but sometimes crucial. The `reactive-` variant invokes the service instance re-deploy as the same user that created the service, rather than the one launching the `re_deploy` call. In this lab, we're using the same NSO user in both situations, so it doesn't matter. We just want this code to do the right thing.

Step 2.6 Build and Deploy

This has been described many times now, and the commands to rebuild and deploy your solution are exactly the same as before. For YANG changes you first need to run `make` on the Linux command line.

```
$ make all
```

After the `make` went through fine, or if you only did Python or template changes, you need to load the new package versions with `packages reload`. We recommend to also taking a look at the package `oper-status` afterwards.

```
admin@ncs# packages reload
...
admin@ncs# show packages package oper-status
```

Step 2.7 Test

If everything is running fine now, the command sequence described earlier in this chapter in section Overview 2.7 should work on your system and produce something similar.

During the development work, there are a couple of log files that may come in handy, to see what is going on in the system.

The most useful one is probably going to be the streaming service log file.

```
$ less logs/ncs-python-vm-streaming.log
```

The development log, and XPath trace log may occasionally also be useful.

```
$ less logs/devel.log  
$ less logs/xpath.trace
```

Summary

Thank you for staying with us all the way till the end! We hope you have enjoyed the lab, and we are sincerely interested in any feedback you may have.

This lab was developed in order to give the network automation levels a concrete form. Hopefully you now have a good intuitive feel for what level 3 means, and how that is different from level 4. The system behavior at level 5 is qualitatively different from level 4, but did you notice how small the code-wise differences between a level 4 system and a level 5 system can be?

The NSO development team is trying hard to enable you to design the system behavior you need with good performance at a minimum amount of modeling and coding work. The DC selection function in the level 5 service implementation shows where we are on this journey.

Any ideas and feedback you have can be shared in the Webex room that remains open for a few weeks after the lab.

The lab is publicly available at gitlab for anyone to go through, or to file suggestions or questions. You will find it at

<https://gitlab.com/nso-developer/nso-automation-levels-example>

Related Sessions at Cisco Live

If your interest in network automation or NSO wasn't satiated by this lab, there are of course plenty of sessions throughout the Cisco Live 2024 Amsterdam week. You will find lots of interesting sessions in the Cisco Events catalog. Here are some highlights we would recommend.

- **DISCUSS**
Book your one-on-one Meet the Engineer meeting or chat in the Webex room for this lab
- **CONNECT TO THE LEARNING NETWORK**
DEVWKS-1961: Demystifying NSO (earn Cisco Continuing Education credits)
- **MORE HANDS-ON**
LABOPS-1305: Real-time Services Automation with NSO and Model-Driven Telemetry
LABOPS-1507: Automating Services with NSO, Walk-in Lab
- **LISTEN TO A USER**
BRKOPS-2040: CiscoLive: Brought To You By Cisco NSO and Cisco Modeling Labs

Beyond Cisco Live

- Discussion: [NSO Developer Hub](#)
- LinkedIn: [Cisco Automation Developer Community](#)
- YouTube: [Cisco Automation Developer Hub](#)
- Book: Network Programmability with YANG
[@Pearson/Addison-Wesley](#) [@O'Reilly](#) [@Amazon](#)
- Annual Conference: [Cisco Automation Developer Days](#)