CISCO Live ! | Amsterdam
February 9–13, 2026

LTRXAR-2008: Accelerate Your Network Operations

Instructor-Led Labs

Andrea Testino, Christopher Hart

Copyright © Cisco

Table of contents

1. Intro 3
   - Learning Objectives 3
   - Lab content and structure 6
   - Disclaimer 7
   - Getting started 8
2. Lab 12
   - Task 01 - SSH to the lab devices 12
   - Task 02 - Editing YAML files in VS Code 18
   - Task 03 - Global configuration 30
   - Task 04 - Device group configuration 48
   - Task 05 - Single-device configuration 57
   - Task 06 - Variables 63
   - Task 07 - Templates type `model` (Optional) 72
   - Task 08 - Templates type `file` (Optional) 81
   - Task 09 - Templates type `cli` (Optional) 91
   - Task 10 - Pre-checks with schema validation 95
   - Task 11 - Post-checks with `nac-test` (Optional) 104
   - Task 12 - Cleanup 112
   - Task 13 - Run a CI/CD pipeline 117
   - Task 14 - Extend the pipeline with automated tests (Optional) 136
   - Task 15 - Branch and merge-request workflow (Optional) 148
   - Conclusion 163
3. Glossary 166
4. Lab topologies 168
   - Lab devices 168
   - Lab VMs 168
5. Appendices 172
   - APPENDIX I 172
   - APPENDIX II 175
   - Appendix III - `nac-test` reference 178
   - Appendix IV - ISP router configuration 180
   - Appendix V - Quick reference card 181

Copyright © Cisco

1. Intro

Learning Objectives

This step-by-step guide walks you through the **Network as Code for IOS XE** lab. By the end, you will know how to deploy and manage IOS XE devices with declarative infrastructure-as-code and CI/CD pipelines.

> **You're on the right guide if this URL is** `cl-ltr.ciscolabs.com/4eb6f2de2e`
>
> The Chrome start page bookmarked inside the Win10 lab VM points to an **older copy of this guide from a previous Cisco Live delivery** - a known deployment-side issue we couldn't correct in time. Always read this guide from the tab on your laptop's left monitor: **https://cl-ltr.ciscolabs.com/4eb6f2de2e/**. See Getting Started for the full context.

> **New to these acronyms?**
>
> See **Glossary** in the top navigation for a brief overview on the definitions. Keep it open in a side tab if you hit one you don't recognize.

What is Network as Code?

Network as Code (NaC) is a methodology that applies DevOps principles to network management through declarative data models. Rather than writing scripts or clicking through GUIs, network engineers describe their intended network state in human-readable YAML files. The Network as Code toolchain - built on Terraform - handles the how: translating intent into device-specific configuration, tracking state, and pushing only what changed.

Network as Code is an umbrella that spans multiple Cisco platforms and architectures. Each platform has its own Terraform module and provider, but they all share the same methodology: YAML intent files → Terraform module → platform provider → device. The supported platforms include ACI, Catalyst SD-WAN, Meraki, Catalyst Center, ISE, NX-OS, IOS XR, and - the focus of this lab - **IOS XE**.

IOS XE as Code is the Network as Code implementation for Cisco IOS XE devices (Catalyst 9000, C8000V, ISR, ASR, etc.). It uses the `terraform-iosxe-nac-iosxe` module and the `terraform-provider-iosxe` provider to push YANG-modelled configuration over NETCONF. If you learn the workflow here, you can apply the same patterns to any other Network as Code-supported platform - the data model schemas differ (after all, these are very different products!), but the overall approach of driving intended configuration through declarative YAML and Terraform is intentionally consistent across the family.

Why this lab matters

Traditional network management relies on manual CLI commands, scripts, and brittle templates. The typical failure modes - configuration drift, human error, no change history, and no safe way to preview or test changes - are well understood but rarely solved at scale.

This lab shows how Network as Code addresses each of them using:

- **Declarative configuration** - define the desired state, not the steps to reach it
- **Terraform** - industry-standard Infrastructure-as-Code engine
- **Schema validation** - catch misconfigurations before they ever reach a device
- **Post-deployment testing** - verify what was deployed actually matches what you asked for
- **GitLab CI/CD pipelines** - automate validation, deployment, and testing end-to-end

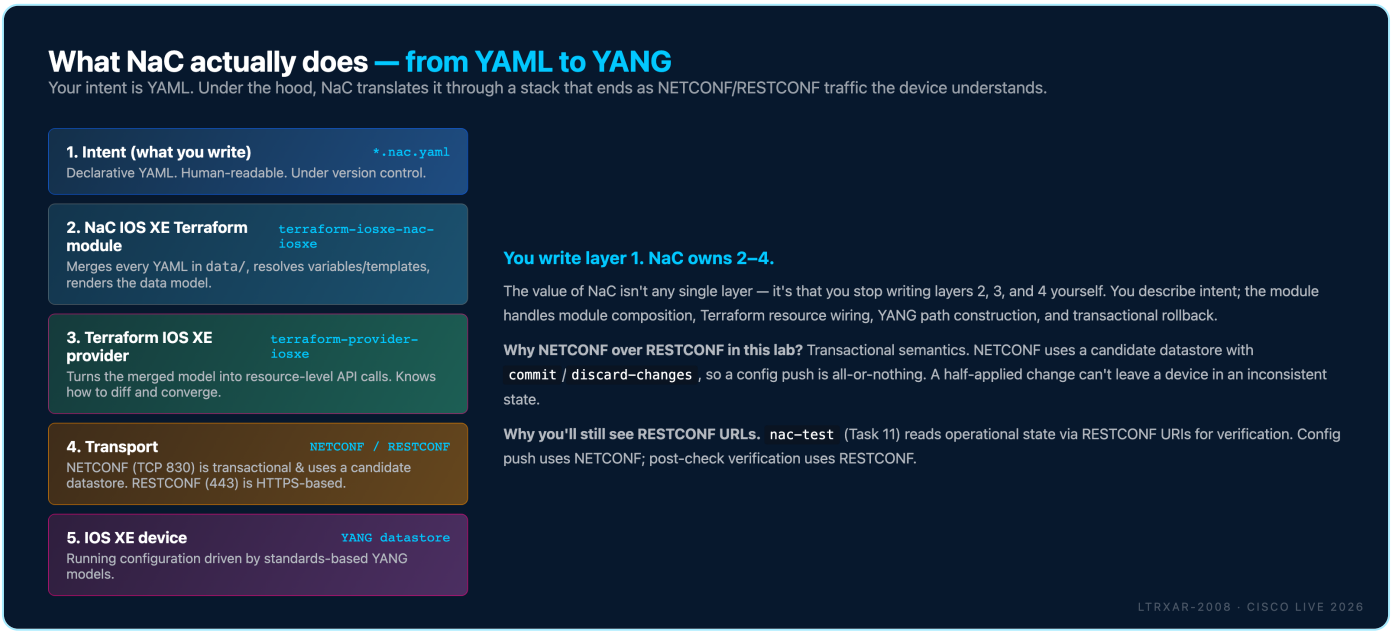
Copyright © Cisco

Terminology

IOS XE as Code and *Network as Code for IOS XE* are used interchangeably in this guide - both refer to the same automation solution.

What the stack looks like

Network as Code isn't a single tool - it's a layered stack. You write intent in YAML; the stack turns that into actual configuration on the device:



You'll spend all your time in **layer 1** (the YAML). The rest of the stack - the Terraform module, the IOS XE provider, the NETCONF transport - is maintained by Cisco and the Network as Code open-source community.

Supported IOS XE versions

The lab's YAML configurations push through the Terraform IOS XE provider, which is built on the IOS XE YANG data models published with IOS XE **17.12, 17.15, and 17.18**. Those three releases are what the provider has been tested against and are the recommended floor for using the full Network as Code data model.

Earlier releases may work for individual resources, but YANG container shapes can shift between IOS XE versions (some keys are silently absent or named differently pre-17.12), so identical YAML may not produce identical config on older devices. The current tested matrix lives in the provider documentation.

New to YANG? It's the standardized data-modelling language IOS XE uses to describe its configuration surface. Cisco DevNet has a short intro to YANG data modelling that's worth ~15 minutes if you want the deeper picture. The lab doesn't require you to read or write YANG yourself (the provider does that translation), but it's why version compatibility matters: a YANG container that exists in 17.15 may not exist (or may have a different shape) in 17.6.

Copyright © Cisco

Lab scope

This lab uses virtualized IOS XE devices in Cisco Modeling Labs (CML) and a GitLab instance running in the lab environment. It is designed for learning; the configurations shown are **not intended for production use as-is**. Production deployments require additional considerations around state storage, secret management, access control, and approval workflows - most of which we note in context but do not implement here.

What you'll learn

By completing this lab, you will gain hands-on experience with:

- Writing declarative IOS XE configurations in Network as Code YAML format
- Deploying configurations using the Network as Code Terraform module
- Understanding the Network as Code configuration hierarchy: **global → group → device**
- Using variables and templates for reusable, scalable configurations
- Pre-deployment validation with `nac-validate` (schema and semantic checks)
- Post-deployment validation with `nac-test` (automated Robot Framework tests)
- Automating the full Infrastructure-as-Code lifecycle with GitLab CI/CD pipelines
- Implementing GitOps workflow safeguards (branch protection, merge requests)

About this guide

Authored by Andrea Testino and Christopher Hart, building on the original Amsterdam delivery by Asier Arlegui and Balu Novak-Bohak.

Lab Content

Copyright © Cisco

Lab content and structure

This lab is designed for a broad range of experience levels - from learners new to VS Code, Git, or Terraform, to engineers who use CI/CD pipelines daily. The guide is segmented into **recommended tasks** (suitable for everyone, with step-by-step guidance) and **optional tasks** (for learners who want to go deeper or finish early).

Fundamentals

- **Task 01** - SSH to network devices in the lab
- **Task 02** - Edit intent configuration YAML files with VS Code

Deploy configuration with Terraform manually

- **Task 03** - Global configuration
- **Task 04** - Device group configuration
- **Task 05** - Single device configuration
- **Task 06** - Variables
- *Task 07* (Optional A) - Templates type `model`
- *Task 08* (Optional A) - Templates type `file`
- *Task 09* (Optional A) - Templates type `cli`

Pre-checks and post-checks (manual)

- **Task 10** - Pre-checks with schema validation
- *Task 11* (Optional B) - Post-checks with `nac-test`

CI/CD pipelines

- **Task 12** - Cleanup the lab
- **Task 13** - Run a CI/CD pipeline (validate + plan + deploy)
- *Task 14* (Optional C) - Extend the pipeline with automated tests
- *Task 15* (Optional C) - Merge Request workflow

Learning Objectives | **Disclaimer**

Copyright © Cisco

Disclaimer

The configurations, patterns, and examples in this lab are provided for educational purposes. They illustrate **how** Network as Code works and are a good starting point for your own experimentation, but they are not a production design.

Before applying any of these patterns to a production network, engage your Cisco account team or a Cisco partner to validate the approach against your specific requirements – including device lifecycle, state storage, secret handling, change-approval workflows, and integration with existing monitoring and ticketing.

Lab Content **Getting Started**

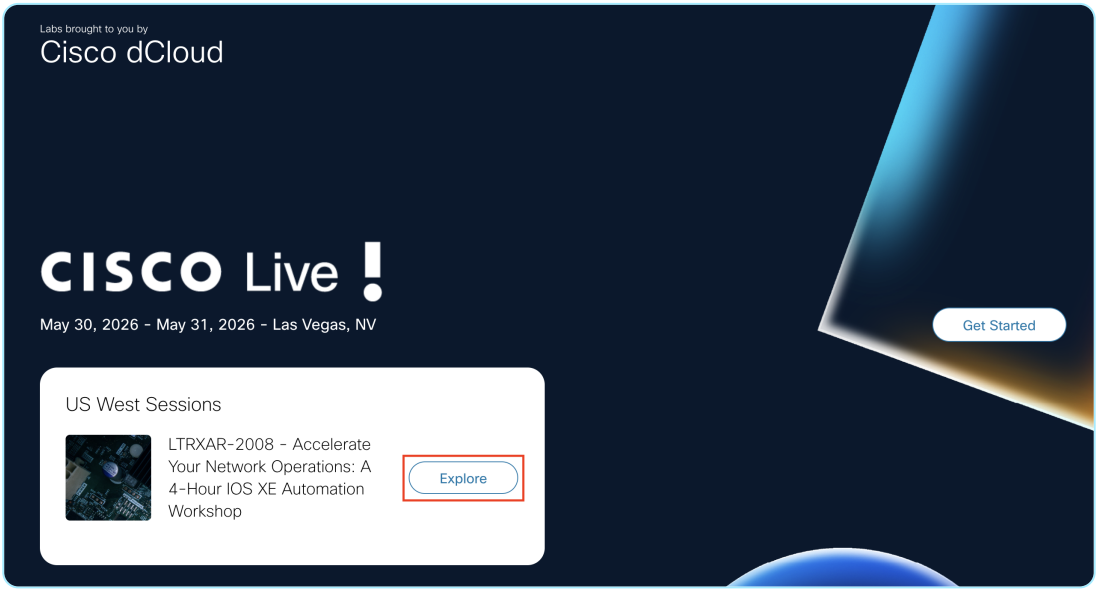
Copyright © Cisco

Getting started

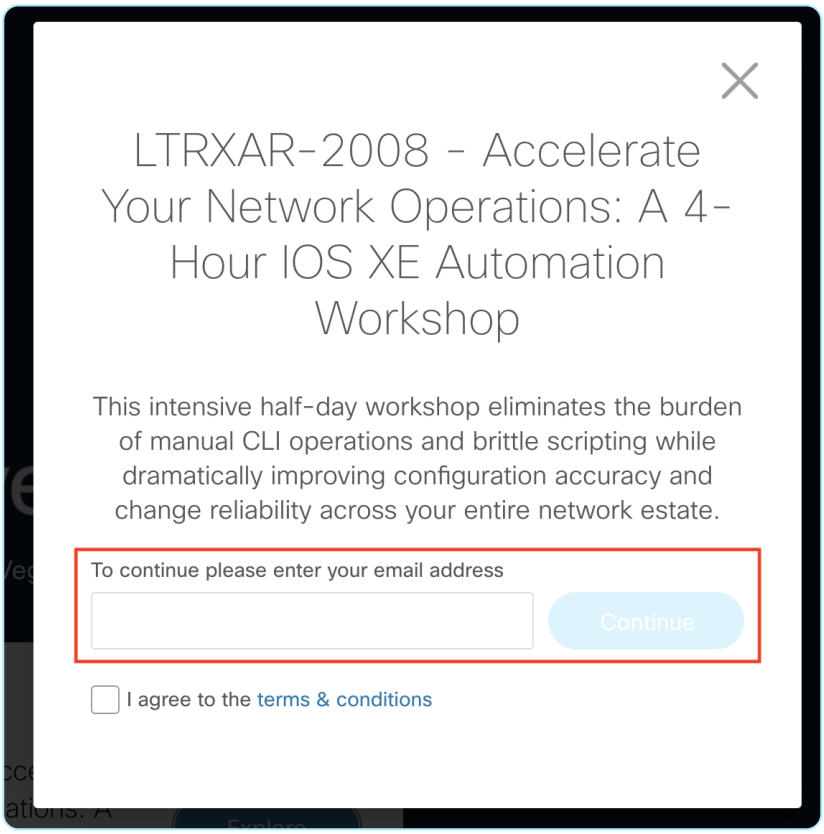
Connecting to your lab environment

Follow these three steps to open the Windows 10 VM where you'll do all your work.

Step 1 - Click Explore. On the right monitor at your station, locate the dCloud session card for **LTRXAR-2008** and click the **Explore** hyperlink.

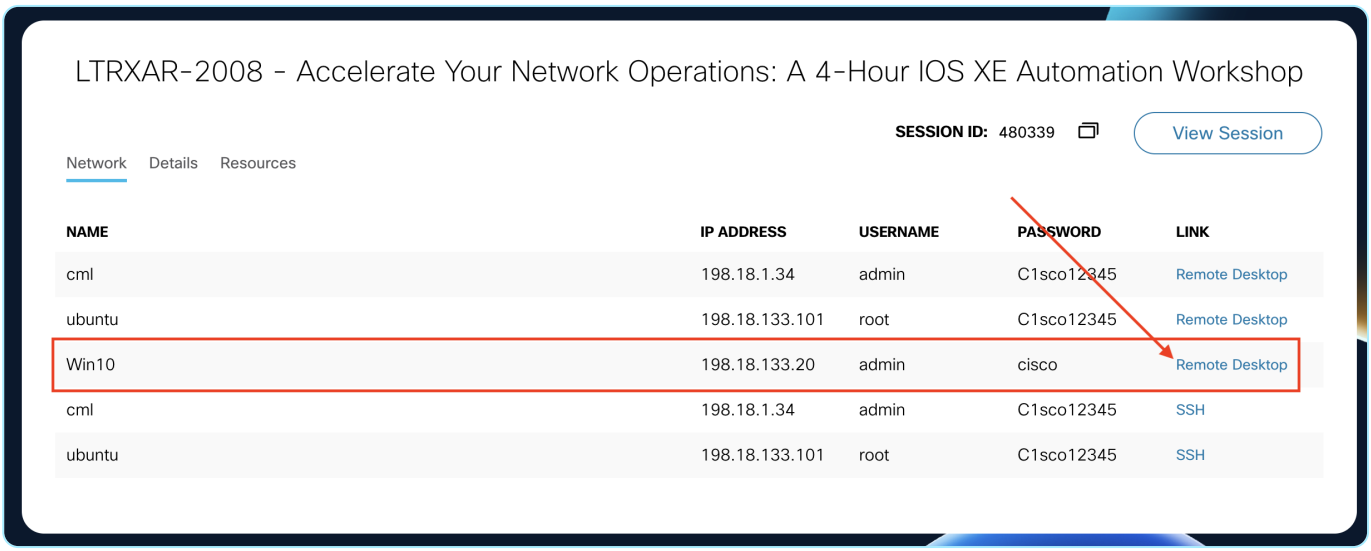


Step 2 - Enter your email. In the dialog that appears, type the email address you used to register for Cisco Live and click **Continue**.



Copyright © Cisco

Step 3 - Open the Win10 Remote Desktop. You'll see a resource details page listing the virtual machines in your lab pod. Find the row for **Win10** and click the **Remote Desktop** link - this opens a Web RDP session directly in your browser.



Once the Web RDP session loads, you're inside the lab VM. Everything you need - VS Code, Solar-PuTTY, WSL Ubuntu, and Chrome - is already installed. You'll open the **WSL Ubuntu terminal** frequently throughout the lab; it's the orange Ubuntu icon on the desktop labelled **Ubuntu 22.04.5 LTS**.

> **Read the lab guide from your laptop, not from inside the VM**
>
> The Chrome start page bookmarked on the Win10 VM currently points to an **older copy of this guide from a previous Cisco Live delivery** - a known deployment-side issue we weren't able to correct in time. Please ignore that bookmark.
>
> The current guide is the one already open on your laptop's left monitor when you sit down: **https://cl-ltr.ciscolabs.com/4eb6f2de2e/**. Keep that tab in front of you for the entire session.
>
> You will still do every hands-on action (VS Code edits, WSL terminal commands, Solar-PuTTY sessions) **inside the Win10 VM**. The only thing that lives on your laptop is this guide. Copy-paste from your laptop into the Web RDP window takes an extra moment compared to staying inside the VM, but that is a much smaller cost than following outdated instructions.

For the full topology diagram and device inventory, see **Topologies** (also linked from the top navigation bar throughout the lab).

How the tasks flow

> **Tasks are sequential - each builds on the previous one**
>
> The **Recommended** path (Tasks 01 through 06, then 10, 12, 13) is designed to be followed **in order**. Each task uses configuration, files, or state produced by earlier tasks. Skipping a task means the next one's instructions won't match what's on your disk.
>
> **Optional** tasks (07-09 Templates, 11 Post-checks, 14 Extended pipeline, 15 Merge requests) can be cherry-picked based on your interest and remaining time. Where the order of optionals matters (e.g. Task 11 must run before Task 12 destroys its artifacts), the affected task page has its own callout at the top.
>
> **If you have ~90 minutes only:** Tasks 01 -> 02 -> 03 -> 05 -> 06 -> 10 -> 12 -> 13 gets you through the full "write intent, validate, deploy via CI/CD" loop. Skip Task 04 and all Optional tasks.

Copyright © Cisco

Readiness check (2 minutes, before Task 01)

Before sinking time into the first task, confirm the lab's three moving pieces are actually reachable. Open the **WSL Ubuntu terminal** - it's the orange Ubuntu icon on the desktop labelled **Ubuntu 22.04.5 LTS** - and run these three commands:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
# 1. Can you reach a lab device over SSH?
ssh -o StrictHostKeyChecking=no cisco@198.18.130.10 "show version | include Cisco IOS"
```

OUTPUT **Expected output**

```
Warning: Permanently added '198.18.130.10' (RSA) to the list of known hosts.
(cisco@198.18.130.10) Password:

Cisco IOS XE Software, Version 17.15.01
Cisco IOS Software [IOSXE], Catalyst L3 Switch Software (CAT9K_IOSXE), Version 17.15.1, RELEASE SOFTWARE (fc4)
Cisco IOS-XE software, Copyright (c) 2005-2024 by Cisco Systems, Inc.
All rights reserved.  Certain components of Cisco IOS-XE software are

client_loop: send disconnect: Broken pipe
```

Password prompt or access denied is fine here

The goal of this check is simply that the device **responds**. If it prompts for a password, SSH is working. If it says `Permission denied` or drops the connection after login, the device is reachable and healthy. Only flag it to a proctor if the command hangs or times out with no output.

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
# 2. Can you reach GitLab?
curl -sk https://198.18.133.101 -o /dev/null -w "gitlab HTTP %{http_code}\n"
```

`200` or `302` are both fine

GitLab usually answers the bare host with a `302` redirect (to `/users/sign_in`) on a freshly-logged-out browser session and a `200` once you've hit it before. Either response means the server is reachable - flag a proctor only if you get a connection timeout, `000`, or a 5xx.

OUTPUT **Expected output**

```
gitlab HTTP 200
```

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
# 3. Is Terraform on your PATH?
terraform version | head -1
```

OUTPUT **Expected output**

```
Terraform v1.12.2
```

If all three produce the expected output, you're ready. If any fails, flag it to a lab proctor now rather than hitting it mid-task.

Copyright © Cisco

Disclaimer

Task 01 - SSH to Devices

Copyright © Cisco

2. Lab

Task 01 - SSH to the lab devices

Estimated Time to Complete: ~10 minutes

Before you deploy any Network as Code configuration, you'll confirm baseline connectivity to the IOS XE lab devices and see what's already on them. You'll use Solar-PuTTY to SSH into each device, verify the running configuration, and identify the minimal "seed" config that makes Terraform-driven automation possible.

What you'll learn

- How to connect to IOS XE devices using Solar-PuTTY
- How to verify device information with basic `show` commands
- What minimal configuration is required to enable NETCONF automation

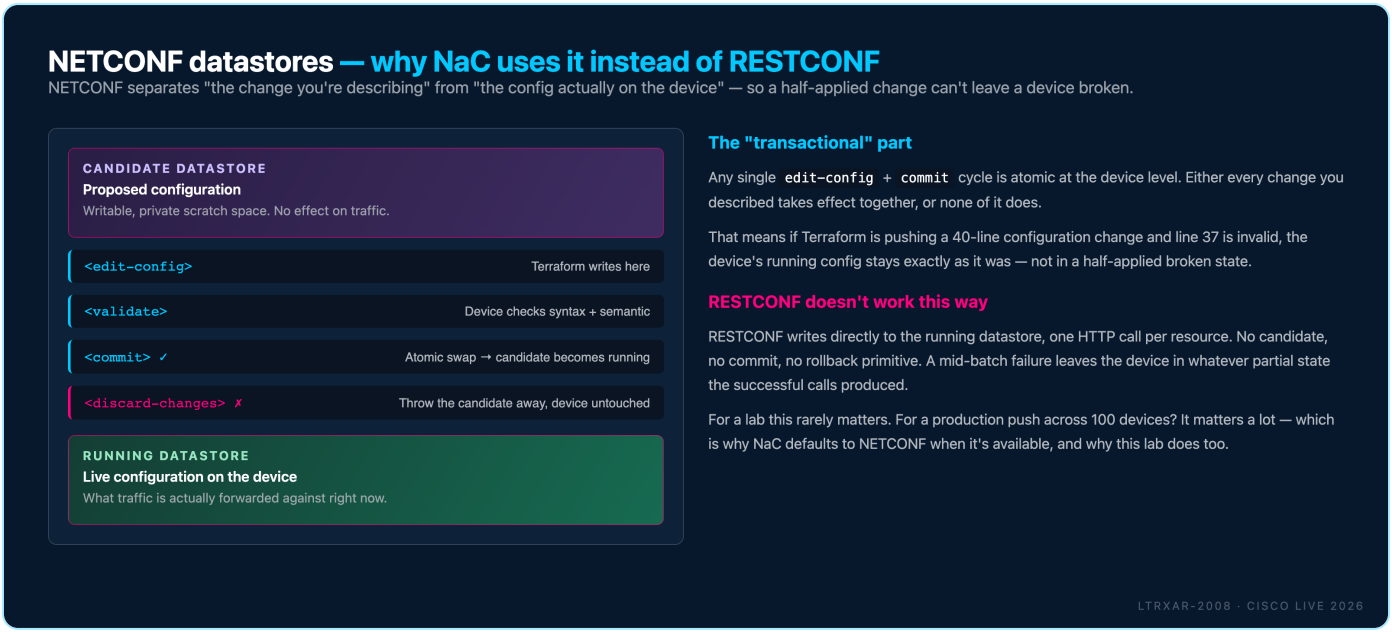
> **This lab uses NETCONF**
>
> IOS XE as Code uses **NETCONF** for both configuration and verification in this lab:
>
> - **Terraform pushes configuration over NETCONF** - it's transactional (all-or-nothing), uses a candidate datastore, and produces rich error reporting.
> - `nac-test` **(Optional Task 11) queries the device over NETCONF** - the generated Robot tests read operational state for post-deployment verification.

WHY NETCONF'S "TRANSACTIONAL" PROPERTY MATTERS

> **What's a NETCONF datastore?**
>
> A NETCONF **datastore** is just a named container for configuration - think of it as a git branch for your device's config. A device exposes multiple datastores; the two that matter here are `candidate` (writable scratch space) and `running` (the live configuration actually driving the forwarding plane). You propose changes to `candidate`, then `<commit>` them into `running` atomically. If anything fails mid-way, `<discard-changes>` throws the proposal away and `running` is untouched.

NETCONF separates the change you're describing (candidate datastore) from the config actually driving traffic (running datastore). Terraform writes to `candidate`, asks the device to validate, then issues `commit` - which atomically swaps candidate into running. If the commit fails at any point, `discard-changes` throws the proposed config away and the device keeps running exactly the same config it was running before.

Copyright © Cisco

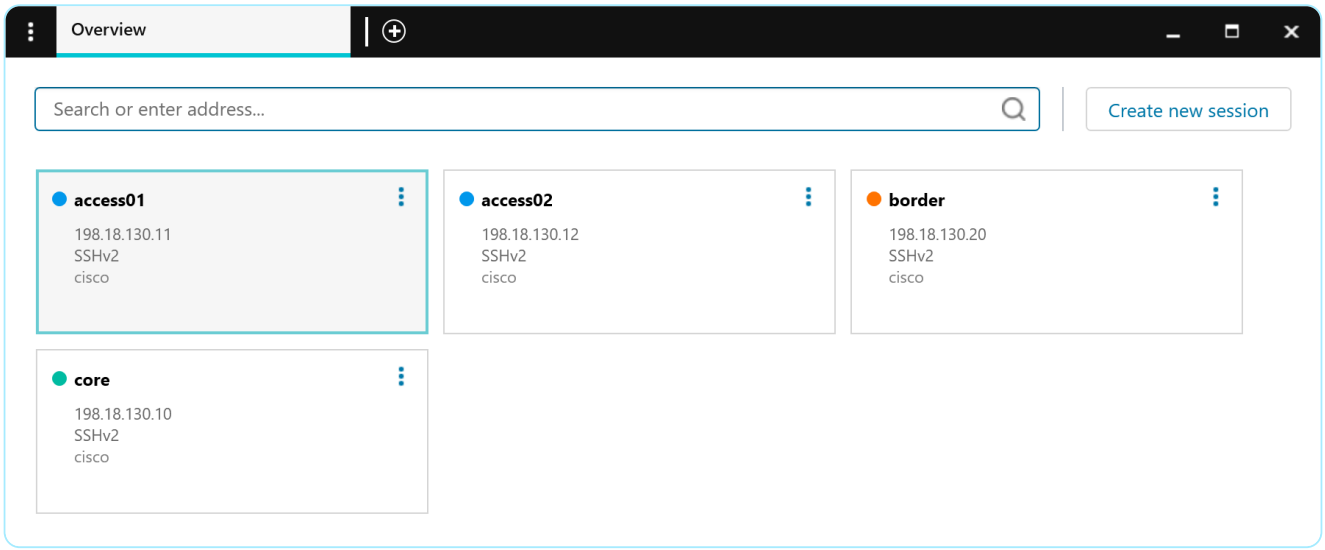


That's the whole "transactional" story in one picture: a broken `apply` can't leave a device half-configured, because "half-configured" isn't a state NETCONF allows. RESTCONF has no equivalent - it writes directly to `running`, one HTTP call per resource. A mid-batch failure leaves whatever the successful calls produced.

Open Solar-PuTTY

Solar-PuTTY is an enhanced SSH client with a tabbed interface for managing multiple device connections. It's pre-installed on the Win10 VM.

1. Double-click the **Solar-PuTTY** icon on the Win10 VM desktop.
2. You'll see the Solar-PuTTY interface with the lab devices already listed.

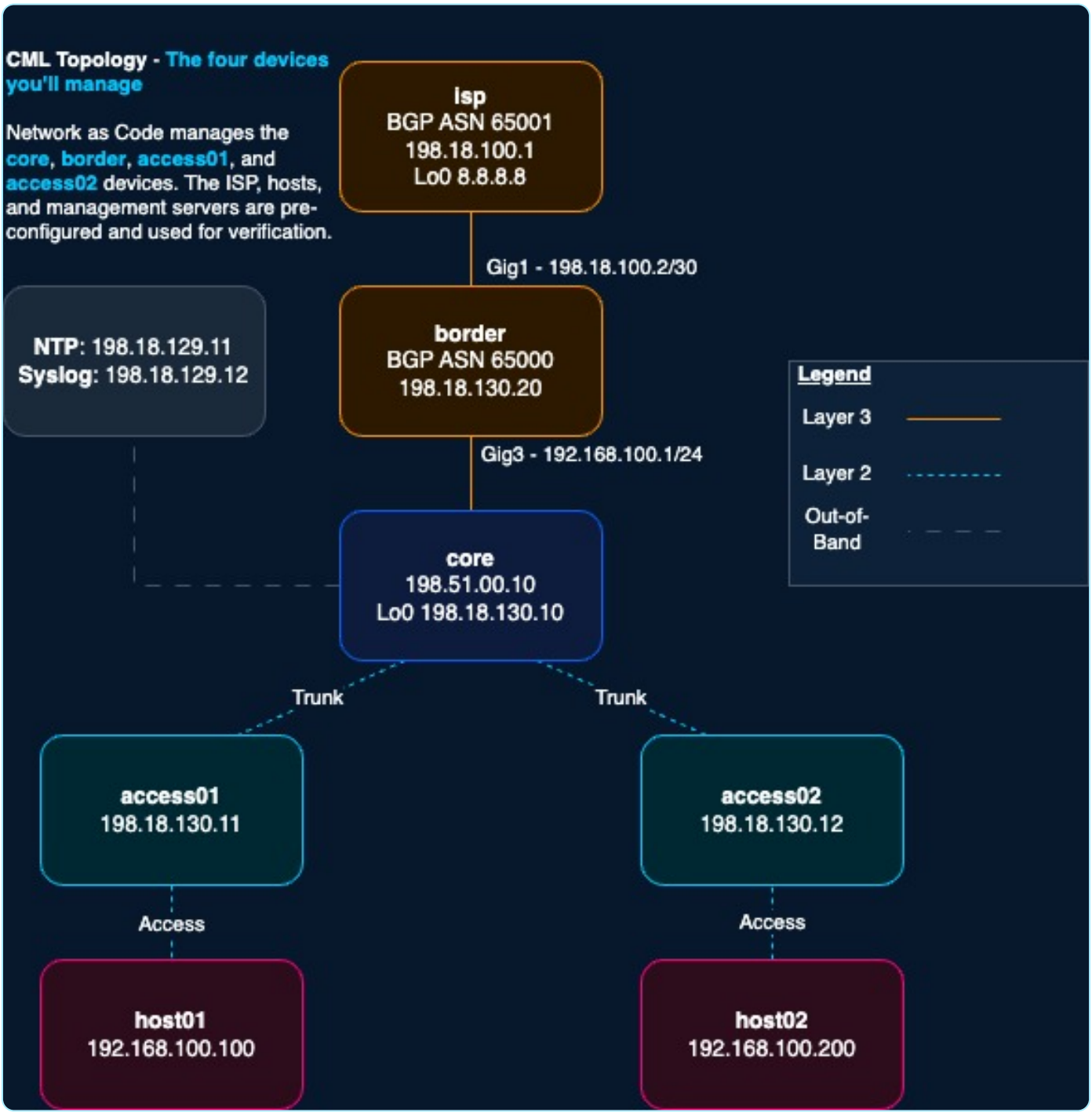


Copyright © Cisco

Connect to the lab devices

Credentials are **pre-configured** for all devices - just double-click the device name and you're in.

Device	Role	Management IP
core	Core switch	198.18.130.10
border	Border router	198.18.130.20
access01	Access switch	198.18.130.11
access02	Access switch	198.18.130.12



Copyright © Cisco

Additional devices in the topology

The lab also includes an **isp** router, **host01** / **host02** end-hosts, and **ntp-server** / **syslog-server** VMs. These are pre-configured for connectivity testing and are not managed via Network as Code in this lab. The NTP and Syslog servers are reachable via the management interface of each lab device (they're omitted from the diagram above for clarity).

Reference any time

The Topologies page (top navigation bar) has the full topology diagram, device IPs, and credentials for quick reference.

To connect: double-click the **core** device. Solar-PuTTY logs you in automatically.



Verify device information

Once you're on the device, run:

DEVICE CLI · PuTTY **core**

```
show version
```

Copy/paste from this guide

Every fenced command block has a copy icon in its top-right corner. Click it, then right-click inside Solar-PuTTY to paste.

You should see output confirming:

- **IOS XE version** (e.g., 17.x)
- **Platform** - `Cisco IOS XE Software, Version 17.x` with `Catalyst 9000` or `C8000V` hardware line
- **Uptime** and system details

This is a virtual Catalyst 9000 switch running IOS XE in CML.

Review the current configuration

DEVICE CLI · PuTTY **core**

```
show run
```


Copyright © Cisco

The running configuration is intentionally minimal. The devices are a clean slate for you to configure via Terraform - but you'll see a few essential lines that make that automation possible.

Look for:

OUTPUT **From show run output**

```
username nac_admin privilege 15 secret cisco
...
netconf-yang
```

WHAT EACH LINE DOES

Line	Purpose
`username nac_admin privilege 15 secret cisco`	Dedicated admin user that Terraform (and `nac-test`) authenticates as. Separating human and automation accounts is a core security practice.
`netconf-yang`	Enables the NETCONF-over-SSH server on TCP/830. This is the channel Terraform uses to push YANG-modelled configuration and `nac-test` uses to verify operational state.

Repeat `show version` and `show run` on **access01**, **access02**, and **border**. All four devices should look the same: minimal config plus the seed automation plumbing above.

What to observe across all devices

- Every device has a near-empty running configuration - ready for Network as Code to take over.
- Every device has the `nac_admin` user provisioned and **NETCONF** enabled for configuration and verification.
- The **candidate datastore** capability is **not yet enabled** - you'll turn it on as one of the first steps in Task 03 before pointing Terraform at the devices.
- No device has any of the configuration you're about to deploy (banners, ACLs, VLANs, BGP, etc.).

Enabling NETCONF manually (mostly done - one piece pending)

The lab devices already have the base NETCONF subsystem (`netconf-yang`) and the `nac_admin` user provisioned for you. **The one piece intentionally left unconfigured is the candidate datastore** - the `netconf-yang feature candidate-datastore` line. You'll enable that yourself in Task 03 - Step 3 so you see the command flow at least once; the rest of this section is reference for what to do on your own devices after Cisco Live.

Copyright © Cisco

Commands to enable NETCONF on your own devices

DEVICE CLI · PuTTY **any IOS XE device**

```
configure terminal
 username nac_admin privilege 15 secret cisco
 netconf-yang
 netconf-yang feature candidate-datastore
end
write memory
```

After enabling `netconf-yang`, give the subsystem ~60 seconds to initialize before pointing Terraform at the device.

The `netconf-yang feature candidate-datastore` command is the one that activates the candidate/commit/discard-changes flow shown in the datastore diagram above. Without it, NETCONF on IOS XE falls back to a simpler mode that writes directly to running - which works for single-RPC changes but loses the "all-or-nothing across many RPCs" guarantee. Enabling the candidate datastore is what makes `device_transaction = true` on the module useful (see Task 03).

In this lab, the first three lines (`username` , `netconf-yang`) are already applied to every device. Only `netconf-yang feature candidate-datastore` is left for you to run during Task 03.

You'll verify NETCONF reachability from WSL Ubuntu in Task 03 using a quick `ssh -s` handshake against port 830, then enable the candidate datastore feature on each device.

What you've accomplished

- Connected to all four IOS XE devices via Solar-PuTTY
- Verified device information with `show version`
- Reviewed the minimal running configuration
- Identified the configuration lines that enable Network as Code automation (`username nac_admin …` and `netconf-yang`)
- Confirmed all devices are ready for Network as Code deployment

In the next task, you'll start creating the YAML configuration files that describe your desired network state.

Getting Started

Task 02 - Editing YAML Files

Copyright © Cisco

Task 02 - Editing YAML files in VS Code

Estimated Time to Complete: ~15 minutes

In this task you'll set up the project directory and use VS Code to edit Network as Code IOS XE intent configuration YAML files.

What you'll learn

By the end of this task you will have:

- Set up an IOS XE as Code project directory under `~/nac-iosxe/` in WSL
- Edited YAML configuration files using VS Code and the Red Hat YAML extension
- Registered all four lab devices in their own per-device YAML files

Create the project directory

You'll create a dedicated project directory in your Windows Subsystem for Linux (WSL) home directory to organize your Network as Code configuration files and related resources. This location will serve as your workspace for storing YAML configuration files, Terraform files, and state information.

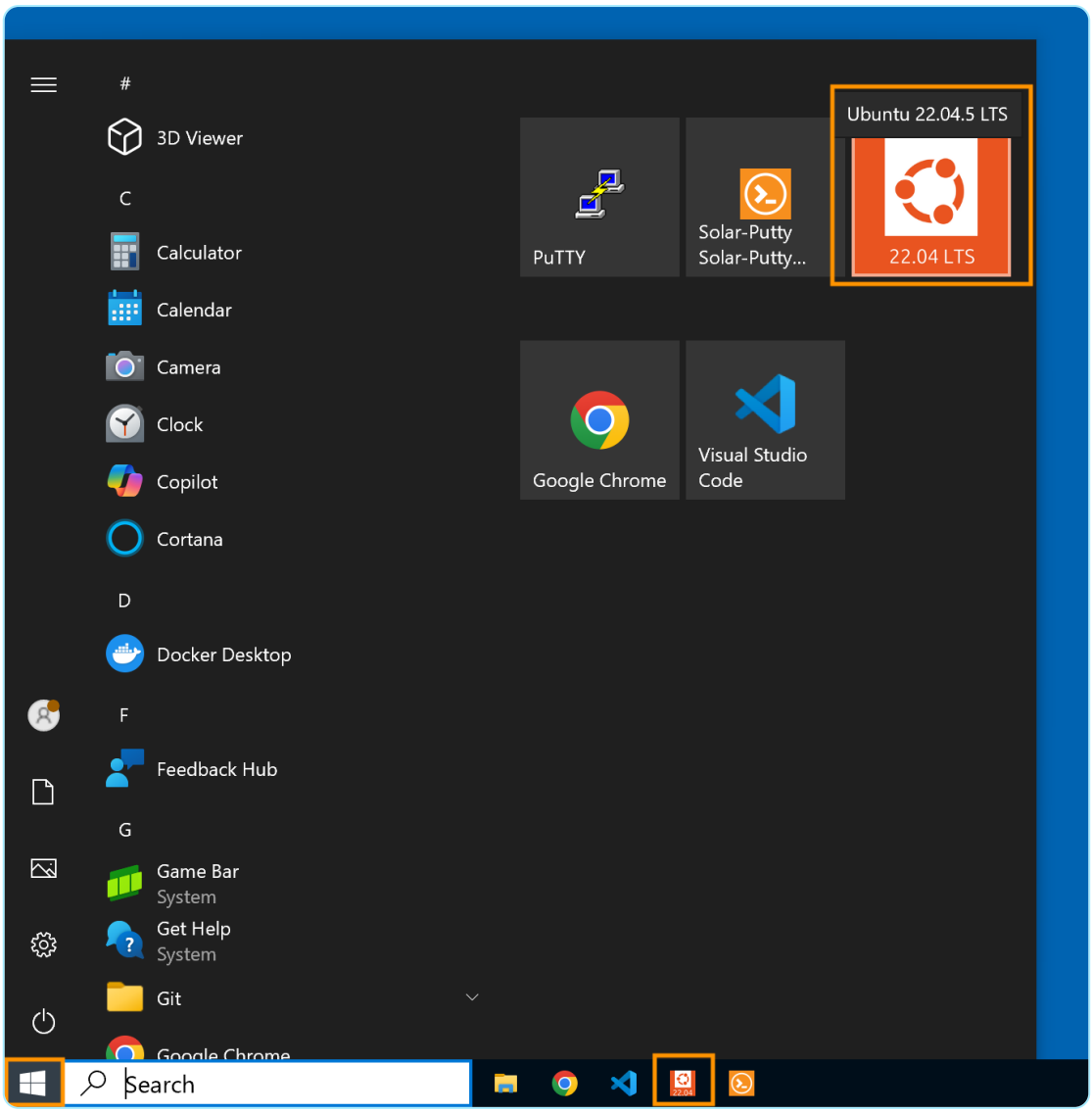
WHAT IS WSL?

Windows Subsystem for Linux (WSL) allows you to run a Linux environment directly on Windows without a virtual machine. WSL is pre-installed in this lab, and you'll use it to run Terraform commands because most DevOps tools are designed for Linux.

OPEN WSL TERMINAL (UBUNTU)

Open Windows Subsystem for Linux (WSL) terminal. You can use the desktop shortcut, the icon on the taskbar, or on the Start menu.

Copyright © Cisco



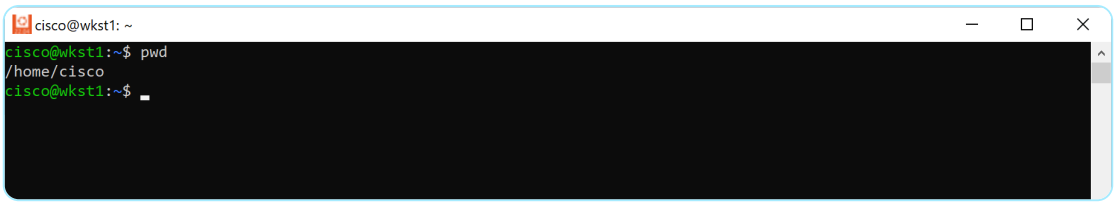
When you open WSL, you'll automatically start in your home directory (`/home/cisco` or `~/`).

Verify your current location:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
pwd
```

You should see `/home/cisco` displayed.



Copyright © Cisco

Create the IOS XE as Code project directory:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
mkdir nac-iosxe
```

> **Copy and Paste**
>
> You can copy commands directly from this lab guide by clicking on the icon at the top right corner of the command block and paste them into the WSL terminal using **right-click**.

This creates a dedicated folder named `nac-iosxe` for all your Network as Code IOS XE project files.

Navigate into the new directory:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

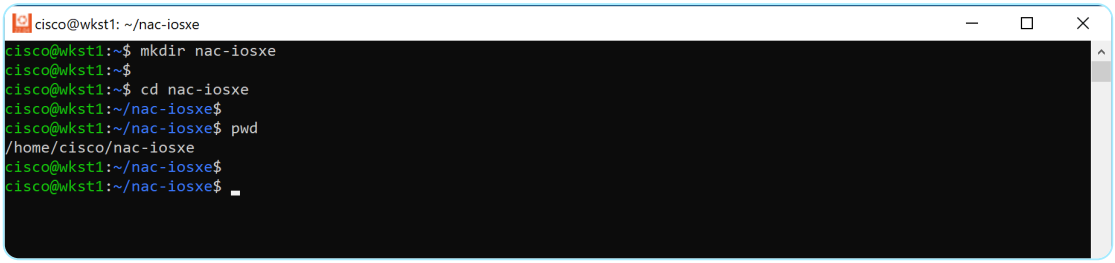
```
cd nac-iosxe
```

Verify you're in the correct location:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
pwd
```

You should now see `/home/cisco/nac-iosxe` displayed.



Create project structure

Now you'll create a folder structure and placeholder files for your Network as Code IOS XE project.

Create the data directory structure:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
mkdir -p data/devices data/groups data/templates
```

This creates a `data/` folder with subdirectories to organize your configuration by scope: `devices/` for per-device files, `groups/` for device-group files, and `templates/` for reusable template definitions.

Create placeholder files

Create the skeleton files you'll fill in below.

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

Copyright © Cisco

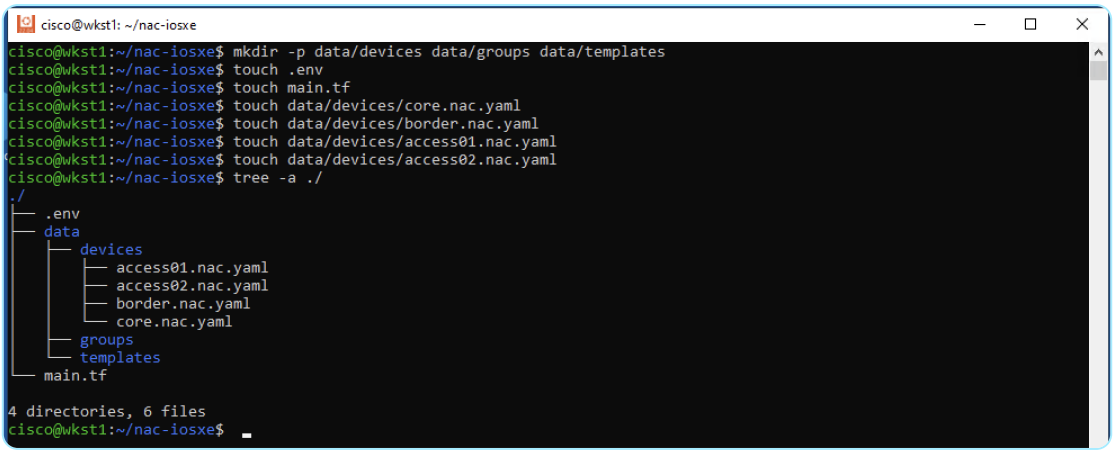
```
touch .env                                  # credentials for the IOS XE devices
touch main.tf                               # Terraform entry point
touch data/devices/core.nac.yaml            # per-device configuration files
touch data/devices/border.nac.yaml
touch data/devices/access01.nac.yaml
touch data/devices/access02.nac.yaml
```

One file per device is the pattern you'll use for the whole lab. It keeps each device's configuration self-contained and makes it obvious which YAML file to edit when you need to change something for a specific device.

Why one file per device, not a single inventory file?

The IOS XE as Code data model exposes a top-level `iosxe.devices` list. YAML allows the same list to be split across multiple files and merged at load time - **but only if every list entry is uniquely identified by its `name` field.** Defining each device in its own file (with its own `name`) keeps that invariant obvious, and it scales cleanly: adding a device later is "create one file," not "edit three."

The pattern works because Network as Code merges entries across files by the `name` field. If you typo a device's `name` in one file, you'll end up with **two separate entries** instead of one merged one - it's the single most common way a learner's config silently misbehaves. You'll see merge behavior in detail in Task 04 once you have three files contributing to one device's config at once.



Verify your project structure

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
tree -a
```

You should see:

OUTPUT **Expected output**

```
cisco@wkst1:~/nac-iosxe$ tree -a
.
├── .env
├── data
│ ├── devices
│ │ ├── access01.nac.yaml
│ │ ├── access02.nac.yaml
│ │ ├── border.nac.yaml
│ │ └── core.nac.yaml
│ ├── groups
│ └── templates
└── main.tf

4 directories, 6 files
```


Copyright © Cisco

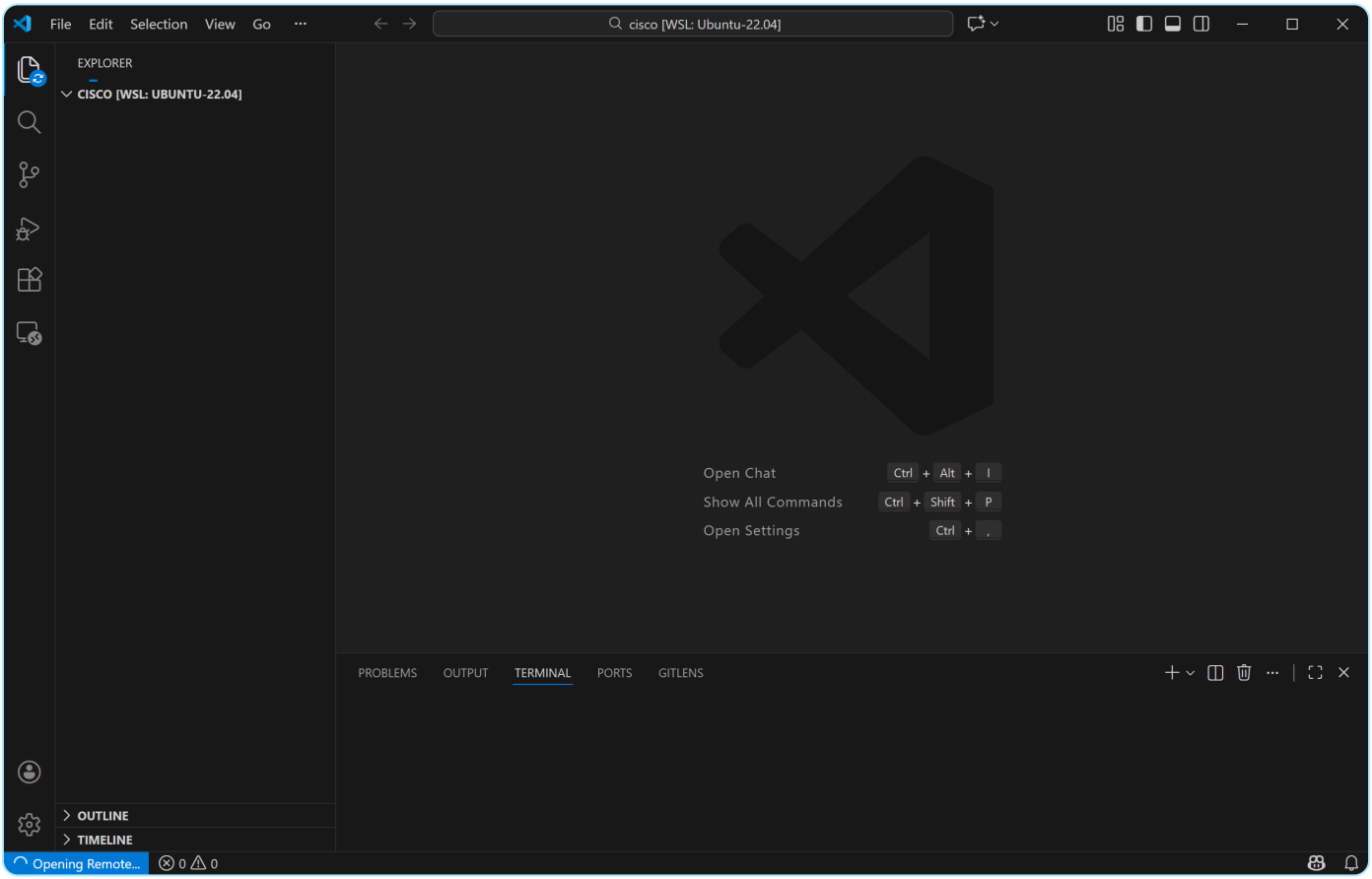
You're now ready to populate the files. Every subsequent step in this guide assumes your current working directory is `/home/cisco/nac-iosxe`.

Open Visual Studio Code

Now that the project structure exists, you'll open it in VS Code to edit the files.

WHAT IS VS CODE?

Visual Studio Code, commonly known as VS Code, is a free, lightweight, yet powerful source code editor developed by Microsoft. It runs on Windows, macOS, and Linux and has become one of the most popular code editors among developers and IT professionals.



Key features that make VS Code ideal for Network as Code:

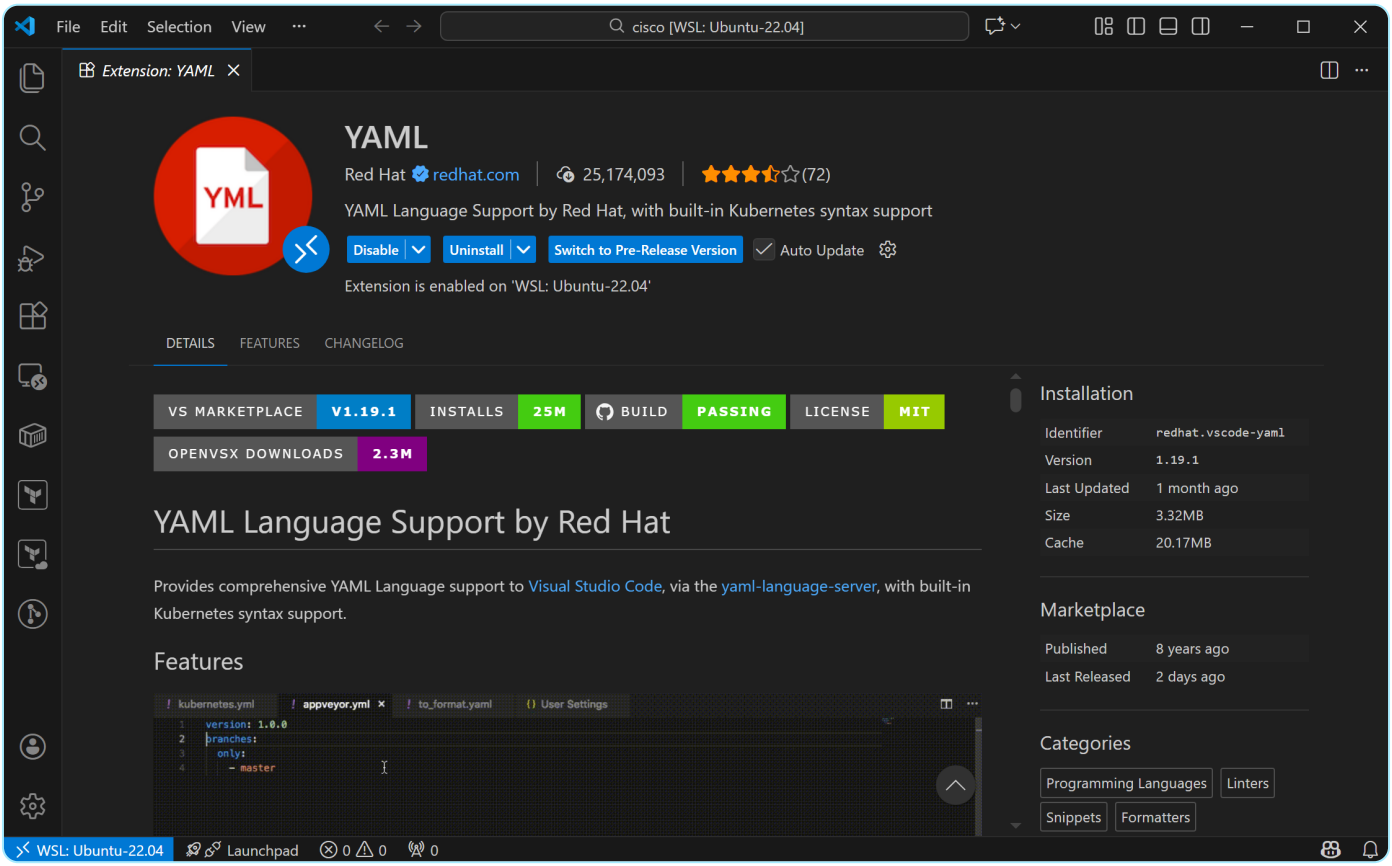
- **Syntax highlighting** - Colors and formats YAML files for easy readability
- **IntelliSense** - Provides intelligent code completion and suggestions
- **Integrated terminal** - Run commands directly within the editor
- **Extensions** - Add functionality like YAML validators and schema support
- **Git integration** - Built-in version control for tracking changes
- **Multi-file editing** - Work with multiple configuration files simultaneously
- **File explorer** - Easy navigation through project folders and files

Copyright © Cisco

YAML LINTING WITH REDHAT EXTENSION (PRE-INSTALLED)

Since Network as Code configurations are written in YAML, having proper syntax validation is essential. VS Code supports YAML linting through the **YAML extension by Red Hat**, which helps catch syntax errors and enforce best practices as you write your configuration files.

> **Pre-installed in your lab VM**
>
> This extension is already installed in your lab's VS Code - no action required.



This extension provides:

- **Real-time syntax validation** - Highlights errors as you type
- **Auto-completion** - Suggests valid YAML structures
- **Formatting** - Automatically formats your YAML files
- **Schema validation** - Can validate against predefined schemas

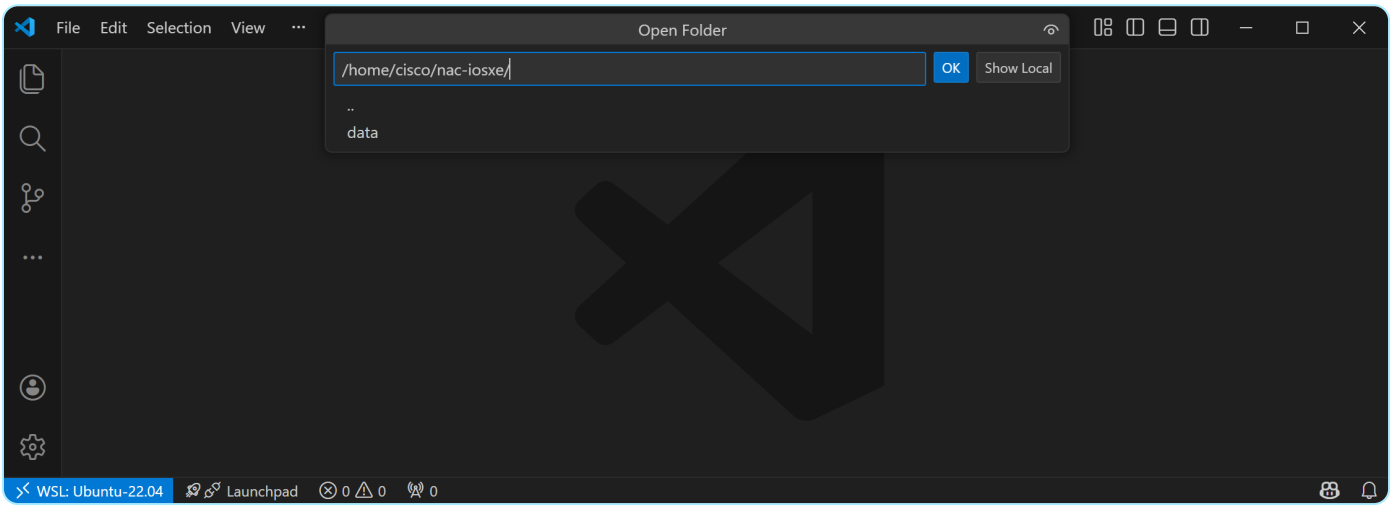
> **File Extension**
>
> The YAML extension recognizes only files ending with `.nac.yaml` as Network as Code YAML files. To benefit from the VS Code extension, ensure your configuration files end with `.nac.yaml`.

Copyright © Cisco

OPEN THE PROJECT FOLDER

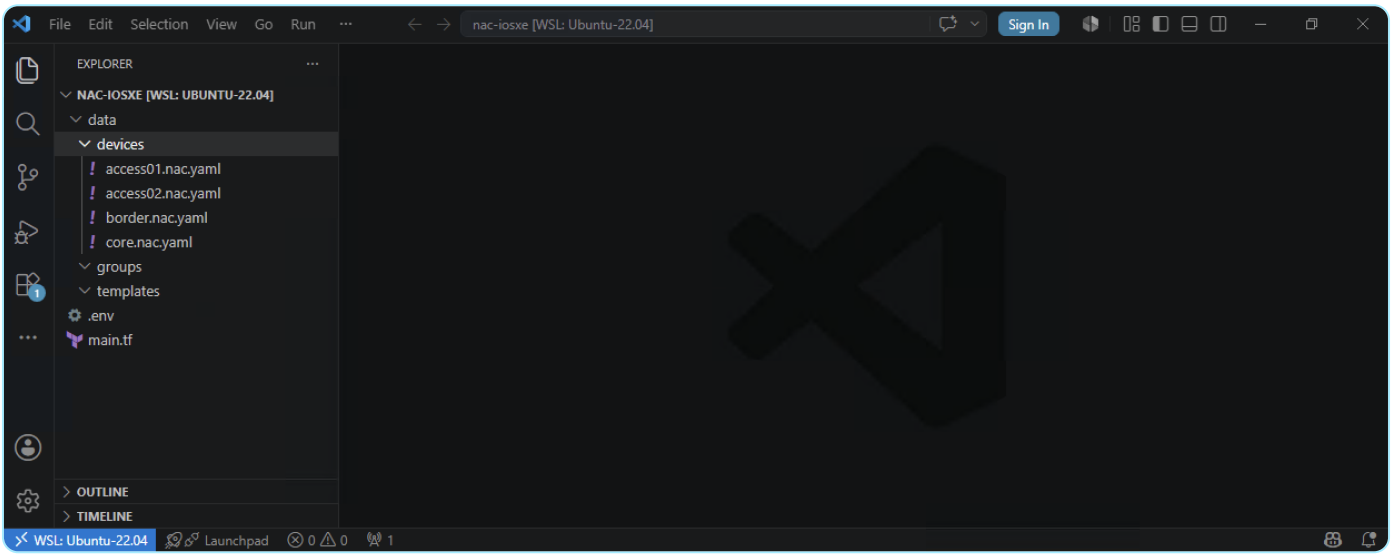
To open the folder in VS Code:

1. Double-click the **Visual Studio Code Code** icon on the Windows desktop
2. Click **File → Open Folder**
3. In the address bar, type or paste: `/home/cisco/nac-iosxe`
4. Click **OK**



> **VS Code automatically connects to WSL (pre-configured)**
>
> The **WSL** VS Code extension is pre-installed, so VS Code connects to your Ubuntu environment automatically. You'll see a `WSL: Ubuntu-22.04` indicator in the bottom-left corner confirming the connection - files you edit in VS Code are saved directly to the WSL filesystem.

VS Code will now open with your project folder, and you'll see the file explorer on the left showing your three configuration files.



Copyright © Cisco

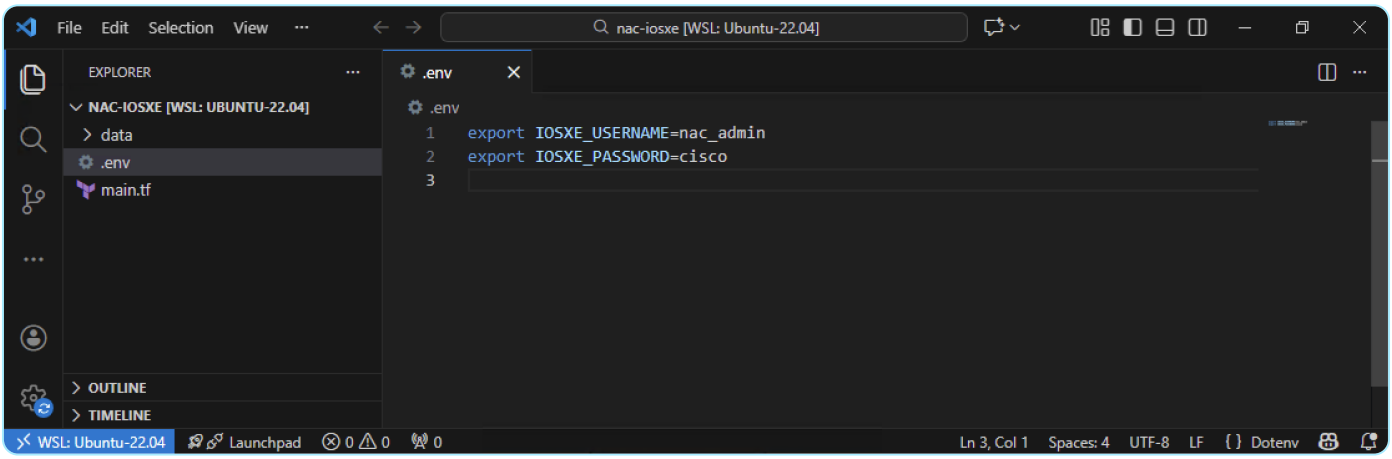
Edit .env file

Edit `.env` file containing the environment variables required by the Network as Code Terraform modules to connect to the Cisco IOS XE devices. This file stores your IOS XE credentials and connection details in a secure and reusable format:

FILE **.env**

```
export IOSXE_USERNAME=nac_admin
export IOSXE_PASSWORD=cisco
```

The figure below illustrates how to edit the `.env` file using Visual Studio Code:



LOAD THE ENVIRONMENT VARIABLES

After saving the file, switch to your **WSL Ubuntu terminal** and source it so the variables are available to Terraform:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
source ~/nac-iosxe/.env
```

You won't see any output - that's expected. The `export` statements silently place the variables into your shell session.

> **Never commit `.env` to a real repository**
>
> The `.env` file holds device credentials in plaintext. In this lab it's harmless (every attendee has the same throwaway lab switches), but in a real project add `.env` to your `.gitignore` **before** the first commit. If credentials ever land on a feature branch and are pushed, rotate them - deleting the file from history isn't enough.

> **If `source .env` fails with a weird error, check line endings**
>
> VS Code on Windows sometimes saves files with Windows-style (`CRLF`) line endings. WSL's Bash chokes on those in a sourced file. If `source .env` errors, run `dos2unix .env` in the WSL terminal and try again. You can also configure VS Code to always use LF - see the status-bar indicator at the bottom-right of the editor window.

Copyright © Cisco

Edit Terraform main.tf file

Next, edit a Terraform `main.tf` file with the following content. This file serves as the entry point for the Terraform configuration and defines the necessary resources and modules to interact with the IOS XE device:

FILE **main.tf**

```
module "iosxe" {
  source                   = "git::https://github.com/netascode/terraform-iosxe-nac-iosxe.git"
  yaml_directories         = ["data/"]
  write_model_file         = "model.yaml"
  write_default_values_file = "defaults.yaml"
}
```

Formatting matters

Terraform (`.tf`) files use braces `{}` and indentation for structure. Unlike YAML, Terraform is forgiving about whitespace, but consistent formatting makes your code readable. The snippets in this guide are formatted with `terraform fmt` ; if you type manually, run `terraform fmt` after to clean up.

Understanding the configuration:

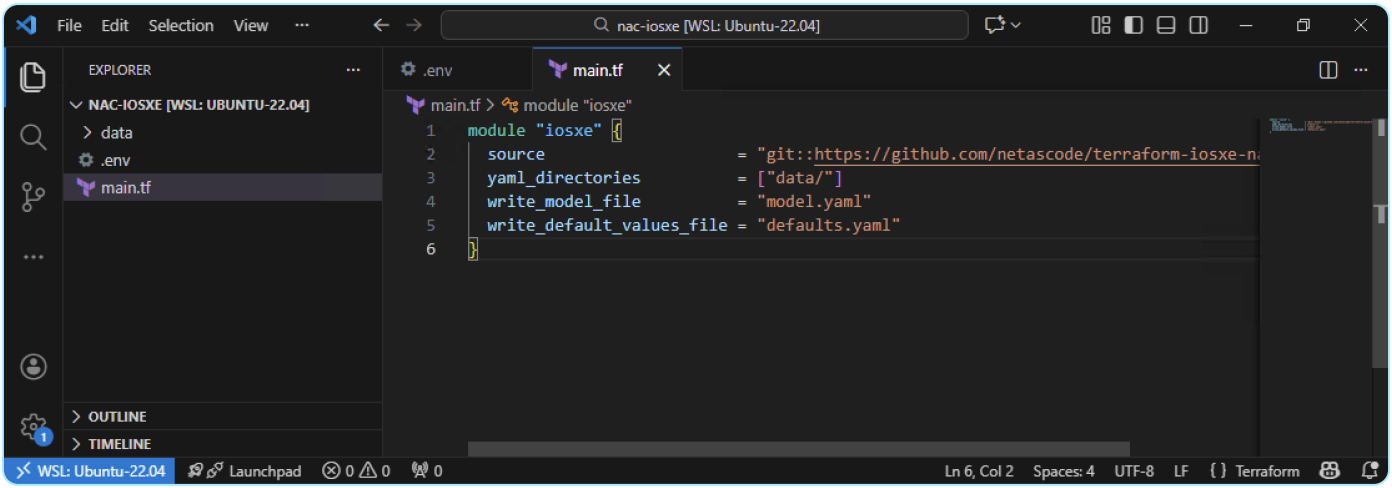
- `module "iosxe"` - declares a Terraform module named `iosxe` . Modules package reusable Terraform logic; you invoke them with `module.iosxe` .
- `source = "git::https://github.com/netascode/terraform-iosxe-nac-iosxe.git"` - tells Terraform where to fetch the module. This is the Cisco-maintained Network as Code module that translates your YAML into the low-level resource calls the `terraform-provider-iosxe` understands.
- `yaml_directories = ["data/"]` - tells the module which directories to scan for YAML. Every `*.nac.yaml` file inside `data/` is auto-discovered and merged. This is why you can split configurations across many files without wiring them up individually.
- `write_model_file = "model.yaml"` - after merging, the module writes the final merged data model to `model.yaml` . You'll use this file to debug variable substitution and as input to `nac-test` for post-deployment validation.
- `write_default_values_file = "defaults.yaml"` - similar, but for the default values the module applies when your YAML omits a field. Useful for understanding "where did this setting come from?"

Why no version pin?

For the lab we source directly from the module's default branch (`main`) to always get the latest schema and features. In production you should pin to a specific tag or commit (for example, `?ref=v0.12.3`) so every `terraform init` yields a reproducible build. Unpinned sources are convenient for experimentation but will occasionally break learners when upstream ships schema changes - pin once you ship.

The figure below illustrates the `main.tf` file in Visual Studio Code:

Copyright © Cisco



Populate the per-device files

Each per-device file registers one device with Network as Code – it declares the device's **name** (which every other YAML file will reference) and its **management IP address**. You'll add actual configuration to these files in later tasks.

Open `data/devices/core.nac.yaml` in VS Code and paste:

FILE **data/devices/core.nac.yaml**

```yaml
---
iosxe:
  devices:
    - name: core
      host: 198.18.130.10
```

Open `data/devices/border.nac.yaml` and paste:

FILE **data/devices/border.nac.yaml**

```yaml
---
iosxe:
  devices:
    - name: border
      host: 198.18.130.20
```

Open `data/devices/access01.nac.yaml` and paste:

FILE **data/devices/access01.nac.yaml**

```yaml
---
iosxe:
  devices:
    - name: access01
      host: 198.18.130.11
```

Open `data/devices/access02.nac.yaml` and paste:

FILE **data/devices/access02.nac.yaml**

```yaml
---
iosxe:
  devices:
    - name: access02
      host: 198.18.130.12
```

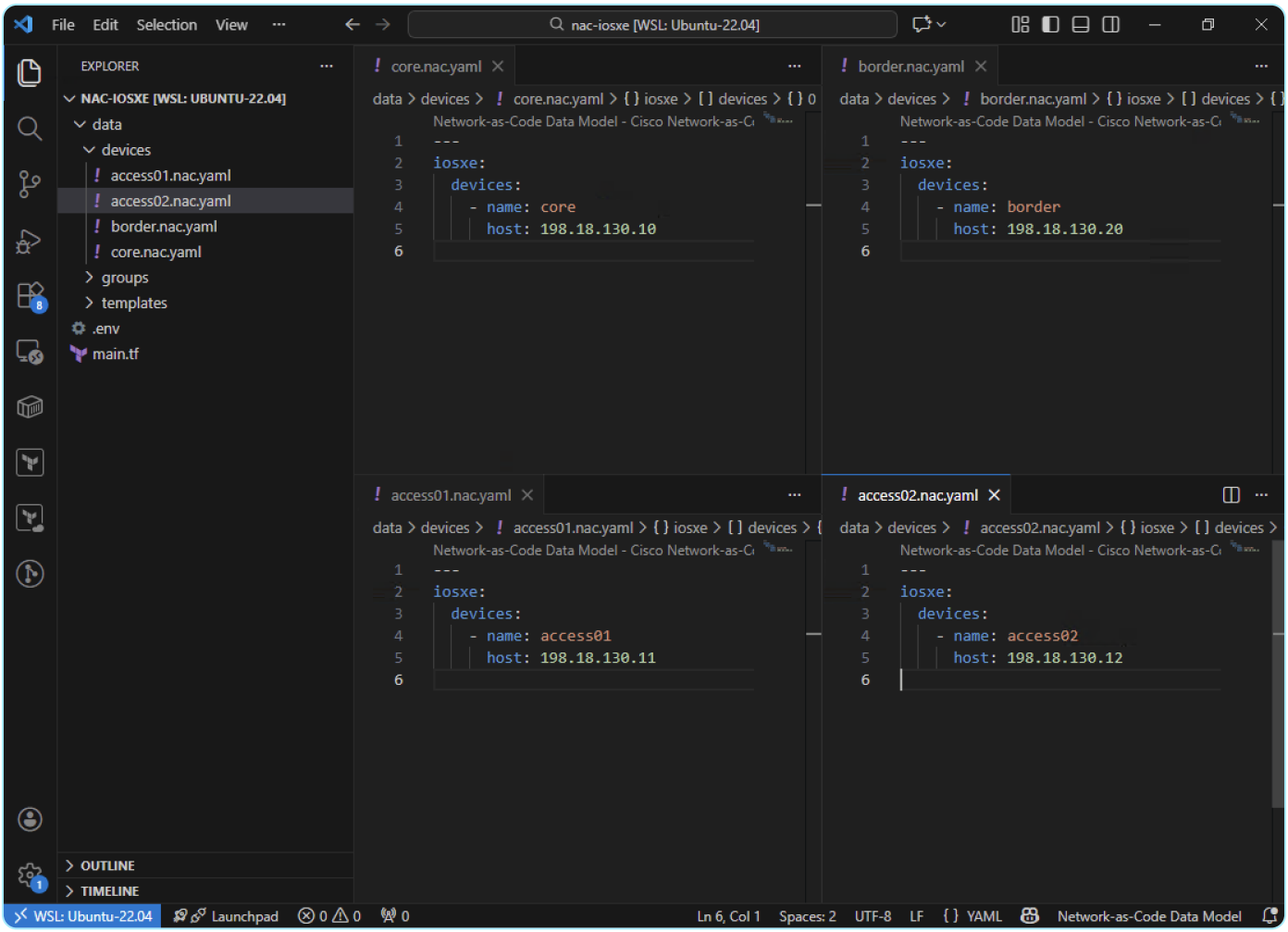

Copyright © Cisco

> **Indentation matters in YAML**
>
> YAML uses **spaces** (not tabs) and relies on consistent indentation to express structure. Each level of nesting should be exactly 2 spaces. Copy-pasting from this guide preserves the right indentation; if you type manually, watch for the red squiggles from the VS Code YAML extension - they point at exactly the line that's off.

What each key means:

- `---` - YAML document-start marker (optional but conventional).
- `iosxe:` - root key. Every IOS XE as Code YAML starts here.
- `devices:` - the top-level list of devices Network as Code manages.
- `name:` - unique identifier for the device. Other YAML files will match against this name.
- `host:` - management IP Network as Code connects to.

> **Why one device per file?**
>
> Network as Code merges all YAML files in `data/` into a single data model at load time. Because `devices` is a list, keeping each device in its own file makes it immediately obvious which file owns which device, and avoids the accidental-merge bugs that can happen when several files all append to the same list. You'll extend these files with real configuration starting in Task 05.



Copyright © Cisco

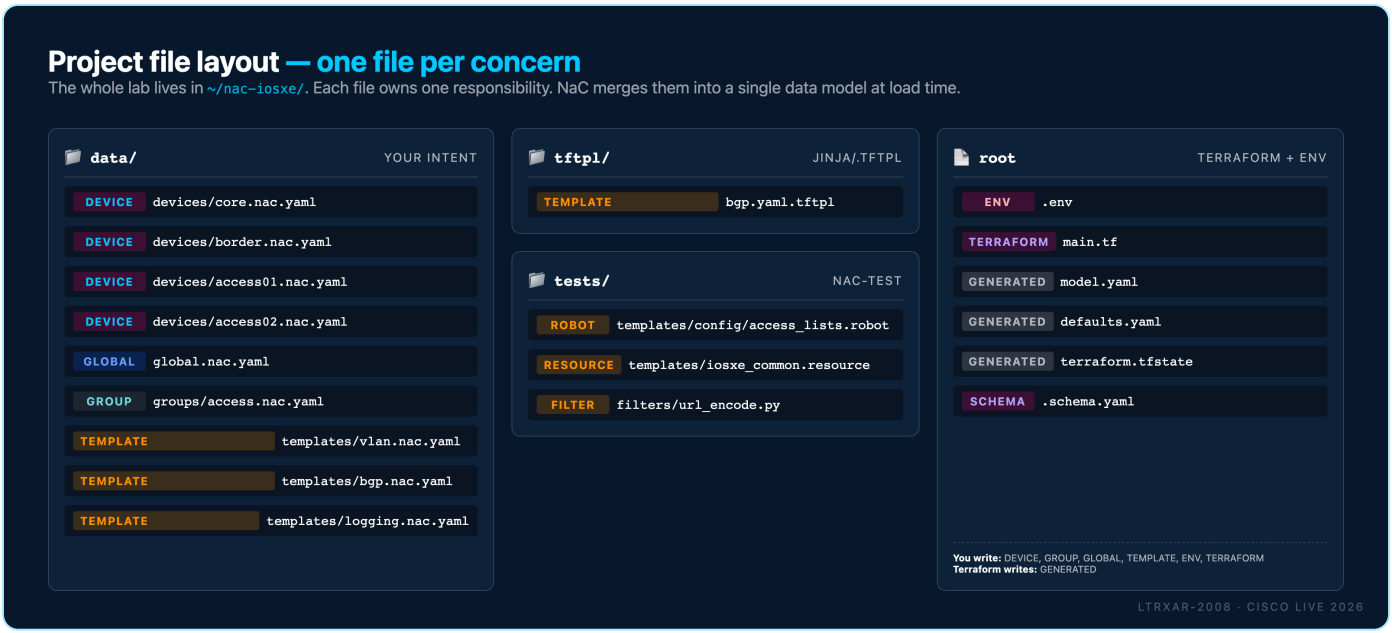
Auto-save is enabled

VS Code has auto-save enabled, so your files are saved a few seconds after you stop typing. At this point your project contains:

- `.env` - device credentials and protocol
- `main.tf` - Terraform module configuration
- `data/devices/*.nac.yaml` - four per-device skeleton files

How the project will grow

Every subsequent task in this lab will either extend the per-device files or introduce a new file type under `data/`, `tftpl/`, or `tests/`. Here's the **final** layout you'll end up with by the time you reach Task 11 - refer back to this any time you lose track of which file owns what:



What you've accomplished

- Set up the project directory `/home/cisco/nac-iosxe` with a `data/` subfolder
- Created `.env` with your IOS XE credentials and the NETCONF protocol setting
- Created `main.tf` pointing to the Network as Code module
- Registered all four lab devices in their own per-device YAML files

Tools introduced:

- **VS Code** (with the Red Hat YAML extension) - YAML-aware editing, validation, linting
- **WSL Ubuntu** - the Linux environment where you'll run Terraform and other CLI tools

In the next task, you'll deploy your first piece of configuration - a login banner - to all four devices using Terraform.

Task 01 - SSH to the lab devices

Task 03 - Global Configuration

Copyright © Cisco

Task 03 - Global configuration

Estimated Time to Complete: ~25 minutes

In this task you'll use IOS XE as Code **global configuration** to apply settings across all devices at once. A login banner is the example - you'll see how global settings eliminate the need to repeat the same configuration on each device individually.

What you'll learn

By the end of this task you will have:

- Written a **global** Network as Code configuration file that applies to every registered device
- Run the full Terraform workflow end-to-end: `init` → `plan` → `apply`
- Verified the deployed banner by SSH-ing to each device

Global configuration

Global configurations define network-wide settings that apply to all devices unless explicitly overridden at the device group or device level. The configuration precedence hierarchy works as follows:

1. **Device** (highest precedence) - device-specific overrides
2. **Device Group** (medium precedence) - role or location-specific settings
3. **Global** (lowest precedence) - organization-wide defaults

By placing the banner in the `global` section, it will automatically apply to all devices listed in your configuration, ensuring consistency without duplication.

Create the global configuration file

First, create the global configuration file using your **WSL Ubuntu terminal**:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
touch ~/nac-iosxe/data/global.nac.yaml
```

> **Alternative: File creation with VS Code**
>
> You can also create the file using VS Code by clicking on the `data/` folder in the Explorer sidebar, then on the *new file* icon next to the `NaC-IOSXE` folder name, or by **right-clicking** the `data/` folder and selecting **New File**.
>
> Throughout this lab guide, you will use the `touch` command in WSL to create files, but feel free to use VS Code if you prefer a graphical interface.

Then open `data/global.nac.yaml` in VS Code and add the following content. Notice how the banner is defined once in the `global` section and will be applied to every device registered in the per-device files you created in Task 02:

FILE **data/global.nac.yaml**

```yaml
---
iosxe:
  global:
    configuration:
      banner:
        login: "Welcome to the IOS XE as Code lab!"
```


Copyright © Cisco

Key elements explained:

- `---` - YAML document start marker
- `iosxe:` - Root key indicating IOS XE specific configuration
- `global:` - Defines configurations that apply to all devices
- `configuration:` - Contains the actual configuration settings
- `banner:` - Specifies banner configurations (note: singular, not "banners")
- `login:` - The login banner text displayed to users before they log in to the device

> **Separation of concerns**
>
> The global configuration lives in its own file (`global.nac.yaml`), separate from the per-device files. This modular layout keeps "what applies to everyone" distinct from "what applies to one device" - NaC merges them all automatically when it reads the `data/` directory.

The figure below illustrates how to create the `data/global.nac.yaml` file with Visual Studio Code:

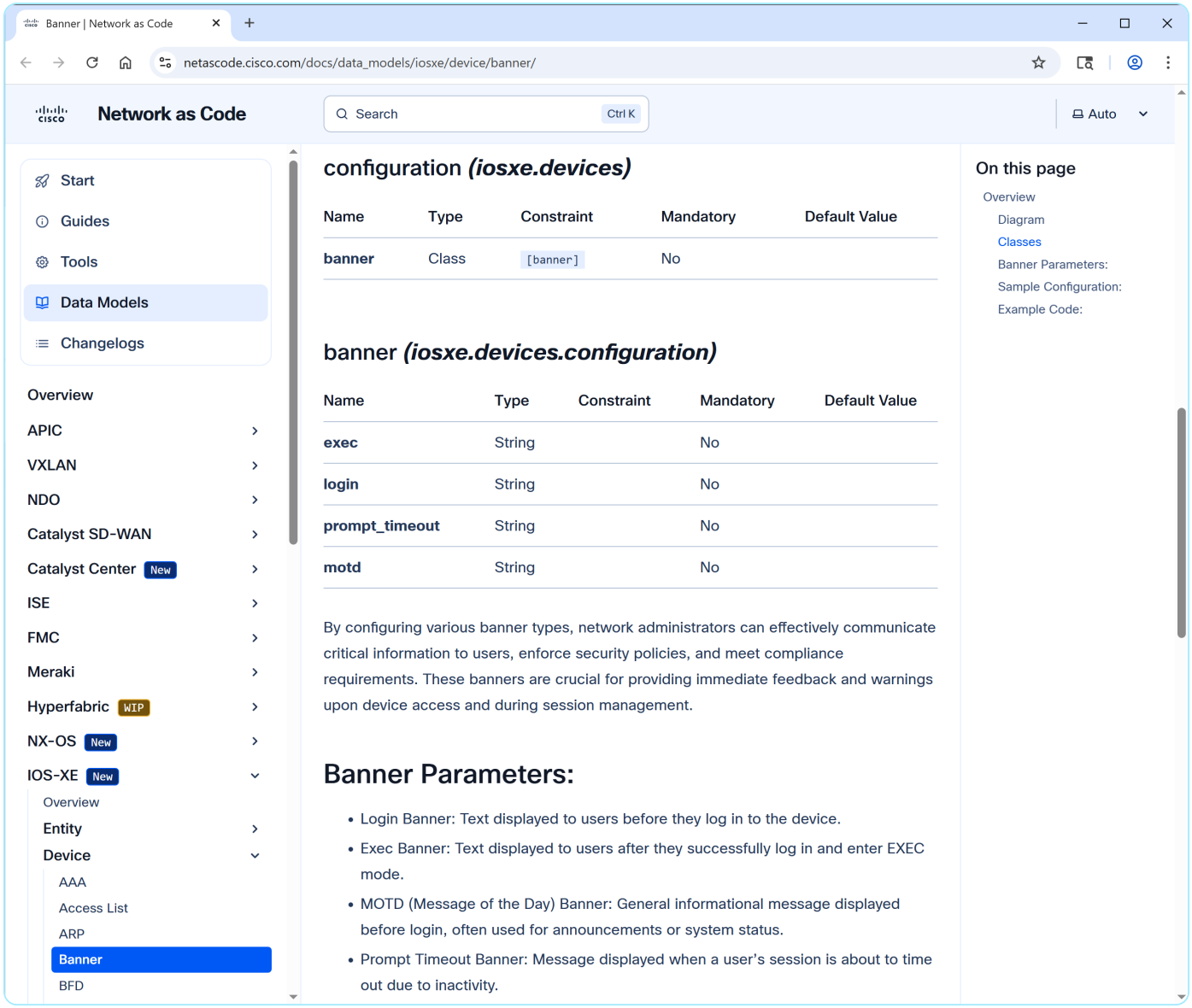


Documentation reference

But how do you know what configuration options are supported, and what the correct YAML structure is?

The data model documentation is published on the Network as Code website. Specifically, the banner configuration is described here: IOS XE Banner Configuration.

Copyright © Cisco



You can refer to this documentation at any time for more details on available configuration options, data types and guides. The curated configuration examples provide excellent references to help you create your own configurations.

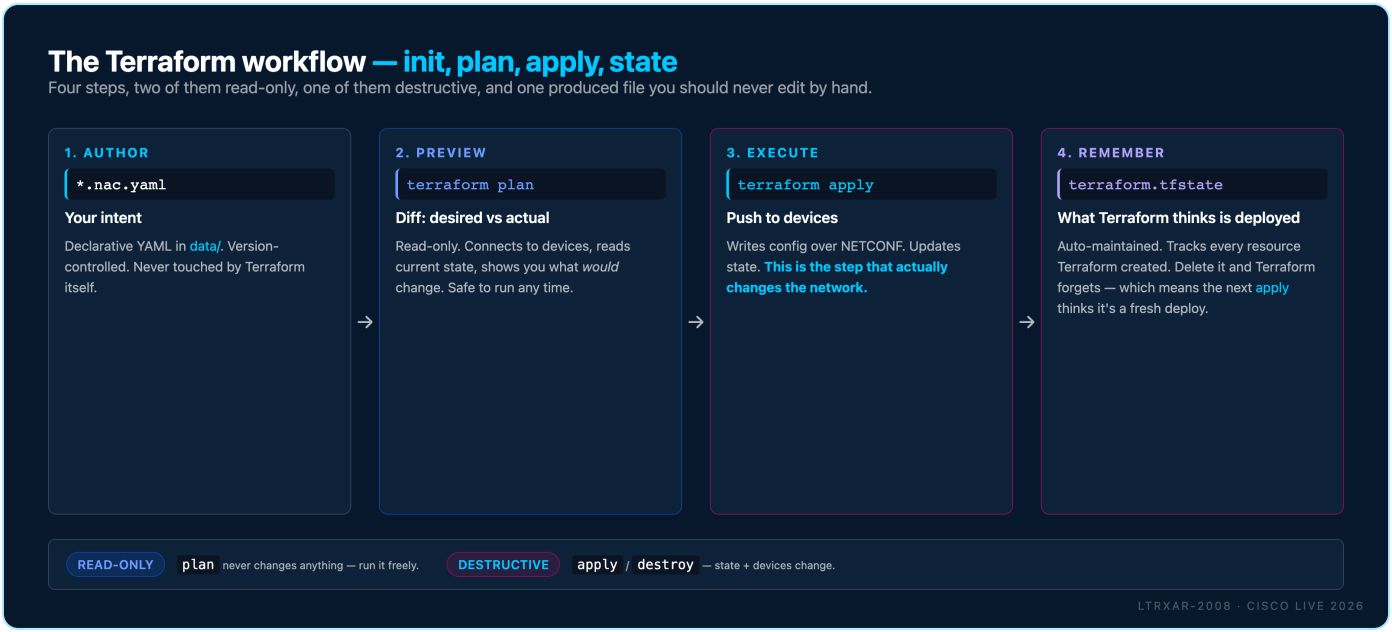
Applying configuration with Terraform CLI

Now that you've created your configuration files, it's time to deploy them to your network devices using Terraform. Terraform follows a simple three-step workflow that ensures safe and predictable infrastructure changes.

Terraform workflow

Terraform uses a declarative approach where you define the desired state (in your YAML files), and Terraform figures out how to achieve that state. There are four moving parts:

Copyright © Cisco



1. **Author** (`*.nac.yaml`) - the YAML you write. Version-controlled, never touched by Terraform itself.
2. **Preview** (`terraform plan`) - read-only. Connects to devices, reads current state, shows you what *would* change. Safe to run any time.
3. **Execute** (`terraform apply`) - writes configuration over NETCONF and updates state. **This is the step that actually changes the network.**
4. **Remember** (`terraform.tfstate`) - auto-maintained. Tracks every resource Terraform has created, so subsequent plans know what's already there.

Of the three commands, only `apply` (and its counterpart `destroy`) change anything on the wire. `plan` is a read-only diff and can be run freely.

STEP 1: IN WSL (UBUNTU) AND NAVIGATE TO YOUR PROJECT

In Windows Subsystem for Linux (WSL) terminal, navigate to your project directory:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
cd ~/nac-iosxe
```

> **Integrated Terminal in VS Code**
>
> You can also open the WSL terminal directly in VS Code by going to the menu and selecting **Terminal > New Terminal**. This opens a terminal at the root of your project, making it easy to run Terraform commands without switching windows.
>
> Whenever this guide mentions running commands in WSL, you can use either the standalone WSL terminal or the integrated terminal in VS Code.

List the files in your directory:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
tree -a
```


Copyright © Cisco

You should see your project structure:

OUTPUT **Expected output**

```
cisco@wkst1:~/nac-iosxe$ tree -a
.
├── .env
├── data
│   ├── devices
│   │   ├── access01.nac.yaml
│   │   ├── access02.nac.yaml
│   │   ├── border.nac.yaml
│   │   └── core.nac.yaml
│   ├── global.nac.yaml
│   ├── groups
│   └── templates
└── main.tf

4 directories, 7 files
cisco@wkst1:~/nac-iosxe$
```

STEP 2: LOAD ENVIRONMENT VARIABLES FROM .ENV FILE

Before running Terraform, you need to load the credentials from your `.env` file. Your `.env` file contains simple key-value pairs (e.g. `IOSXE_USERNAME=nac_admin`).

Convert the file to Unix format to avoid encoding issues

You might encounter a situation where you have edited the `.env` file on Windows, causing it to have Windows-style line endings (CRLF). Run the following command in your **WSL Ubuntu terminal** to convert the `.env` file to Unix format: `dos2unix .env`

To load these variables and make them available to Terraform, use this simple command:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
source .env
```

Verify the variables are loaded:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
env | grep IOSXE
```

You should see the environment variables displayed:

OUTPUT **Expected output**

```
cisco@wkst1:~/nac-iosxe$ env | grep IOSXE
IOSXE_USERNAME=nac_admin
IOSXE_PASSWORD=cisco
cisco@wkst1:~/nac-iosxe$
```

These credentials allow Terraform to authenticate with your IOS XE devices over NETCONF.

Making Environment Variables Persistent:

Environment variables exported in your current shell session are not persistent - they disappear when you close the terminal. If you exit WSL and later open a new session, you must export them again.

To avoid manually exporting variables every time you open WSL, you can add the export command to your `~/.bashrc` file. This file runs automatically whenever you start a new bash session, so your environment variables will be loaded automatically.

Copyright © Cisco

To make the export permanent, add it to your bashrc

TERMINAL · WSL/Ubuntu cisco@wkst1:~/nac-iosxe$

```
echo 'source ~/nac-iosxe/.env' >> ~/.bashrc
```

This appends the source command to your `~/.bashrc` file. Now every time you open WSL, your IOS XE credentials will be automatically loaded from the `.env` file.

STEP 3: VERIFY NETCONF REACHABILITY

Before running Terraform, confirm the NETCONF subsystem is up on one of the lab devices. NETCONF listens on TCP/830 by default, and the simplest way to check it from WSL is a one-shot SSH handshake against the `netconf` subsystem - the device replies with an XML `<hello>` message containing its supported capabilities.

Test against **access01** (`198.18.130.11`):

TERMINAL · WSL/Ubuntu cisco@wkst1:~/nac-iosxe$

```
ssh -s -p 830 -o StrictHostKeyChecking=no $IOSXE_USERNAME@198.18.130.11 netconf
```

When prompted, enter the password (`cisco`).

What each flag does:

- `-s` - request an SSH **subsystem** (rather than an interactive shell).
- `-p 830` - NETCONF's default port.
- `-o StrictHostKeyChecking=no` - skip the first-connection host-key prompt (lab-only; never use this in production).
- `netconf` - the subsystem name.

OUTPUT **Expected output (truncated)**

```
<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:base:1.1</capability>
    <capability>urn:ietf:params:netconf:capability:writable-running:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:rollback-on-error:1.0</capability>
    ...
  </capabilities>
  <session-id>123</session-id>
</hello>
]]>]]>
```

If you see the `<hello>` XML with a capability list, NETCONF is reachable and your credentials work. Press **Ctrl+C** to exit - Terraform will manage its own NETCONF sessions from here on.

> **No `candidate:1.0` capability yet - that's expected**
>
> Notice that the highlighted `writable-running:1.0` line appears, but `candidate:1.0` and `confirmed-commit:1.1` are **not** in the capability list. That's because the `netconf-yang feature candidate-datastore` CLI was intentionally left unconfigured on the lab devices (see Task 01). You'll enable it in **Step 4** below, and the capability list will then advertise `candidate:1.0` instead.

STEP 4: ENABLE THE NETCONF CANDIDATE DATASTORE ON EACH DEVICE

While the base NETCONF subsystem is already running, the **candidate datastore** capability has not yet been enabled. This is the missing CLI line that activates the `<candidate>` -> `<commit>` -> `<discard-changes>` flow Terraform relies on for transactional, all-or-nothing pushes (see the NETCONF datastore diagram in Task 01). Without it, NETCONF on IOS XE falls back to writing directly to `running` , which loses the "all RPCs commit together or none of them do" guarantee.

Copyright © Cisco

Enable it on **all four** lab devices using **Solar-PuTTY**. Open Solar-PuTTY, connect to each of `core`, `border`, `access01`, `access02` as `nac_admin` / `cisco`, and run the same three commands on each:

DEVICE CLI · PuTTY **core, border, access01, access02**

```
configure terminal
netconf-yang feature candidate-datastore
end
write memory
```

Repeat for all four devices before moving on. Each box should accept the commands and save without errors.

> ⚠ **Wait 1-2 minutes before testing NETCONF**
>
> Enabling `candidate-datastore` causes the NETCONF subsystem to restart internally. Until it finishes reinitializing (typically 1-2 minutes), SSH connections to port 830 will be refused with a **"Connection refused"** error. If you see this, wait a minute and try again - the device is not broken, the NETCONF process is simply restarting.

Verify the capability now appears. Re-run the Step 3 NETCONF handshake against `access01`:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
ssh -s -p 830 -o StrictHostKeyChecking=no $IOSXE_USERNAME@198.18.130.11 netconf
```

OUTPUT **Expected output (after enabling candidate-datastore)**

```
<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:base:1.1</capability>
    <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:confirmed-commit:1.1</capability>
    <capability>urn:ietf:params:netconf:capability:rollback-on-error:1.0</capability>
    ...
  </capabilities>
  <session-id>124</session-id>
</hello>
]]>]]>
```

The `candidate:1.0` and `confirmed-commit:1.1` capability lines are what tells you the feature is now active. Press **Ctrl+C** to exit. Terraform will use this datastore for every subsequent `apply`.

STEP 5: INITIALIZE TERRAFORM

Initialize your Terraform project to download the required Network as Code module:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

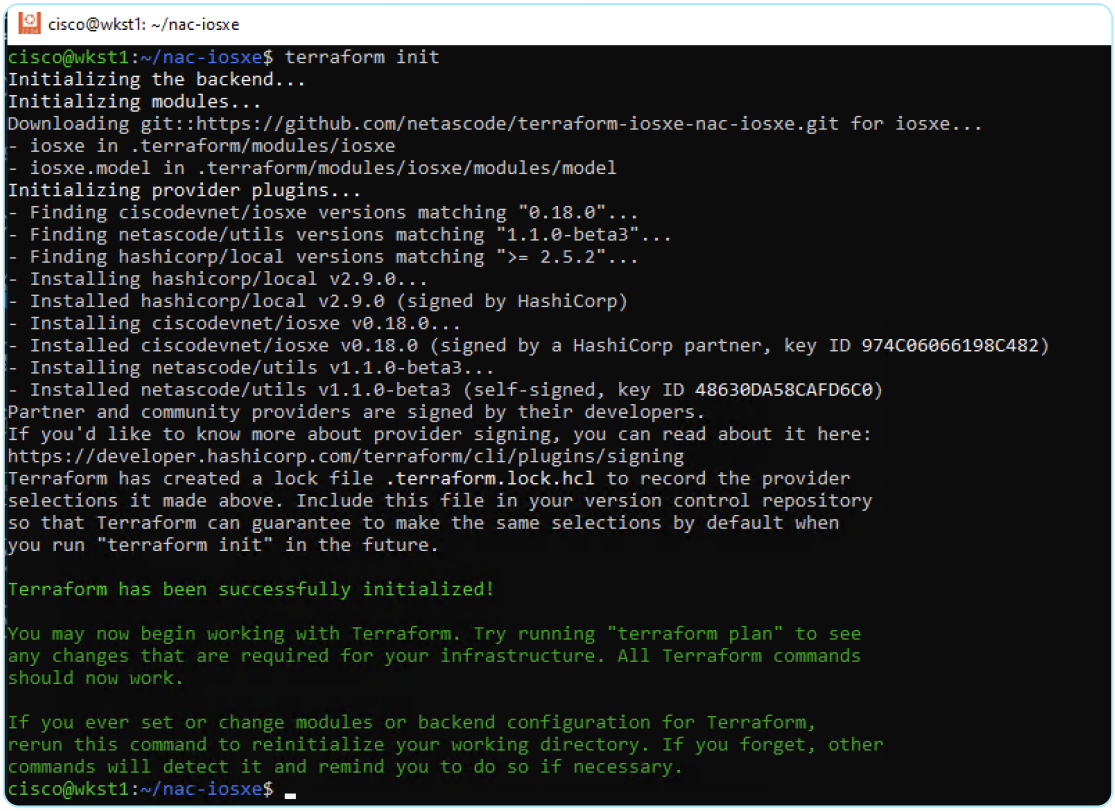
```
terraform init
```

What happens during initialization:

- Terraform reads your `main.tf` file
- Downloads the `netascode/nac-iosxe` module from GitHub (as defined in `main.tf`)
- Creates a `.terraform` directory with downloaded modules
- Creates a `.terraform.lock.hcl` file to lock module versions

Expected output:

Copyright © Cisco



Copyright © Cisco

What the `terraform init` output tells you (annotated)

First-time Terraform users can find the init output intimidating because Terraform is chatty - it narrates everything it does. Here's a typical healthy run, annotated:

OUTPUT **Annotated terraform init output**

```
Initializing the backend...             ← Reading backend config from main.tf.

Initializing modules...                 ← Found `module "iosxe"` in main.tf.
Downloading git::https://github.com/
  netascode/terraform-iosxe-nac-iosxe.
  git for iosxe...                      ← Cloning the Network as Code module from GitHub.
- iosxe in .terraform/modules/iosxe     ← Stored locally under .terraform/.
- iosxe.model in .terraform/modules/
  iosxe/modules/model                   ← Submodule that builds the data model.

Initializing provider plugins...        ← Now downloading the providers
                                          the Network as Code module declares.
- Finding ciscodevnet/iosxe versions
  matching "0.18.0"...
- Finding netascode/utils versions
  matching "1.1.0-beta3"...
- Finding hashicorp/local versions
  matching ">= 2.5.2"...
- Installing ciscodevnet/iosxe
  v0.18.0...
- Installed ciscodevnet/iosxe v0.18.0
  (signed by a HashiCorp partner)       ← Signature-verified. Good.
- Installing netascode/utils
  v1.1.0-beta3...
- Installed netascode/utils
  v1.1.0-beta3 (self-signed)            ← Helper provider for YAML merging.
- Installing hashicorp/local v2.9.0...
- Installed hashicorp/local v2.9.0
  (signed by HashiCorp)                 ← Writes rendered files to disk.

Terraform has created a lock file
  .terraform.lock.hcl                   ← Next init will pin these versions.

Terraform has been successfully
  initialized!                          ←  You're ready to run `plan`.
```

Things that look scary but are normal:

- **"Downloading git::https://..."** - Terraform is pulling the Network as Code module fresh. Happens every `init` unless the module's already cached.
- **"Finding ... versions matching"** - version-constraint resolution. Lines like "Reusing previous version of ciscodevnet/iosxe" on subsequent runs mean the lock file is doing its job.
- **"self-signed"** - the `netascode/utils` provider is published by the NaC community, not HashiCorp or a verified partner. This is expected.
- **"Terraform has created a lock file"** - this is good. `.terraform.lock.hcl` pins provider versions. Commit it alongside `main.tf` in a real project.

Things that actually mean something's wrong:

- **"Could not download module"** - internet access broken, or the module URL is wrong, or the referenced ref/tag doesn't exist.
- **"Failed to install provider"** - provider registry unreachable or the version constraint can't be satisfied. Re-run `terraform init`; if it keeps failing, check the version constraints in the module source.
- **"Backend configuration changed"** - not applicable for this lab (we use the default local backend), but in CI/CD projects with remote state you'll see this if someone reconfigured the backend and you need to migrate state.

Copyright © Cisco

> **Internet Connection Required**
>
> As shown in this lab, the `terraform init` step requires an active internet connection to download the necessary modules from GitHub. In production environments, you may want to set up a private module registry or use a local mirror to avoid dependency on external connectivity.

> **Initialization Only Once**
>
> You only need to run `terraform init` once per project. To add or update modules later, you can run it again to download any additional dependencies. To upgrade to a newer version of a module or provider in a production environment, you would update the version reference in your `main.tf` (e.g. change `?ref=v0.12.3` to `?ref=v0.13.0`), then run `terraform init -upgrade` to pull the new version. However, in this lab environment, there is no need to change version references or run `terraform init -upgrade`.

STEP 6: PREVIEW CHANGES WITH TERRAFORM PLAN

Before applying any changes to the lab devices, preview what Terraform will do:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

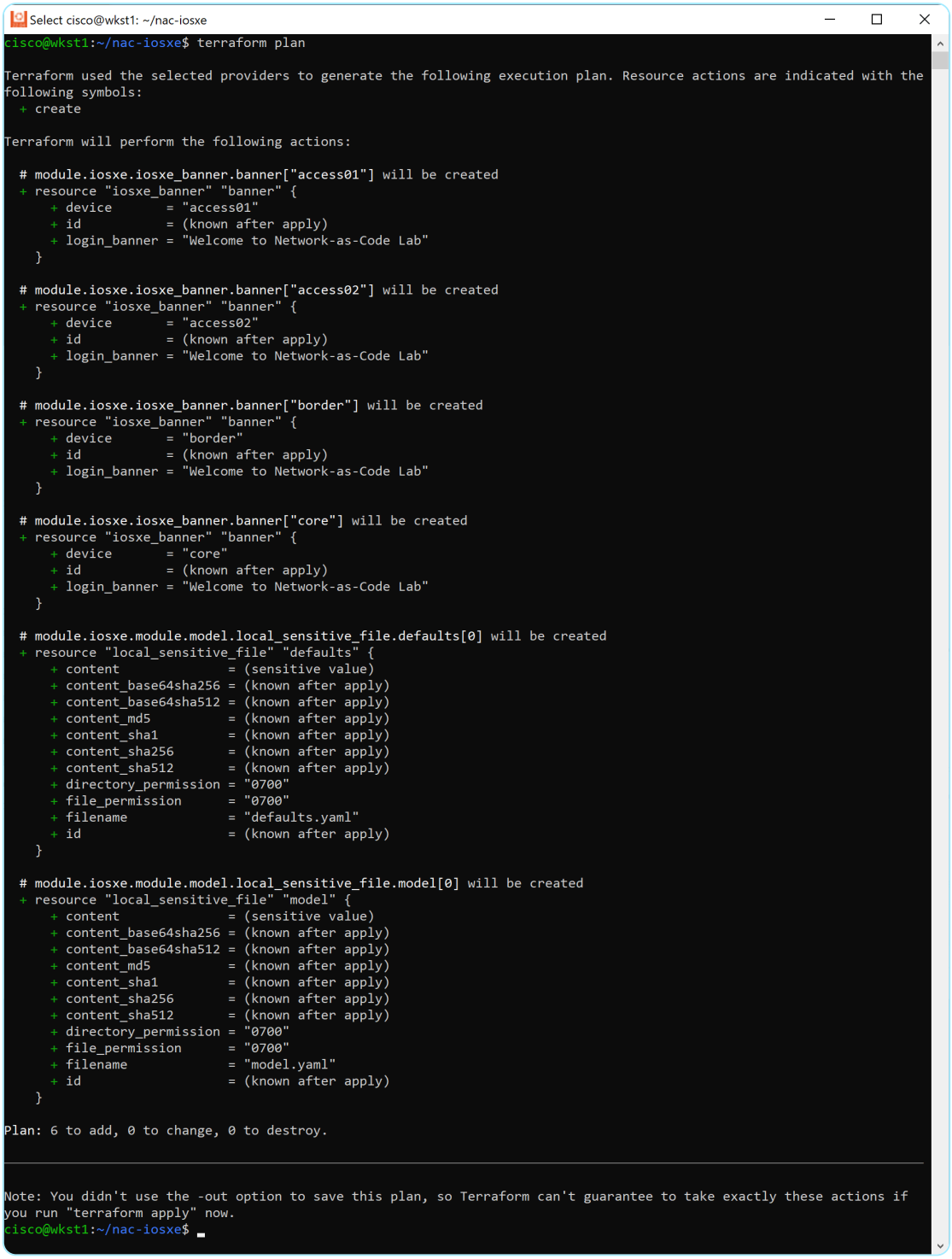
```
terraform plan
```

What Terraform plan does:

- Reads your `data/*.yaml` configuration files
- Connects to your IOS XE devices (using credentials from environment variables)
  - Checks the current state of all previously applied resources
  - This is skipped on the first run since no resources have been created yet
- Compares desired state (YAML) vs. current state (device configuration)
- Shows you what will be added, changed, or deleted

Expected output:

Copyright © Cisco



Review the plan carefully to ensure Terraform will make the changes you expect. This is your safety check!

In this case, you will configure the login banner on all four devices. Terraform will create a resource for each banner on each device. This is indicated by the `+` signs in the plan output.

Copyright © Cisco

What are the `local_sensitive_file` resources in the plan?

You'll also see Terraform planning to create `local_sensitive_file.model` and `local_sensitive_file.defaults`. These are **not** device configurations - they're local files the NaC module writes to disk based on the `write_model_file` and `write_default_values_file` settings in your `main.tf`. The module uses `local_sensitive_file` (rather than plain `local_file`) because the merged model may contain device credentials or other sensitive data that shouldn't appear in Terraform's log output. The resulting files (`model.yaml` and `defaults.yaml`) are useful for debugging variable substitution and as input to `nac-test` (Task 11), but they don't affect what gets pushed to the devices.

STEP 7: APPLY CONFIGURATION TO DEVICES

If the plan looks good, apply the configuration:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform apply
```

Terraform will show you the plan again and ask for confirmation:

OUTPUT **Expected output**

```
...
Plan: 6 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

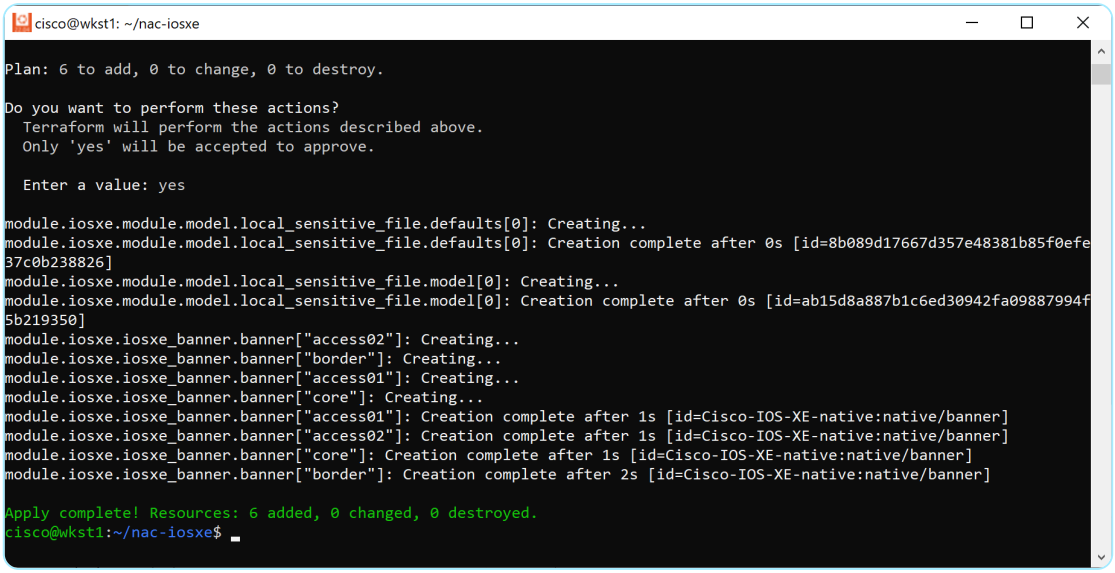
Type `yes` and press Enter to proceed.

What happens during apply:

- Terraform opens a NETCONF session to each device over SSH (TCP/830)
- Uses the Network as Code modules and Terraform IOS XE provider to translate your YAML configuration into YANG data and writes it to the device's `candidate` datastore over NETCONF
- Issues `<commit>` so the candidate config is atomically swapped into `running`
- Applies the commands to the devices
- Tracks the applied state in `terraform.tfstate` file

Expected output:

Copyright © Cisco



Automate approval with `-auto-approve`

To skip the confirmation prompt, add the `-auto-approve` flag:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform apply -auto-approve
```

Useful in CI/CD pipelines (Tasks 13–15) where the job runs unattended. The plan + apply still happen; only the "Do you want to proceed? yes/no" prompt is skipped.

...but not everywhere

Do **not** run `terraform apply -auto-approve` against production devices from a developer laptop. The interactive "yes/no" prompt is the last line of defence against a `plan` that accidentally proposes destroying a loopback, re-shutting an interface, or deleting every route-map because someone renamed a YAML key.

In a CI/CD pipeline the equivalent safeguard is different and higher-leverage: protected branches + merge-request review gates (Task 15). `-auto-approve` is safe **inside** a pipeline because the pipeline itself can only run after a reviewed merge to the target branch.

Rule of thumb: interactive shell → always confirm. Pipeline job behind a reviewed merge → `-auto-approve` is fine.

Skipping the Plan Step

As shown above, `terraform apply` first generates and displays the plan before asking for confirmation to proceed with the changes. This makes it possible to skip the previous redundant `terraform plan` step when performing `terraform apply` manually.

However, when automating with CI/CD pipelines, you can rather save the plan output to a file and supply it to `terraform apply` for non-interactive execution.

Copyright © Cisco

Want the config to survive a device reboot? Set `save_config = true`

By default, Network as Code writes to the **running configuration** only. If the device reboots, anything Terraform pushed is lost unless it was explicitly saved to `startup-config`. This is usually fine in the lab (devices don't reboot mid-session), but it's a real production footgun.

The module exposes a flag for this:

FILE **main.tf**

```
module "iosxe" {
  source      = "git::https://github.com/netascode/terraform-iosxe-nac-iosxe.git"
  # ... other arguments ...
  save_config = true     # persist to startup-config after every apply
}
```

With `save_config = true`, the module instantiates an `iosxe_commit` resource that executes `write memory` (via RESTCONF/NETCONF) after config apply - so subsequent reboots boot with the configuration intact.

For this lab you can leave it at the default (`false`) - the lab devices don't reboot. For your own devices at home, turn it on before you walk away from the terminal.

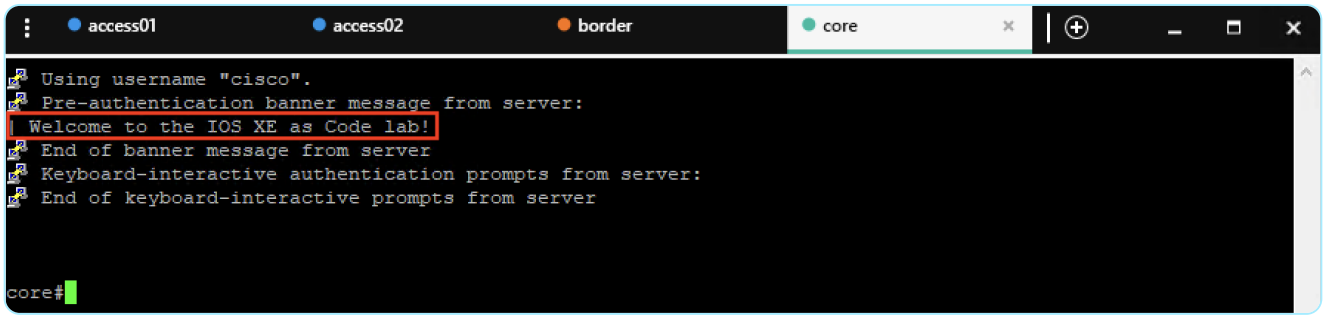
STEP 8: VERIFY THE GLOBAL CONFIGURATION

After Terraform completes successfully, verify the banner was applied to **all devices**. Because you used **global configuration**, the banner should be deployed to all four switches automatically.

Open Solar-PuTTY and connect to each switch:

1. Open **Solar-PuTTY** from your desktop
2. Connect to the **core** switch first
3. Verify that the pre-authentication banner appears upon ssh connection (shown below)
4. Repeat for **border** router, then the **access01** and **access02** switches

Upon successful config deployment, you should see the following banner message:



Additionally, you can also verify the banner configuration by examining the running configuration. Once connected to each switch, run the following command:

DEVICE CLI · PuTTY **core, border, access01, access02**

```
show run | include banner
```

OUTPUT **Expected Output**

Copyright © Cisco

```
core#show run | include banner
banner login ^CWelcome to the IOS XE as Code lab!^C
core#
```

The `^C` characters represent control characters used by IOS XE to delimit the banner text. The important part is that you see your configured text in the output.

What you should observe:

- The banner appears on the **core** switch (198.18.130.10)
- The banner appears on the **border** switch (198.18.130.20)
- The banner appears on the **access01** switch (198.18.130.11)
- The banner appears on the **access02** switch (198.18.130.12)

Check all four in one shot (WSL one-liner)

If you don't want to open Solar-PuTTY four times, this loop hits every managed device and prints the banner line from each running config. Paste it into your WSL terminal:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
for ip in 198.18.130.10 198.18.130.20 198.18.130.11 198.18.130.12; do
  echo "--- $ip ---"
  sshpass -p cisco ssh -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null \
    nac_admin@$ip "show run | include banner"
done
```

You should see `banner login ^CWelcome to the IOS XE as Code lab!^C` four times - one per device. If any device prints no banner line, that apply didn't land; re-run `terraform apply` to retry.

`sshpass` is already pre-installed on the lab VM; no extra setup.

You've just deployed your first Network as Code configuration using Terraform!

Notice how you defined the banner once in the global section, and it was automatically applied to all four devices. This is the power of Network as Code!

Terraform Command Reference

Here's a quick reference of the most common Terraform commands:

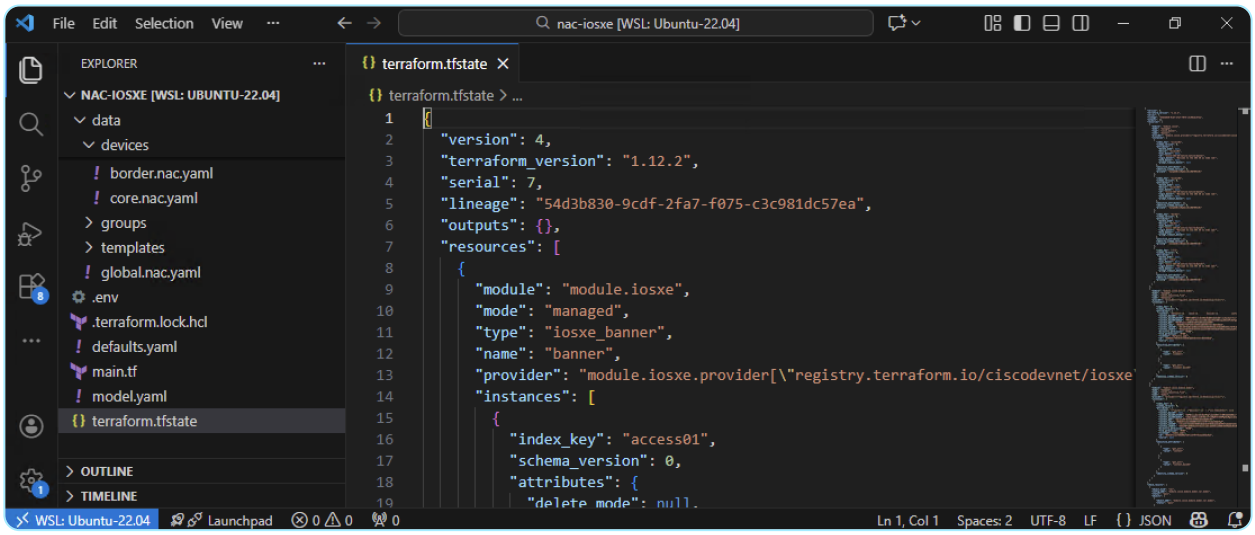
Command	Purpose
`terraform init`	Initialize project and download modules
`terraform plan`	Preview changes without applying them
`terraform apply`	Apply configuration to devices
`terraform apply -auto-approve`	Apply without confirmation prompt
`terraform destroy`	Remove all managed resources
`terraform show`	Display current state
`terraform validate`	Check configuration syntax

Copyright © Cisco

Terraform state

After running `terraform apply`, Terraform creates a `terraform.tfstate` file that tracks:

- What resources have been created
- Current configuration of each resource
- Device connection details



> **Important**
>
> The state file is sensitive and critical for Terraform to manage your infrastructure. Don't manually edit or delete it!

> **State File Location**
>
> This lab uses the default local state file (`terraform.tfstate`) stored in your project directory.
>
> While this is easy to use for learning and small projects, it's not suitable for production environments. In real-world scenarios, consider using remote state backends like **Terraform Cloud**, **AWS S3**, **Azure Blob Storage**, **HTTP backends**, **Postgres databases**, etc. to securely store and share state files.
>
> For more information, refer to the Terraform documentation here.

Troubleshooting common issues

> **Error: Failed to connect to device**
>
> **Solution:** Verify your device host address is correct and the device is reachable.

Copyright © Cisco

Error: Invalid credentials

Solution: Check that your environment variables are set correctly with `env | grep IOSXE`. If they're not set, run `source .env` again

Module not found

Solution: Run `terraform init` again to download the required modules

HOW TO READ A TERRAFORM + IOS XE PROVIDER ERROR

Most failures you'll see during `terraform apply` follow the same shape. Learning the anatomy once means every future error is readable. Here's a real example, annotated:

OUTPUT

```
Error: Client Error                                    ← (1) error category
                                                         (Client/Server/Provider)
  with module.iosxe.iosxe_banner.banner[0],            ← (2) Terraform resource
  on .terraform/modules/iosxe/iosxe/banner.tf line 5,    identifier + source file
  in resource "iosxe_banner" "banner":
   5: resource "iosxe_banner" "banner" {

Failed to commit (iosxe_banner.banner[0]),             ← (3) operation that failed
  got error: NETCONF RPC error from 198.18.130.10:830:   and which device
    bad-element: banner                                ← (4) device-side detail -
    error-message: 'banner' is not a valid element       this is what the device
    error-severity: error                                actually said back
```

Four parts, always in this order:

1. **Error category** - `Client Error` means your input or config is wrong; `Server Error` means the device rejected something; `Provider Error` means the Terraform provider itself had a bug. Category tells you whose problem it is.
2. **Terraform resource identifier** - which specific resource Terraform was trying to operate on. Match this to your YAML to find the source.
3. **Operation and target device** - `Failed to commit` / `Failed to create` / `Failed to delete`, plus the device IP. Tells you exactly which device to investigate.
4. **Device-side detail** - the actual NETCONF/RESTCONF error payload from the device. This is what the IOS XE box said back when Terraform tried to push the config. 90% of the time this line names the offending element or value explicitly.

Working error messages right-to-left (detail -> device -> resource -> category) usually gives you the root cause in under a minute. For very noisy errors, re-run with `TF_LOG=DEBUG terraform apply` to see the full NETCONF XML payload the provider sent - often the fastest way to spot a subtle field mismatch.

What you've accomplished

Congratulations! You've successfully:

- Created YAML configuration files
- Initialized Terraform with the Network as Code module
- Previewed changes with `terraform plan`
- Applied configuration to your network devices with `terraform apply`
- Verified the banner on your devices

Copyright © Cisco

Why that `plan` ran so fast - Terraform vs. stateless tools

Now that you've seen `plan` and `apply` in action, here's *why* the workflow feels different from tools you may have used before: Terraform maintains a local state file (`terraform.tfstate`) that records what it has configured on each device. That changes the math for every operation.

Capability	Terraform (stateful)	Stateless tools (e.g. Ansible)
Detecting drift	Diffs state vs. YAML - instant, no device calls	Must connect to every device and read config before comparing
Idempotency	Native - pushes only what changed	Per-module; depends on each module being written correctly
Dry-run (preview)	`terraform plan` , architecturally read-only	`--check` mode, opt-in per module; coverage varies
Audit trail	State file + plan output record exactly what changed when	Run logs only; no authoritative record of current state
Network traffic	Low - no reads when nothing changed	Full read on every run regardless of whether anything needs to change
Rollback primitive	Revert YAML, re-run `apply` , state tracks the revert	Re-run a previous playbook, hope it's still idempotent

Practical effect: when nothing in your YAML changed, `terraform plan` reports "No changes" in seconds without touching the network. Ansible in the same scenario still SSHs every device to re-read config before it can reach the same conclusion. Try it: run `terraform plan` again right now - the second run is orders of magnitude faster than the first because state already matches intent.

What's next?

In the next task, you'll learn how to use device groups to apply configurations to multiple devices efficiently.

Task 02 - Editing YAML files

Task 04 - Device group configuration

Copyright © Cisco

Task 04 - Device group configuration

Estimated Time to Complete: ~15 minutes

In this task you'll apply configuration to a **group of devices** at once using device groups. An Access Control List (ACL) is the example - device groups let you apply the same configuration to every device that shares a role or location, without duplicating the YAML.

What you'll learn

By the end of this task you will have:

- Defined an `ACCESS_SWITCHES` device group and its two member devices
- Applied a standard ACL to the group via a single YAML file
- Confirmed the ACL landed on `access01` + `access02` only (not `core` or `border`)

Device groups

Device groups provide a mechanism for applying configurations to multiple devices without repeating the same settings for each device.

Device groups are particularly effective for:

- **Role-based configuration**: Grouping devices by function (access switches, core switches, border switches)
- **Location-based configuration**: Grouping devices by physical or logical location (data center, branch office)
- **Service deployment**: Rolling out consistent service configurations across multiple devices
- **Security policies**: Applying common ACLs or security settings to device groups

Use case: standard ACL for access switches

In this example, you'll create a device group called **ACCESS_SWITCHES** that includes the **access01** and **access02** switches. These switches need a standard ACL to permit traffic from specific network ranges (`10.0.0.0/24` and `20.0.0.0/24`) - a typical requirement for access layer devices controlling traffic from known networks.

> **The power of Device Groups: Scalability**
>
> Using device groups may appear unnecessary for just two access switches - in this example. However, consider a large network with more than 1,000 access switches. Utilizing device groups helps keep the configuration organized and scalable.

Step 1: Create the device group configuration file

First, create the file using your **WSL Ubuntu terminal**:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
touch ~/nac-iosxe/data/groups/access.nac.yaml
```

The file will appear in VS Code's Explorer panel. Click on `groups/access.nac.yaml` to open it and add the following content. Notice how the ACL is defined once in the device group and automatically applies to both **access01** and **access02** switches:

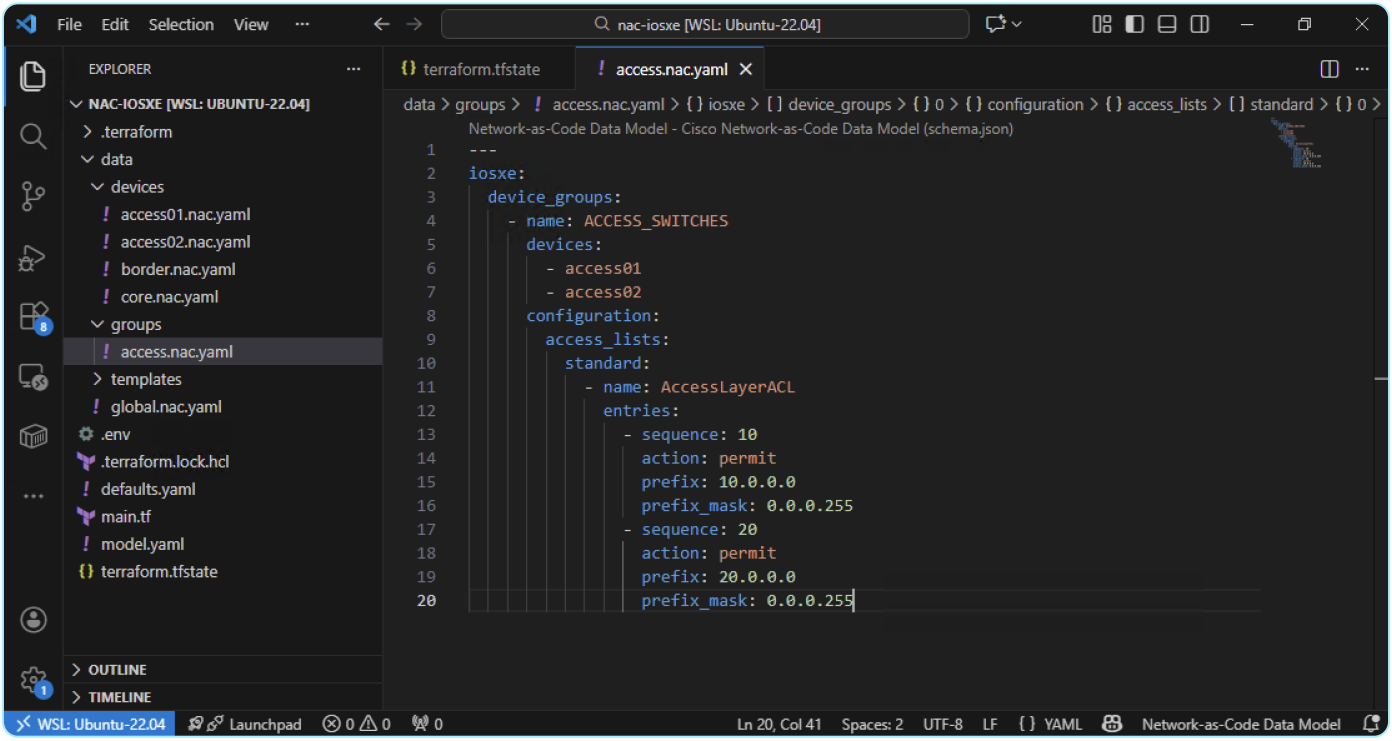
FILE **groups/access.nac.yaml**

```
---
iosxe:
```


Copyright © Cisco

```yaml
device_groups:
  - name: ACCESS_SWITCHES
    devices:
      - access01
      - access02
    configuration:
      access_lists:
        standard:
          - name: AccessLayerACL
            entries:
              - sequence: 10
                action: permit
                prefix: 10.0.0.0
                prefix_mask: 0.0.0.255
              - sequence: 20
                action: permit
                prefix: 20.0.0.0
                prefix_mask: 0.0.0.255
```

The image below illustrates the ACL configuration in VS Code:



CONFIGURATION BREAKDOWN

Let's break down the key elements:

Device Group Section:

- `device_groups:` - Defines one or more device groups
- `name: ACCESS_SWITCHES` - The group name identifier
- `devices:` - Lists member devices (`access01` , `access02`)
- `configuration:` - Contains settings applied to all group members

Copyright © Cisco

Alternative: You could also define group membership from devices

Try this at home!

You can also add a device to a device group from under the device configuration section by specifying the `device_groups` attribute. This is useful when you want to assign a device to multiple groups or prefer defining group membership alongside device-specific settings.

FILE **data/devices/example-device.nac.yaml (illustrative)**

```yaml
---
iosxe:
  devices:
    - name: example-device
      device_groups:
        - EXAMPLE_GROUP1
        - EXAMPLE_GROUP2
        - EXAMPLE_GROUP3
      configuration:
        # device-specific configuration here
        ...
```

The reference between the device and device group can be configured from both sides.

Access List Configuration:

- `access_lists:` - Top-level ACL configuration
- `standard:` - Specifies standard ACL type (filters based on source address only)
- `name: AccessLayerACL` - The ACL name
- `entries:` - Individual ACL rules (processed in sequence order)

ACL Entry Details:

- `sequence 10` : Permits traffic from the `10.0.0.0/24` network
  - `action: permit` - Allows matching traffic
  - `prefix: 10.0.0.0` - The network address
  - `prefix_mask: 0.0.0.255` - Wildcard mask (matches `10.0.0.0` through `10.0.0.255`)
- `sequence 20` : Permits traffic from the `20.0.0.0/24` network
  - `action: permit` - Allows matching traffic
  - `prefix: 20.0.0.0` - The network address
  - `prefix_mask: 0.0.0.255` - Wildcard mask (matches `20.0.0.0` through `20.0.0.255`)

How device groups work

When Terraform processes this configuration:

1. The **global banner** applies to all devices (**border**, **core**, **access01**, **access02**)
2. The **ACCESS_SWITCHES** group's configuration (in our case the ACL **AccessLayerACL**) applies only to **access01** and **access02** switches
3. If you later add device-specific configuration to the **access01** device, it could override global or group settings (e.g., changing the banner)

This hierarchical approach ensures:

- No configuration duplication (ACL defined once, applied to multiple devices)
- Easy maintenance (update ACL in one place, changes apply to all group members)
- Scalability (add more switches by just adding them to the group's device list)

Copyright © Cisco

Step 2: Apply access-list configuration

Open your WSL Ubuntu terminal and navigate to your project directory. Run Terraform to deploy the ACL configuration to the device group:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
cd ~/nac-iosxe
```

terraform init not required

You do not need to run `terraform init` again, as the project has already been initialized in a previous task.

It is good practice to run `terraform plan` first to preview the changes that will be made. You could also skip this step as the plan is automatically generated during `terraform apply`.

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform plan
```

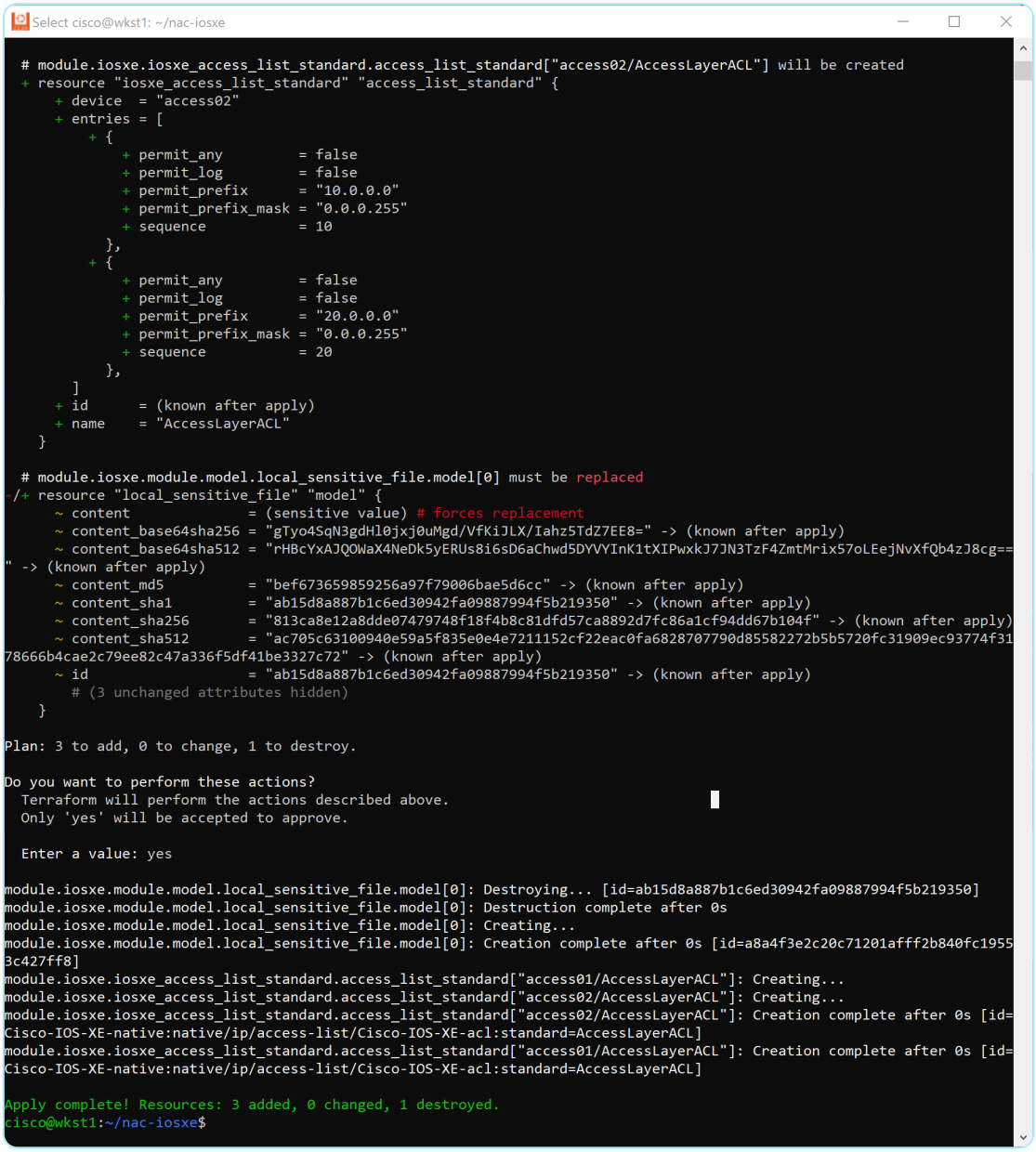
Now, run the following command to apply the configuration:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform apply
```

When prompted, type `yes` to confirm the deployment. Terraform will create the standard ACL on all devices in the **ACCESS_SWITCHES** group (**access01** and **access02**).

Copyright © Cisco



Step 3: Verify device group configuration

After successfully running `terraform apply`, verify that the ACL was deployed only to the switches in the **ACCESS_SWITCHES** group.

Use Solar-PuTTY to connect and verify:

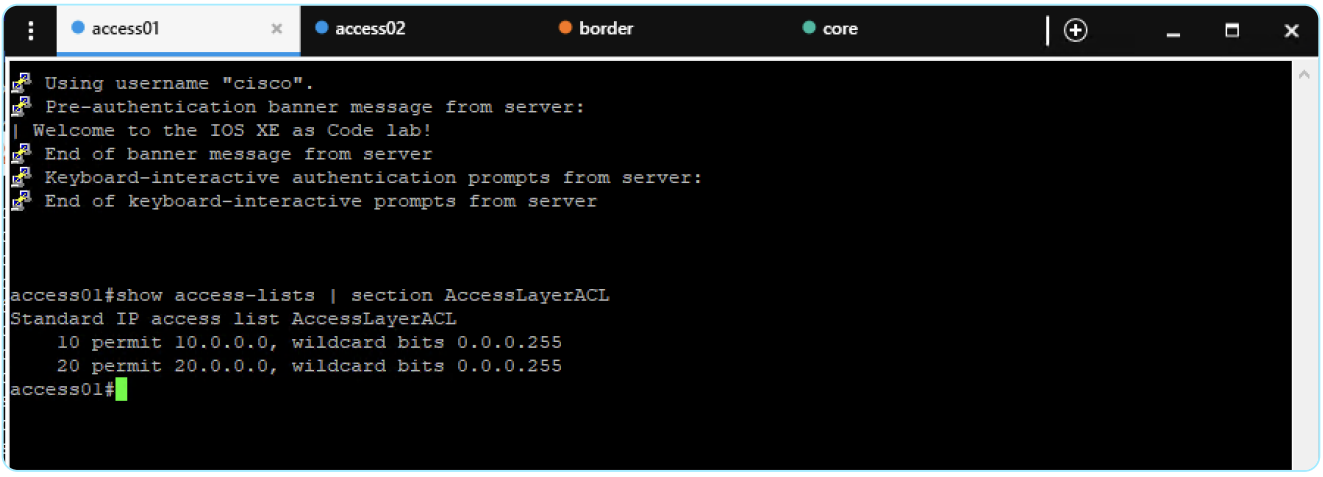
1. Open **Solar-PuTTY** from your desktop
2. Connect to the **access01** switch
3. Check if the ACL is present using the command below
4. Repeat for the **access02** switch

Use the following command on both **access01** and **access02** switches to verify the ACL:

DEVICE CLI · PuTTY **access01, access02**

```
show access-lists | section AccessLayerACL
```


Copyright © Cisco



This confirms the standard ACL was successfully deployed to both **access01** and **access02** switches with both network permit entries.

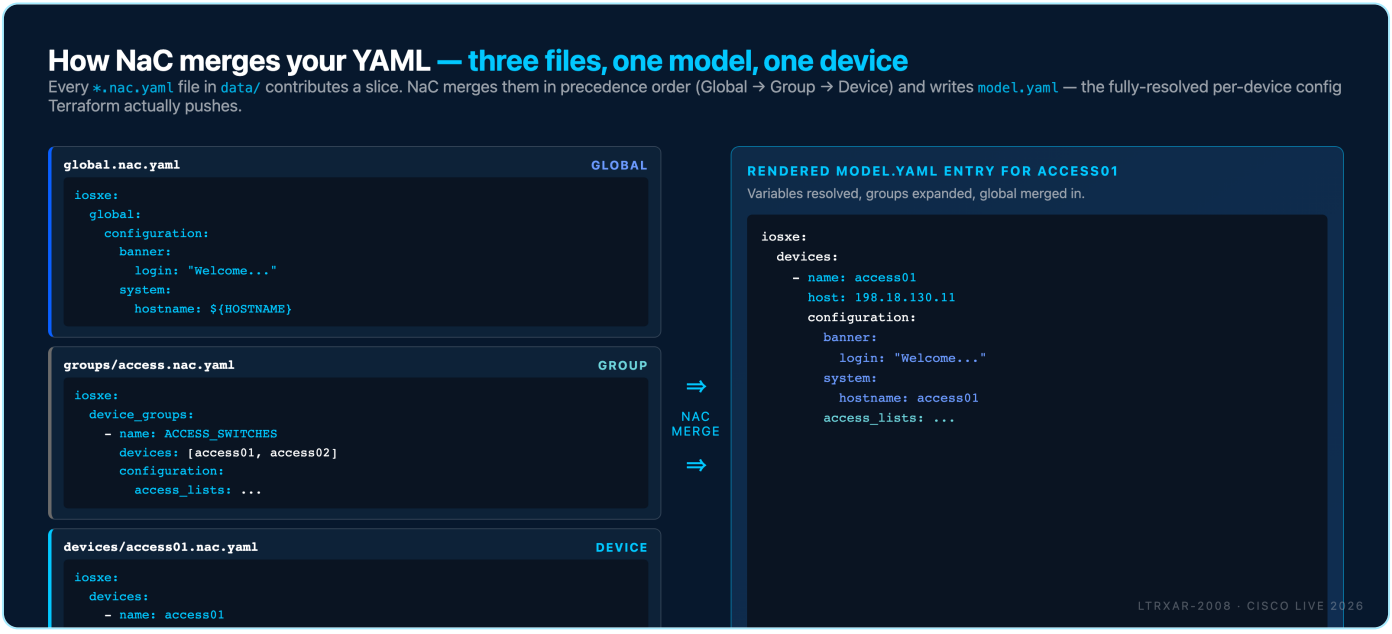
Key observation

The ACL only appears on devices that are members of the **ACCESS_SWITCHES** group. If you check border or core switches (not in the group), they won't have this ACL, demonstrating the selective deployment capability of device groups.

Generated model file

When Terraform applies your configuration, the Network as Code module performs a deep merge of all your YAML files - global configuration, device group configuration, and device-specific configuration - into a single per-device view. The result is written to `model.yaml` (because you set `write_model_file = "model.yaml"` in `main.tf`).

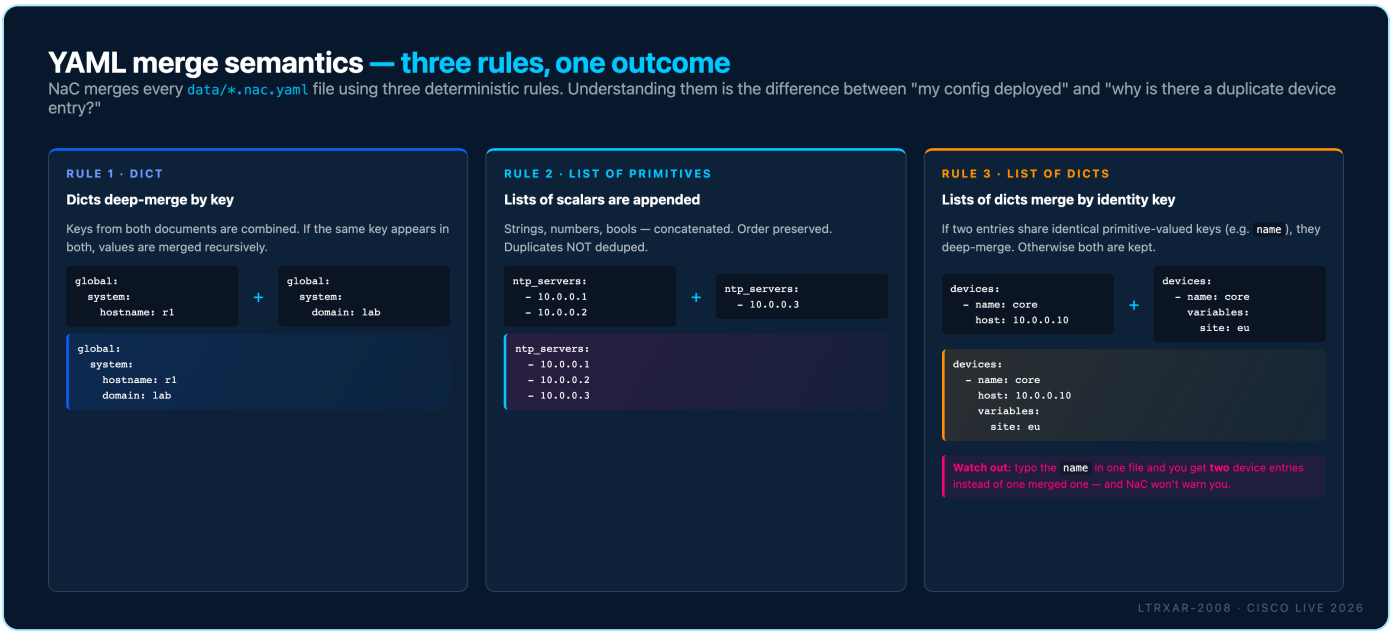
The merge follows the same precedence hierarchy you've been working with: **global → device group → device**, where more specific levels override less specific ones for the same keys. Variables are substituted, templates are rendered, and the output is a flat list of devices, each with its fully resolved configuration.



Copyright © Cisco

THE THREE MERGE RULES

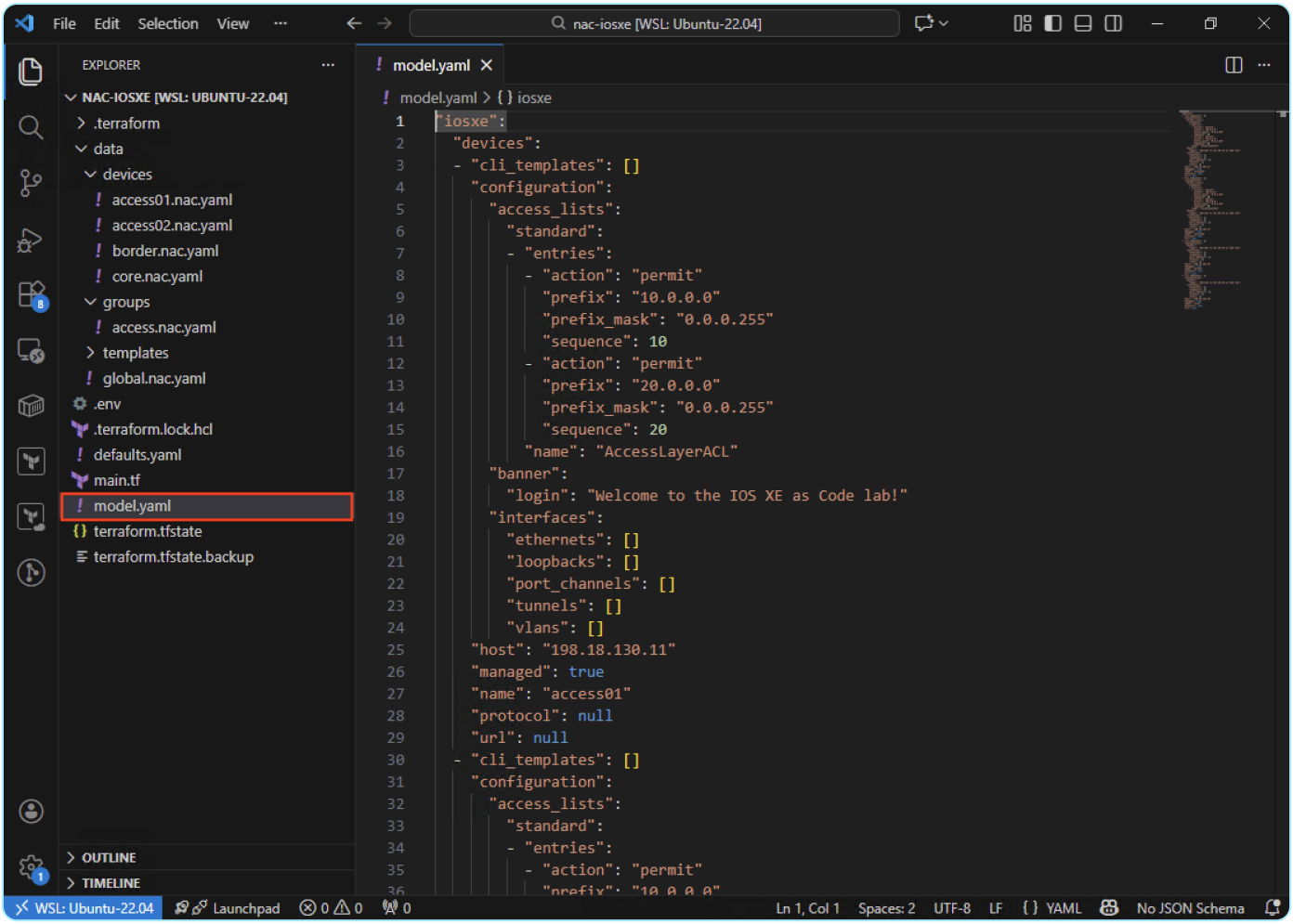
The merge behavior is formally documented, but you've just produced a concrete example of each rule in action. Three rules govern what happens when two YAML files touch the same structure:



- **Rule 1 (dict deep-merge)** – the `configuration:` block under each device is a dict, and the global banner + group ACL + per-device settings all deep-merge into one nested dict for each device.
- **Rule 2 (lists of scalars append)** – not exercised yet, but the same behavior applies to e.g. `ntp_servers: [...]` lists across files.
- **Rule 3 (lists of dicts merge by identity key)** – this is why your four `devices:` entries (one per file) produce four device entries, not eight. Network as Code identifies `devices` list members by `name` and deep-merges entries that share the same `name`. Typo a `name` in one file and you silently end up with two device entries instead of one.

Open `model.yaml` in VS Code to see the result. Notice how the ACL from the **ACCESS_SWITCHES** group appears under `access01` and `access02` but not under `core` or `border`:

Copyright © Cisco



Why this matters:

- **Debugging** – if a device gets unexpected configuration, `model.yaml` shows you exactly what the module computed for that device after all merges and variable substitutions.
- **Verification** – `nac-test` (Task 11) uses `model.yaml` as its input to generate post-deployment tests. The tests assert against what the model says *should* be on the device.
- **Transparency** – rather than trusting the module as a black box, you can inspect the fully resolved intent for every device in one file.

Copyright © Cisco

What the module is actually saving you from - a concrete before/after

Your `data/` directory right now is about **40 lines of YAML** across four files: four per-device registration entries, one global banner, one group-level ACL. That's the input.

If the Network as Code module didn't exist and you were writing equivalent Terraform directly against the `terraform-provider-iosxe` resources, you'd be hand-writing HCL that looks like this:

FILE **what you'd write without the module**

```
resource "iosxe_banner" "core_banner"     { device = "core"     ; ... }
resource "iosxe_banner" "border_banner"   { device = "border"   ; ... }
resource "iosxe_banner" "access01_banner" { device = "access01" ; ... }
resource "iosxe_banner" "access02_banner" { device = "access02" ; ... }

resource "iosxe_access_list_standard" "access01_acl" {
  device = "access01"
  name   = "AccessLayerACL"
  entry  = [
    { sequence = 10, remark = "..." },
    { sequence = 20, remark = "..." },
    # ...etc, repeated inline per device
  ]
}
resource "iosxe_access_list_standard" "access02_acl" {
  device = "access02"
  name   = "AccessLayerACL"
  entry  = [
    # ...same entries, repeated verbatim
  ]
}

# ...plus provider blocks, variables, outputs, locals, for_each
# maps, and one resource per (device, feature) pair going forward.
```

That's roughly **150-200 lines of HCL** for what you wrote in **~40 YAML lines**, and the ratio gets *worse* as you add devices and features. The module is doing four distinct jobs under the hood:

1. **Deep-merging** your YAML files into one view per device (global -> group -> device precedence).
2. **Fanning out** each merged device's config into ~dozens of provider resource calls (one per feature).
3. **Rendering** `${variables}` and templates at merge time so your YAML stays declarative.
4. **Tracking** which resources apply to which devices so `terraform plan` shows a per-device diff instead of a flat list.

You can write everything you've written here against raw `terraform-provider-iosxe` resources if you need to - the module IS just convention + code generation on top of that provider. But for any fleet beyond about two devices, the module is what makes the math practical.

What you've accomplished

In this task, you have:

- Learned about device groups and configuration hierarchy
- Created an **ACCESS_SWITCHES** device group with **access01** and **access02** switches
- Applied a standard ACL to multiple devices using a single definition
- Verified selective configuration deployment to group members only

Task 03 - Global configuration

Task 05 - Single-device configuration

Copyright © Cisco

Task 05 - Single-device configuration

Estimated Time to Complete: ~15 minutes

In this task, you'll apply configuration to a **single specific device** rather than globally or to a group. This is the top of the precedence hierarchy - device-specific settings override anything a device might have inherited from its group or from the global defaults.

What you'll learn

By the end of this task you will have:

- Extended `devices/core.nac.yaml` with device-specific configuration (a `Loopback0` interface)
- Internalized the Network as Code precedence hierarchy: **Global < Device Group < Device**
- Verified selective deployment - loopback exists on `core`, absent from `border` / `access01` / `access02`

Device-specific configuration

Device-specific config is the right tool when a setting is meaningful only for one device. Typical use cases:

- **Intrinsically unique values** - loopback interfaces, router IDs, management IPs, OSPF/BGP/MPLS identifiers - every device needs its **own** value.
- **Overrides** - one device needs to diverge from what its group or the global default says (a different banner on a lab device, a stricter ACL on the border router, etc.).
- **Special-purpose devices** - a single out-of-band management switch that doesn't fit any group.

Precedence hierarchy (reminder):

1. **Global** (lowest precedence) - organization-wide defaults ← *Task 03*
2. **Device group** (medium) - role- or location-specific settings ← *Task 04*
3. **Device** (highest) - the specific-device override you're adding now ← *this task*

Use case: Loopback0 on the core switch

`core` is acting as the routing core of the lab topology. In any routed network, every device needs its **own** unique loopback interface for router IDs (OSPF, BGP, MPLS-LDP) and for policies that key on a stable IP. That's the textbook "single-device configuration" use case - and a cleaner fit than, say, NTP or syslog pointers, which every device in the network would typically need.

You'll configure `core` with:

- `Loopback0` - IP `198.51.100.10/32` (RFC 5737 documentation range), described as "Router-ID loopback"

Only `core` will receive this. The other three devices (`border`, `access01`, `access02`) will be untouched.

Step 1: Add device-specific configuration

You already created a per-device file for `core` in Task 02 - you'll extend it now by adding a `configuration:` block. The file currently registers `core` with Network as Code; adding `configuration:` tells Network as Code what to actually push to that specific device.

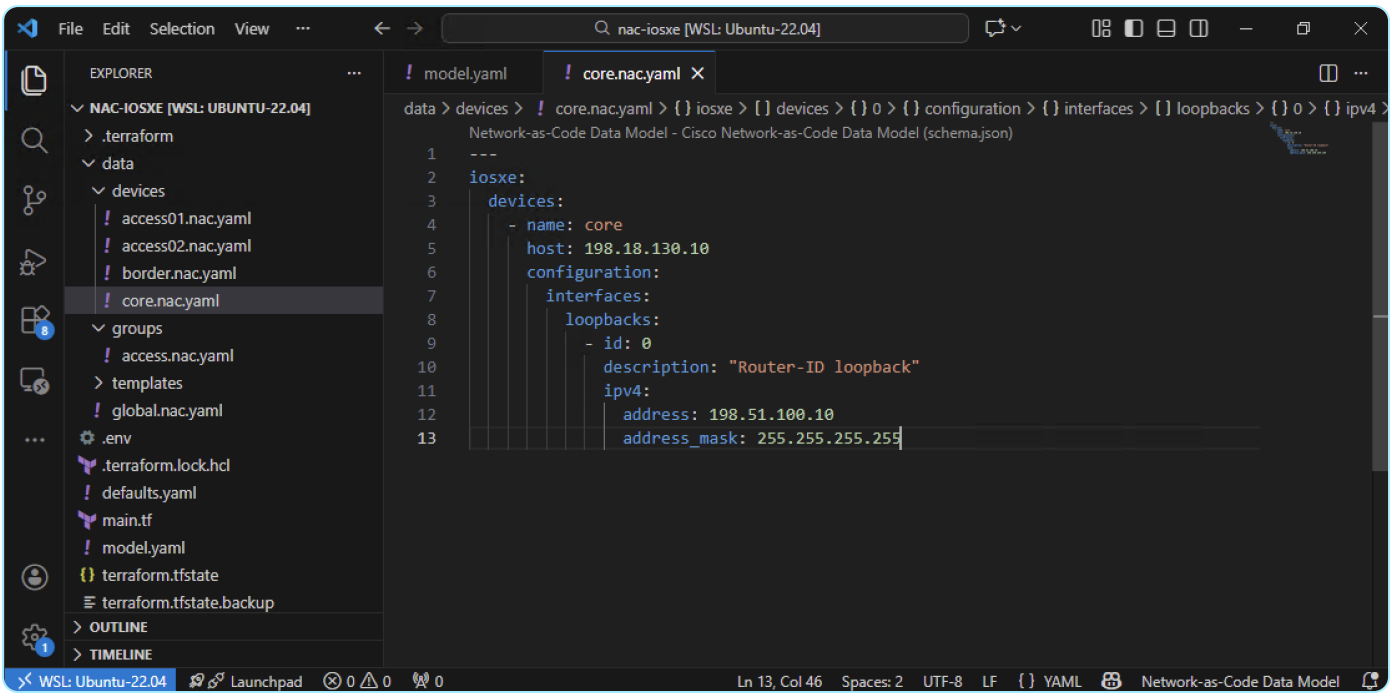
Open `data/devices/core.nac.yaml` in VS Code and replace its contents with:

FILE **data/devices/core.nac.yaml**

```
---
iosxe:
  devices:
```


Copyright © Cisco

```yaml
  - name: core
    host: 198.18.130.10
    configuration:
      interfaces:
        loopbacks:
          - id: 0
            description: "Router-ID loopback"
            ipv4:
              address: 198.51.100.10
              address_mask: 255.255.255.255
```



CONFIGURATION BREAKDOWN

Device section:

- `devices:` - the top-level device list. Every per-device file contributes exactly one entry here.
- `name: core` - unique device identifier. Network as Code matches this against the same `name` in other files (global, group, etc.) when it decides what to apply to which device.
- `host: 198.18.130.10` - carries over from Task 02. Same device, same IP, now with configuration attached.
- `configuration:` - everything under this key applies **only to** `core`, no other device.

Loopback configuration:

- `interfaces.loopbacks` - the IOS XE as Code data model path for virtual loopback interfaces.
- `id: 0` - creates `Loopback0`.
- `description` - free-form description, visible in `show interfaces`.
- `ipv4.address` / `ipv4.address_mask` - IP (RFC 5737 documentation range) and mask. `/32` is conventional for loopbacks used as router IDs.

> **Why a loopback here?**
>
> A loopback is the canonical single-device configuration. Every device in a routed network needs its **own** unique loopback for router-ID (OSPF, BGP, MPLS-LDP, etc.), so this example naturally belongs under device-specific config rather than global or group config. It's the pattern you'll use any time a setting is meaningful only in the context of one specific device.

Copyright © Cisco

What about NTP, syslog, DNS entries?

The lab also has `ntp-server` (`198.18.129.11`) and `syslog-server` (`198.18.129.12`) reachable from every device. Things like those - an `ntp server` pointer, a `logging host` - should typically be **global** (every device needs to know about them), not device-specific. That's a good exercise to try at home.

FILE ORGANIZATION

Your `data/` folder now contains one file per configuration concern:

OUTPUT **Project layout**

```
/home/cisco/nac-iosxe/
├── .env
├── main.tf
└── data/
    ├── devices/
    │   ├── core.nac.yaml       # Device: core  (registration + Loopback0)  ← this task
    │   ├── border.nac.yaml     # Device: border (registration only, for now)
    │   ├── access01.nac.yaml   # Device: access01 (registration only)
    │   └── access02.nac.yaml   # Device: access02 (registration only)
    ├── groups/
    │   └── access.nac.yaml     # Group   - ACL for access switches       ← Task 04
    └── global.nac.yaml         # Global  - login banner                    ← Task 03
```

One concern per file

This is how the lab guide organizes things. In your own projects, organize however makes sense for your team - `nac-validate` (a tool for pre-change validation that we will cover in Task 10) and the Network as Code module don't care about file boundaries, only about the merged data model.

Step 2: Apply device-specific configuration

Open your WSL Ubuntu terminal and run the following steps:

Navigate to your project directory:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
cd ~/nac-iosxe
```

Optionally, preview the changes Terraform will make:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform plan
```

Apply the configuration:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform apply
```

When prompted, type `yes` to confirm the deployment. Terraform will create `Loopback0` on `core` only.

What to observe in the plan output:

- Terraform shows changes only for the **core** device
- No changes are proposed for **border**, **access01**, or **access02**

Copyright © Cisco

```
Terraform will perform the following actions:

  # module.iosxe.iosxe_interface_loopback.loopback["core/Loopback0"] will be created
  + resource "iosxe_interface_loopback" "loopback" {
      + description       = "Router-ID loopback"
      + device            = "core"
      + id                = (known after apply)
      + ipv4_address      = "198.51.100.10"
      + ipv4_address_mask = "255.255.255.255"
      + name              = 0
    }

  # module.iosxe.module.model.local_sensitive_file.model[0] must be replaced
-/+ resource "local_sensitive_file" "model" {
      ~ content              = (sensitive value) # forces replacement
      ~ content_base64sha256 = "NI/VhWlTIF+keC1djoWli6QBvTyGONnivc+ClF8kvfA=" -> (known after apply)
      ~ content_base64sha512 = "VDQk0Gca6OBMYELmrdFPFQ+PN+y0MgjUtJTJuUyEgBQjhR4yF9DSa32elQ2vrC2xyGcEpgoN9id0BYIIyX9Neg==" -> (known after apply)
      ~ content_md5          = "98f5d4b7cd0ee230b2bd17c675d655fe" -> (known after apply)
      ~ content_sha1         = "3ae33b9a1db22b68b897af6e2b02d1a2da69399b" -> (known after apply)
      ~ content_sha256       = "348fd5856953205fa4782d5d8e85a58ba401bd3c8638d9e2bdcf82945f24bdf0" -> (known after apply)      ~ content_sha512       = "5
43424d0671ae8e04c6042e6add14f150f8f37ecb43208d4b494c9b94c84801423851e3217d0d26b7d9e950dafac2db1c86704a60a0df62774058208c97f4d7a" -> (known after apply)
      ~ id                   = "3ae33b9a1db22b68b897af6e2b02d1a2da69399b" -> (known after apply)
        # (3 unchanged attributes hidden)
    }

Plan: 2 to add, 0 to change, 1 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

module.iosxe.module.model.local_sensitive_file.model[0]: Destroying... [id=3ae33b9a1db22b68b897af6e2b02d1a2da69399b]
module.iosxe.module.model.local_sensitive_file.model[0]: Destruction complete after 0s
module.iosxe.module.model.local_sensitive_file.model[0]: Creating...
module.iosxe.module.model.local_sensitive_file.model[0]: Creation complete after 0s [id=41db732a16f2fed534aee92e77b268a393140a9b]
module.iosxe.iosxe_interface_loopback.loopback["core/Loopback0"]: Creating...
module.iosxe.iosxe_interface_loopback.loopback["core/Loopback0"]: Creation complete after 30s [id=Cisco-IOS-XE-native:native/interface/Loopback=0]

Apply complete! Resources: 2 added, 0 changed, 1 destroyed.
```

Step 3: Verify the device-specific configuration

After `terraform apply` finishes, verify that `Loopback0` exists on `core` and **only** on `core`.

Verify on `core` (should have Loopback0)

1. Open Solar-PuTTY and connect to **core**.
2. Run the verification commands below.

Check the loopback interface:

DEVICE CLI · PuTTY **core**

```
show ip interface brief | include Loopback
```

OUTPUT **Expected output**

```
core#show ip interface brief | include Loopback
Loopback0              198.51.100.10   YES NVRAM  up                    up
core#
```

Inspect the full interface:

DEVICE CLI · PuTTY **core**

```
show interfaces Loopback0
```

OUTPUT **Expected output (truncated)**

```
core#show interfaces Loopback0
Loopback0 is up, line protocol is up
  Hardware is Loopback
  Description: Router-ID loopback
  Internet address is 198.51.100.10/32
  MTU 1514 bytes, BW 8000000 Kbit/sec, DLY 5000 usec,
  ...
```

You should see `Loopback0` up/up with IP `198.51.100.10/32` and your description.

Verify on the other devices (should NOT have Loopback0)

Copyright © Cisco

Connect to **border** and run:

DEVICE CLI · PuTTY **border, access01, access02**

```
show ip interface brief | include Loopback
```

The command should return no output - confirming that `Loopback0` was not configured on `border`. Repeat on `access01` and `access02` if you want to be thorough; same expected result.

Key observation

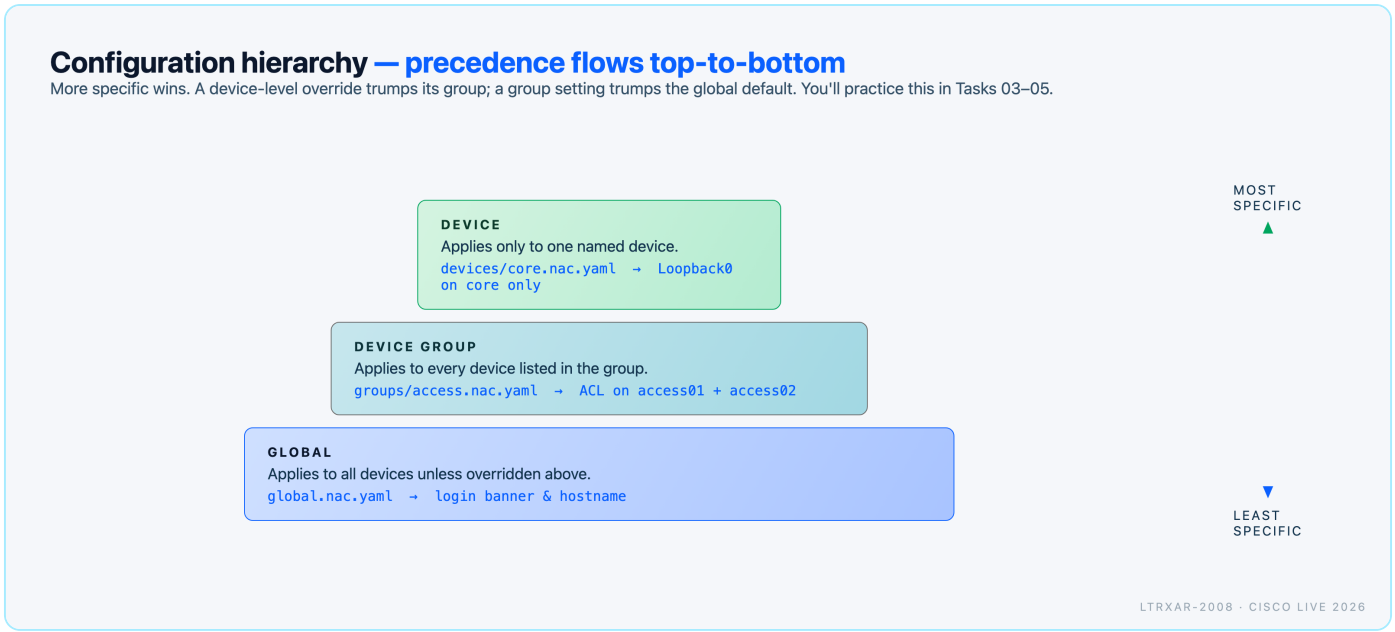
`Loopback0` only appears on `core` because you declared it inside that device's per-device file. This is the essence of device-specific configuration: the YAML scope **is** the deployment scope.

Configuration hierarchy comparison

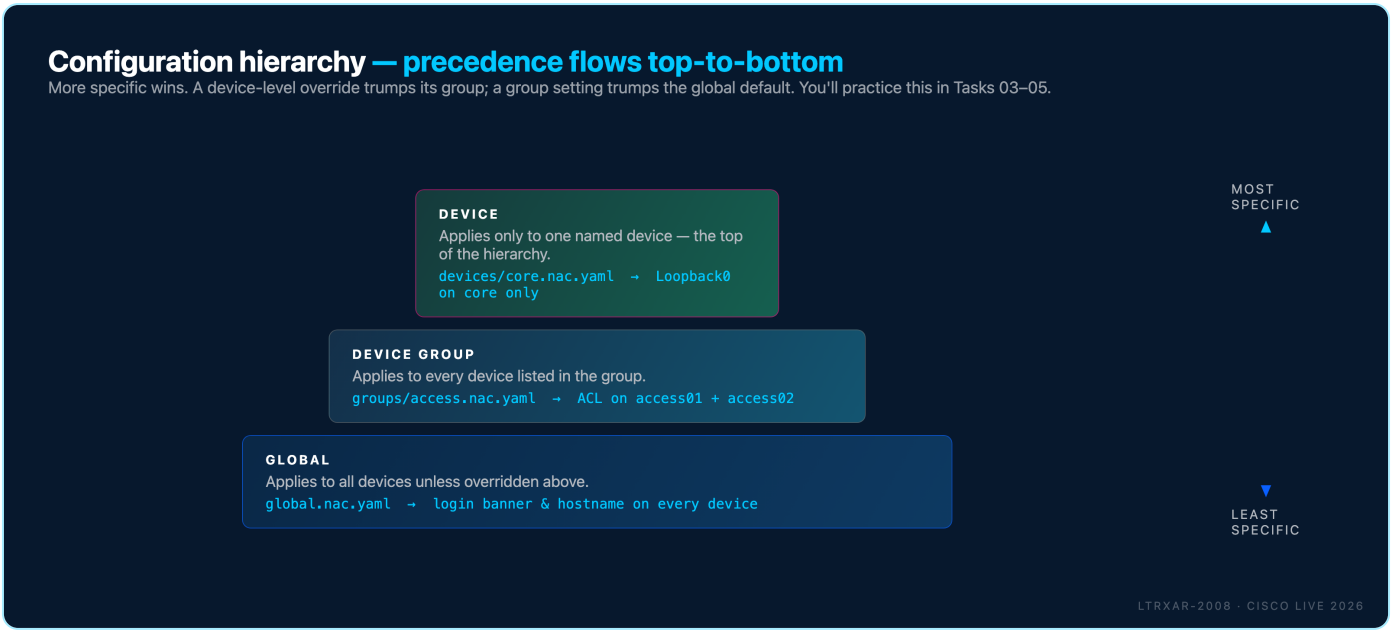
Now that you've completed Tasks 03, 04, and 05, you've experienced all three levels of the configuration hierarchy. Here's a summary:

Level	Scope	Example	File
Global	All devices	Login banner	`global.nac.yaml`
Device Group	Subset of devices	Standard ACL	`groups/access.nac.yaml`
Device	Single device	`Loopback0`	`devices/core.nac.yaml`

In case of conflicting settings, the higher precedence level overrides the lower. For example, you can have a global banner, and configure a different banner for a specific device.



Copyright © Cisco



When to use each configuration level

Use Case	Recommended Level
Organization-wide standards (banners, NTP, logging)	**Global**
Role-based settings (ACLs for access layer, routing for core)	**Device Group**
Location-specific settings (site-specific config, Timezone)	**Device Group**
Intrinsically unique values (loopbacks, router-IDs, per-device IPs)	**Device**
Overriding group or global settings for one device	**Device**

What you've accomplished

- Extended `devices/core.nac.yaml` with device-specific configuration (`Loopback0`)
- Deployed a configuration that applies to exactly one device
- Verified the deployment is selective: loopback exists on `core` , absent from `border` / `access01` / `access02`
- Internalized the configuration precedence hierarchy: **Global < Device Group < Device**

You've now worked with all three levels of the Network as Code hierarchy.

Task 04 - Device group configuration

Task 06 - Variables

Copyright © Cisco

Task 06 - Variables

Estimated Time to Complete: ~15 minutes

In this task you'll use **variables** in your Network as Code configurations. Variables let you define a value once (at the global, group, or device level) and reference it everywhere it's needed, so you can keep shared settings in one place while letting each device override only what's unique to it.

What you'll learn

By the end of this task you will have:

- Used `${VARIABLE_NAME}` placeholders inside a global configuration
- Defined a **global** variable (`${SITE}`) that every device inherits
- Defined a **per-device** variable (`${HOSTNAME}`) that resolves differently on each device

Variables in Network as Code

Variables in Network as Code work similarly to variables in programming languages. You define a variable with a value at one level (device, device group, or global), and then reference it elsewhere using the `${VARIABLE_NAME}` syntax. When the Network as Code module processes your configuration, it substitutes the variable references with their actual values.

Benefits of using variables:

- **Consistency**: Define a value once, use it everywhere
- **Maintainability**: Change a value in one place, it updates everywhere it's used
- **Reusability**: Same configuration template can be used with different variable values
- **Device-specific customization**: Each device can have its own variable values while sharing configuration structure

Use case: dynamic hostname in banner and system hostname config

In this example, you'll touch two different pieces of configuration on each device: the hostname and the login banner.

- You'll define the `HOSTNAME` variable **per device**.
- You'll use that variable to set the **system hostname** in the **global configuration**.
- You'll update the login banners on each device to display its own hostname. Instead of creating separate banner configurations for each device, you'll use the same **global banner configuration** with the `HOSTNAME` variable.
- Let Network as Code substitute the variable automatically in both the system hostname and banner configuration

> **The power of variables**
>
> One banner template, multiple device-specific outputs!
>
> One place to define the hostname variable per device, used in multiple configuration sections.

Copyright © Cisco

Step 1: Update the global configuration file

First, update the global configuration to use a variable in the banner. Open `data/global.nac.yaml` in VS Code and update it with the following content:

FILE **data/global.nac.yaml**

```yaml
---
iosxe:
  global:
    variables:
      SITE: "LTRXAR-2008 Lab"
    configuration:
      banner:
        login: |-
          ########################################
          #                                      #
          #   Welcome to the IOS XE as Code Lab! #
          #                                      #
          ########################################
          Site:   ${SITE}
          Device: ${HOSTNAME}
      system:
        hostname: ${HOSTNAME}
```

Two kinds of variable reference at work here:

- `${SITE}` - defined **globally** (at the `iosxe.global.variables` level). Every device inherits the same value unless a device explicitly overrides it. This is how you keep organization-wide constants (site name, location, lab ID, DNS domain) in exactly one place.
- `${HOSTNAME}` - referenced globally, but you'll define it **per-device** in Step 2 so each device resolves it to its own value.

Banner config:

- `banner: login:` - the login banner shown on SSH connect.
- `|-` - YAML block-scalar indicator; preserves newlines and indentation inside the string while removing the trailing newline.

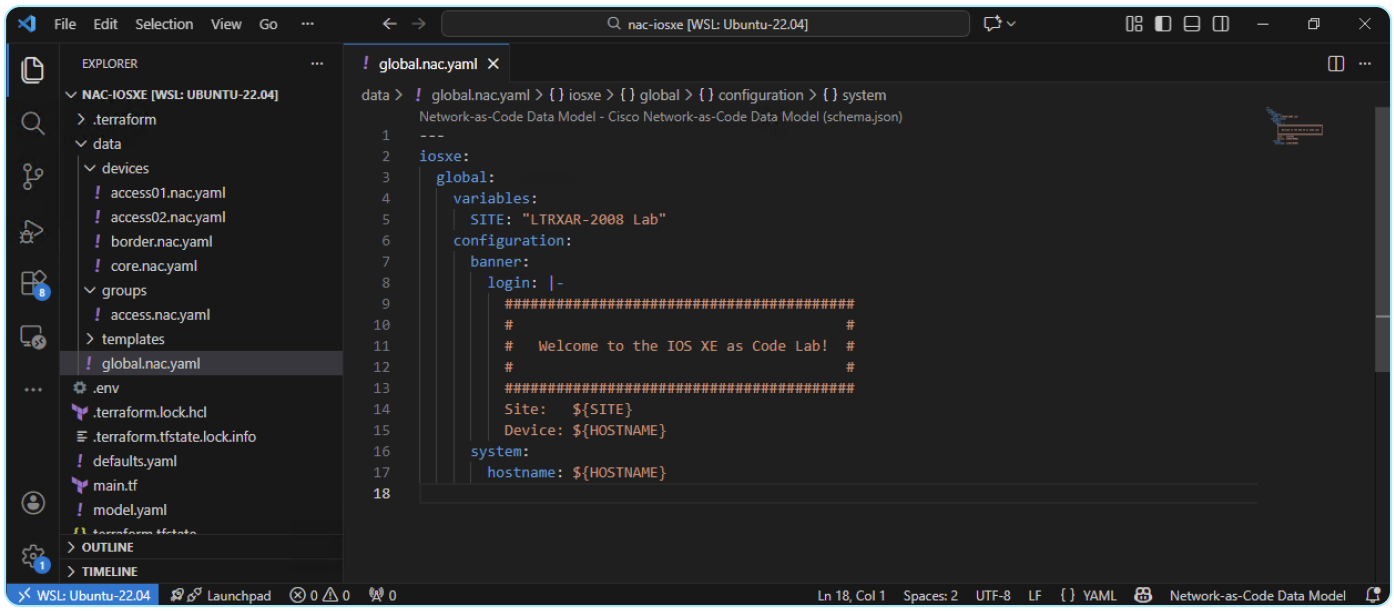
System config:

- `system: hostname:` - sets the IOS XE `hostname` CLI to the resolved value of `${HOSTNAME}`.

> **Variable syntax**
>
> Variables use the `${VARIABLE_NAME}` syntax. Names are case-sensitive: `${HOSTNAME}` and `${hostname}` are different variables. The lab uses uppercase by convention - it makes variables visually distinct from the surrounding YAML, and is a common pattern across Network as Code data models.

The image below illustrates the updated global configuration in VS Code:

Copyright © Cisco



> **YAML Formatting**
>
> When pasting multi-line strings in YAML (like this banner), ensure the indentation is correct. The `|` character indicates a multi-line string, and the following lines should be indented properly to be part of that string.
>
> To learn more about the multi-line string (block scalar) YAML syntax, refer to yaml-multiline.info.

Step 2: Define variables at device level

Now you need to define the `HOSTNAME` variable for each device. Open `data/devices/core.nac.yaml` in VS Code and update it with the following content:

FILE **data/devices/core.nac.yaml**

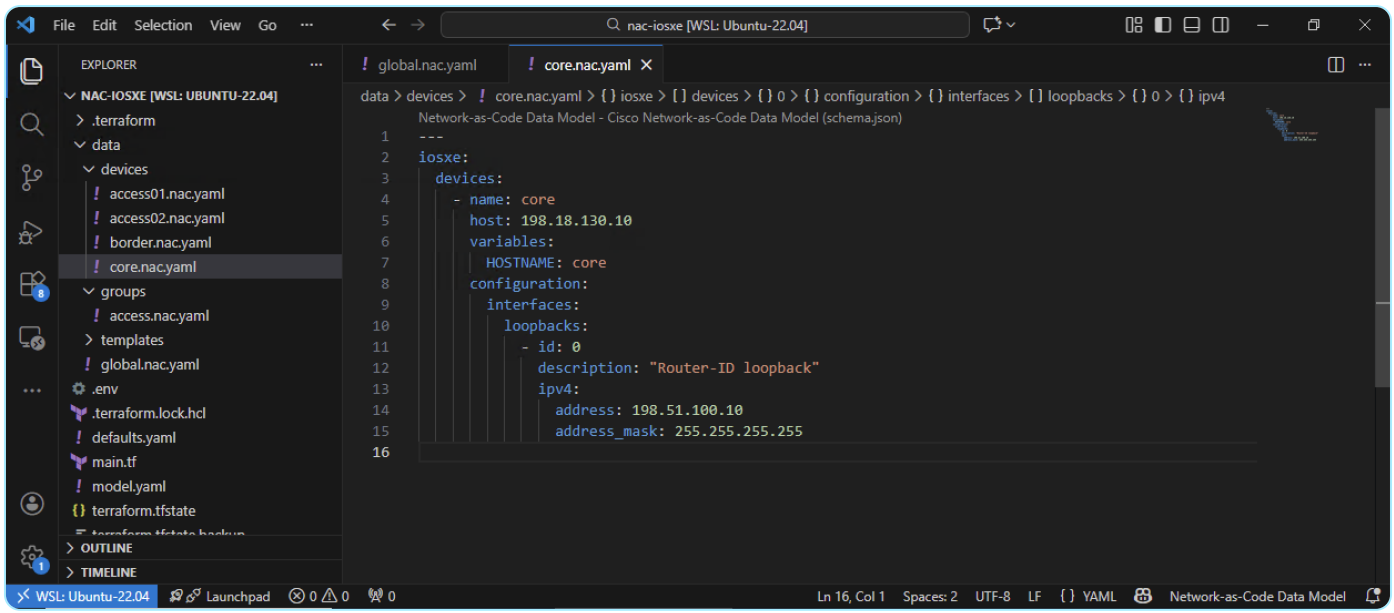
```yaml
---
iosxe:
  devices:
    - name: core
      host: 198.18.130.10
      variables:
        HOSTNAME: core
      configuration:
        interfaces:
          loopbacks:
            - id: 0
              description: "Router-ID loopback"
              ipv4:
                address: 198.51.100.10
                address_mask: 255.255.255.255
```

What's new:

- `variables:` - section where you define device-specific variables.
- `HOSTNAME: core` - sets `${HOSTNAME}` to `core` for this device only. The `system.hostname` and `banner.login` in `global.nac.yaml` will resolve to this value when the merged model is rendered for `core`.

The image below illustrates the device configuration with variables in VS Code:

Copyright © Cisco



Step 3: Add variables to other devices

Now add the `HOSTNAME` variable to the other device configuration files.

Update `data/devices/border.nac.yaml`:

FILE **data/devices/border.nac.yaml**

```yaml
---
iosxe:
  devices:
    - name: border
      host: 198.18.130.20
      variables:
        HOSTNAME: border
```

Update `data/devices/access01.nac.yaml`:

FILE **data/devices/access01.nac.yaml**

```yaml
---
iosxe:
  devices:
    - name: access01
      host: 198.18.130.11
      variables:
        HOSTNAME: access01
```

Update `data/devices/access02.nac.yaml`:

FILE **data/devices/access02.nac.yaml**

```yaml
---
iosxe:
  devices:
    - name: access02
      host: 198.18.130.12
      variables:
        HOSTNAME: access02
```

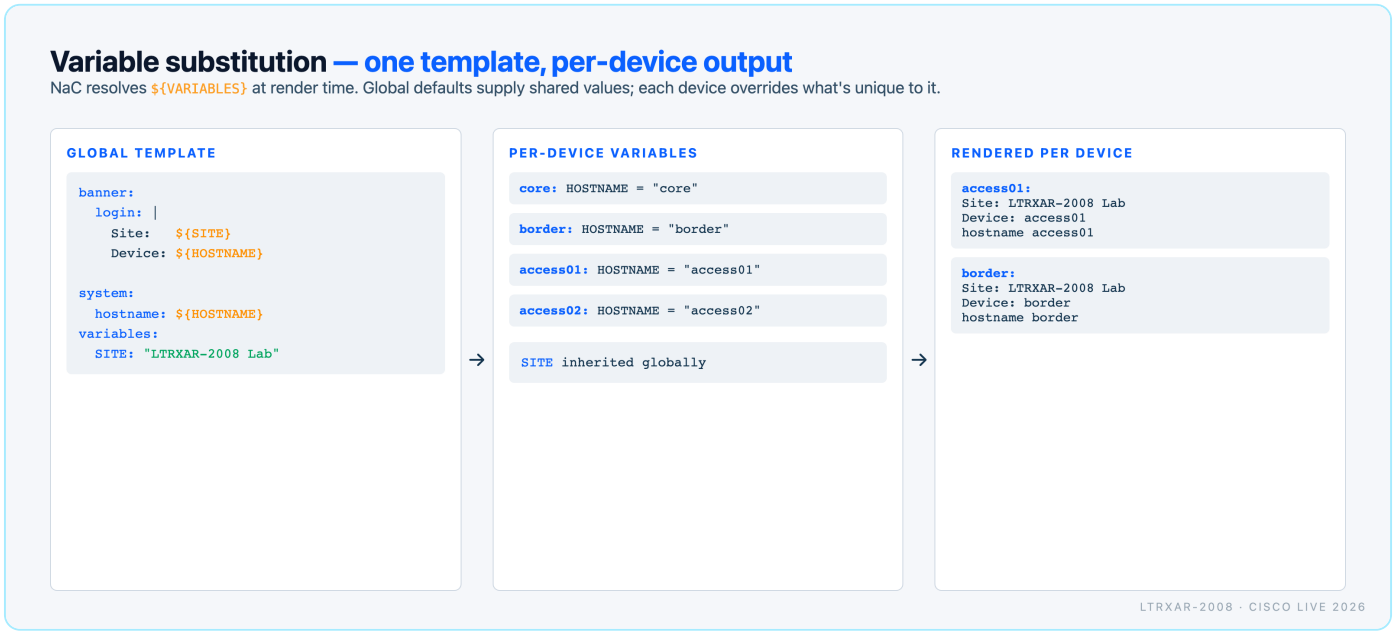

Copyright © Cisco

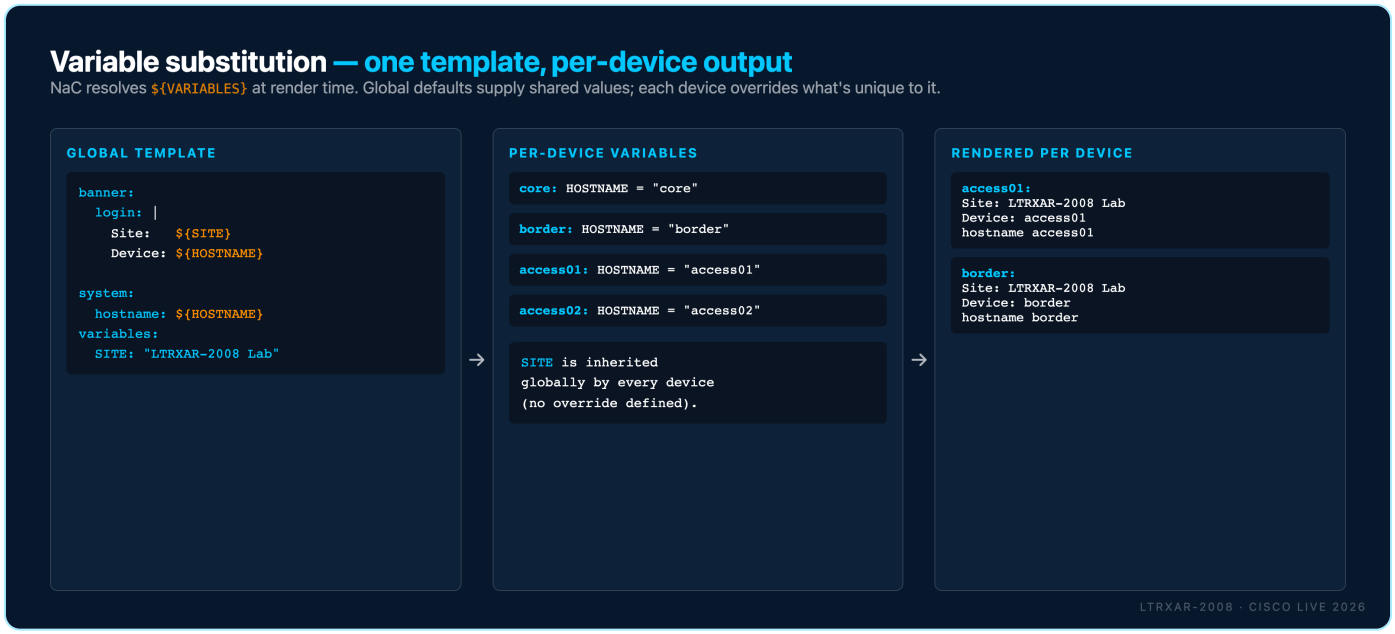
Name and hostname

The `name` attribute under `devices` identifies the device in Network as Code. The `HOSTNAME` variable value you set is what will be used in the system hostname and banner configuration. They can be the same or different, depending on your design.

How variable substitution works

When Terraform processes your configuration, it performs variable substitution at each level, for each device.





Copyright © Cisco

What `access01` actually ends up with after substitution

After you run `terraform apply` in Step 4, open `model.yaml`. The `access01` entry will have every `${HOSTNAME}` reference already replaced with the literal string – no variable markers remain in the final model:

OUTPUT **model.yaml (excerpt – access01 after resolution)**

```
iosxe:
  devices:
    - name: access01
      host: 198.18.130.11
      configuration:
        banner:
          login: |
            ######################################
            #                                    #
            #   Welcome to Network as Code Lab!  #
            #                                    #
            ######################################
            Device: access01              # ← ${HOSTNAME} resolved
        system:
          hostname: access01              # ← ${HOSTNAME} resolved
```

Substitution happens **once, at merge time**, before the Terraform provider ever opens a connection to a device. By the time anything hits NETCONF, the payload is fully-resolved concrete YAML – no `${...}` strings ever travel to the device. The `model.yaml` file is the authoritative record of "what was actually sent."

Running `access02`, `core`, and `border` through the same logic produces `hostname: access02`, `hostname: core`, and `hostname: border` respectively – same template, four different results.

Variable precedence

Variables can be defined at multiple levels. When the same variable is defined at different levels, the more specific level takes precedence:

1. **Device variables** (highest precedence) – Override everything
2. **Device group variables** (medium precedence) – Override global
3. **Global variables** (lowest precedence) – Default values

This allows you to define default values globally and override them per device or device group when needed.

Variable Precedence Example

The config below is only an example, you do not need to add this in your lab.

FILE **example: variable precedence (illustrative)**

```
---
iosxe:
  global:
    variables:
      TIMEZONE: UTC  # Global default
  devices:
    - name: example-device
      variables:
        TIMEZONE: America/New_York  # Overrides global default
```


Copyright © Cisco

Step 4: Apply configuration

Open your WSL Ubuntu terminal and navigate to your project directory:

TERMINAL · WSL/Ubuntu cisco@wkst1:~$

```
cd ~/nac-iosxe
```

Optionally, preview the changes Terraform will make:

TERMINAL · WSL/Ubuntu cisco@wkst1:~/nac-iosxe$

```
terraform plan
```

In the plan output, look for **four** separate hostname changes - one per device - even though you only edited **one** banner template in `global.nac.yaml`. That's variable substitution producing four distinct renders from one source, which is the whole point of this task.

Apply the configuration:

TERMINAL · WSL/Ubuntu cisco@wkst1:~/nac-iosxe$

```
terraform apply
```

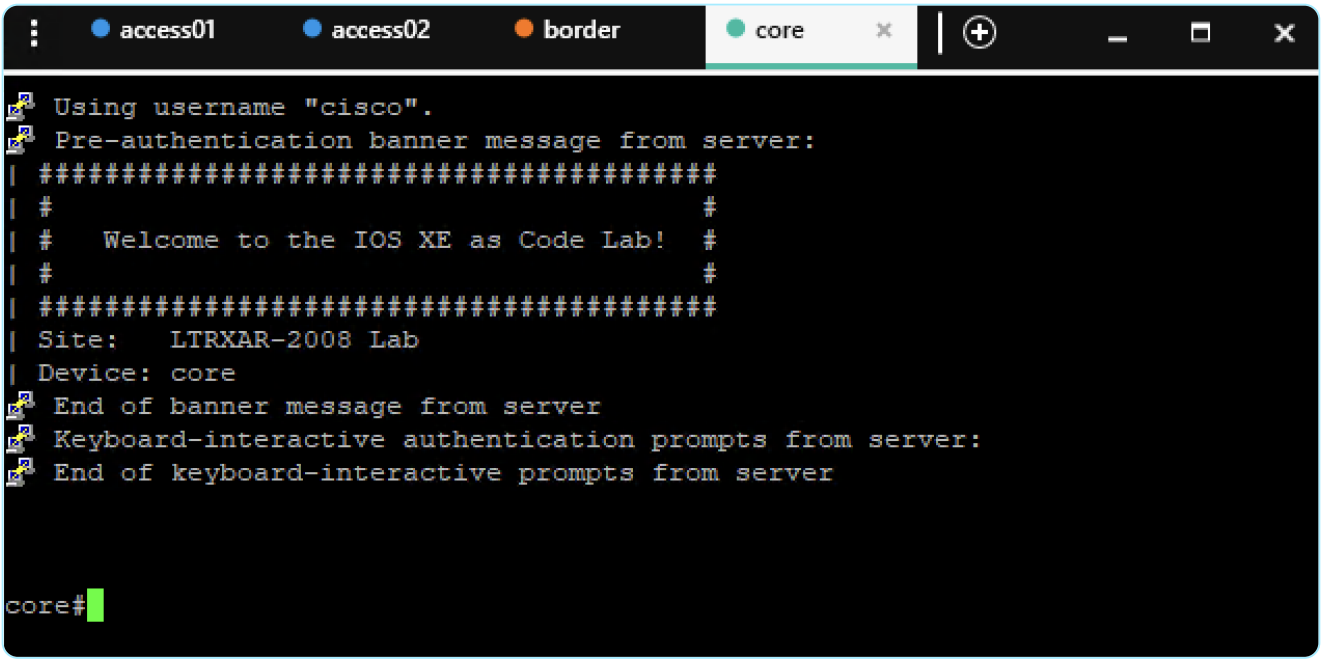
When prompted, type `yes` to confirm the deployment.

Step 5: Verify variable substitution

After Terraform completes successfully, verify the configuration was applied correctly on each device.

Use Solar-PuTTY to connect and verify:

1. Open **Solar-PuTTY** from your desktop
2. Connect to the **core** switch
3. Verify the updated hostname and banner
4. Repeat for other switches (**border**, **access01**, **access02**) to see their specific hostnames



Copyright © Cisco

Each device shows its own hostname in the banner, demonstrating that the same template produced device-specific results.

Review the `model.yaml` file

After running `terraform apply`, open `model.yaml` in VS Code to see how variables are resolved. Each device appears with its variable values already substituted into the rendered configuration - this is the final form Terraform sends to the devices.

Common variable use cases

Variables are powerful for many scenarios beyond hostnames (device identity). Here are some common use cases:

Use Case	Variable Example	Benefit
Network segments	`${MGMT_VLAN}`, `${DATA_VLAN}`	Easy VLAN changes across devices
IP addressing	`${LOOPBACK_IP}`, `${GATEWAY}`	Device-specific IP configuration
Credentials	`${SNMP_COMMUNITY}`	Centralized credential management
Thresholds	`${LOG_LEVEL}`, `${TIMEOUT}`	Environment-specific tuning

Environment variables

Environment variables can also be used in Network as Code configurations. They are defined outside of the configuration files and can be referenced using the syntax below:

Environment Variable Example

The config below is only an example, you do not need to add this in your lab.

FILE **example: !env reference (illustrative)**

```
---
iosxe:
  devices:
    - name: example-device
      system:
        enable_secret: !env ENABLE_SECRET
        enable_secret_type: "0"
```

In the example above, the `ENABLE_SECRET` environment variable is referenced and used for the device's enable secret.

Security Best Practice

Using environment variables is good practice for sensitive information like passwords.

Copyright © Cisco

What you've accomplished

In this task, you have:

- Learned how variables work in Network as Code configurations
- Updated the global configuration to use a variable placeholder
- Defined device-specific variable values
- Understood variable substitution and precedence rules
- Applied and verified variable-based configuration

What's next

Tasks 07-09 cover the three template types (`model` , `file` , `cli`) and are **optional** - they demonstrate advanced patterns but aren't prerequisites for any later task. If you're short on time, jump straight to Task 10 - Schema validation. Otherwise, continue with Task 07.

Task 05 - Single device config

Task 07 - Templates 'model'

Copyright © Cisco

Task 07 - Templates type `model` (Optional)

Estimated Time to Complete: ~10 minutes

In this task you'll use **templates of type** `model` to define reusable YAML-based configuration blocks that can be applied to multiple devices. Templates promote reuse, reduce duplication, and keep configuration consistent across the network.

What you'll learn

By the end of this task you will have:

- Written a reusable `model`-type template (`access_switch_vlans`) that defines three VLANs
- Referenced the template from the `ACCESS_SWITCHES` device group
- Verified the VLANs expanded onto both `access01` and `access02` with `show vlan brief`

Templates

Templates in Network as Code allow you to define configuration once and apply it to multiple devices by reference. Instead of repeating the same configuration in each device's YAML file, you define a template and simply reference it where needed. This works for any type of configuration - VLANs, interfaces, security policies, QoS settings, and more.

A template can be referenced at the individual device level, device group level, or even globally.

As described in the IOS XE Template documentation, templates provide:

- **Reusability**: Define configuration once, use it many times
- **Consistency**: Ensure identical configuration across devices
- **Maintainability**: Update template in one place, changes propagate everywhere
- **Modularity**: Keep configuration organized in separate, purpose-specific files

Template types:

Type	Description	Use Case
model	YAML-based configuration template	Standard configurations (VLANs, ACLs, etc.) ← *This chapter*
file	External `.tftpl` template files	Large configurations stored separately ← *Task 08*
cli	Raw CLI commands	IOS XE features not supported in the IOS XE as Code data model ← *Task 09*

WHICH TEMPLATE TYPE DOES THIS TASK COVER?

Templates come in three flavors - `model` , `file` , and `cli` - and you'll meet all three across the next three tasks:

- **This task (Task 07)** introduces `model` templates: pure YAML against the data model, schema-validated, the right default for straightforward reuse.
- **Task 08** introduces `file` templates (`.tftpl`), which add loops and conditionals on top of the same reference-by-name mechanic.
- **Task 09** introduces `cli` templates - the escape hatch for features not yet in the data model.

You'll see a side-by-side comparison of `device_groups` vs `model` vs `file` templates inside Task 08, and a final "which type should I pick?" decision tree at the end of Task 09 - once you've actually used all three.

Copyright © Cisco

Use case: standard VLANs for access switches

Access switches typically share the same VLAN configuration – they need identical VLANs for user traffic, voice, and management. Instead of defining VLANs separately for **access01** and **access02**, you'll create a single template and apply it to both devices.

VLANs to configure:

- VLAN 10: `DATA` – User data traffic
- VLAN 20: `VOICE` – VoIP traffic
- VLAN 99: `MGMT` – Management traffic

> **Why use Templates here?**
>
> Using a template for VLANs ensures that both access switches have identical VLAN configurations. If you need to add or modify VLANs in the future, you can do so in one place (the template) rather than updating each device's configuration individually.
>
> You may wonder what's the difference between using a **template** versus a **device group**. The key distinction is that templates focus on **what configuration** to apply (the VLANs), while device groups focus on **which devices** receive that configuration (the access switches). In most cases, you can combine both approaches.

Step 1: Create the template file

First, create the file using your **WSL Ubuntu terminal**:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

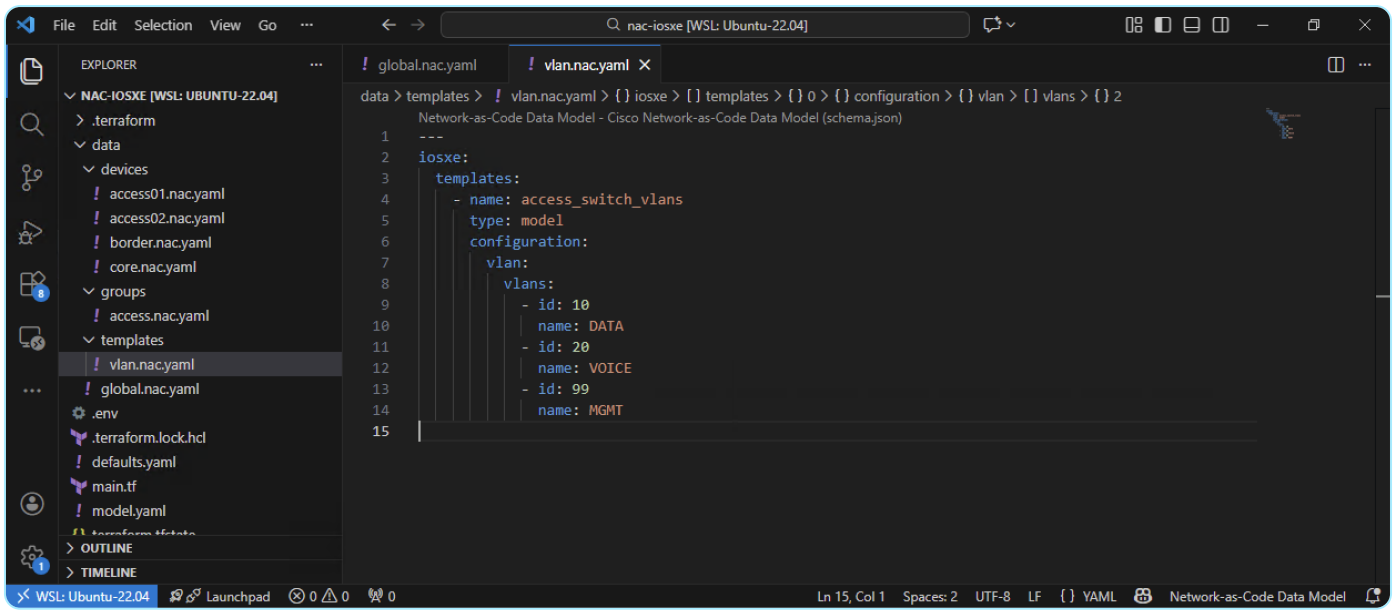
```
touch ~/nac-iosxe/data/templates/vlan.nac.yaml
```

Then open `data/templates/vlan.nac.yaml` in VS Code and add the following content. This file defines a reusable VLAN template:

FILE **data/templates/vlan.nac.yaml**

```yaml
---
iosxe:
  templates:
    - name: access_switch_vlans
      type: model
      configuration:
        vlan:
          vlans:
            - id: 10
              name: DATA
            - id: 20
              name: VOICE
            - id: 99
              name: MGMT
```


Copyright © Cisco



CONFIGURATION BREAKDOWN

Let's break down the key elements:

Template Definition:

- `templates:` - List of template definitions at the top level
- `name: access_switch_vlans` - Unique identifier for this template
- `type: model` - Indicates this is a YAML-based configuration template
- `configuration:` - Contains the actual configuration to be applied

VLAN Configuration:

- `vlan:` - VLAN configuration section
- `vlans:` - List of individual VLAN definitions
- `id:` - VLAN ID number (1-4094)
- `name:` - Descriptive name for the VLAN

Step 2: Apply template to access switches

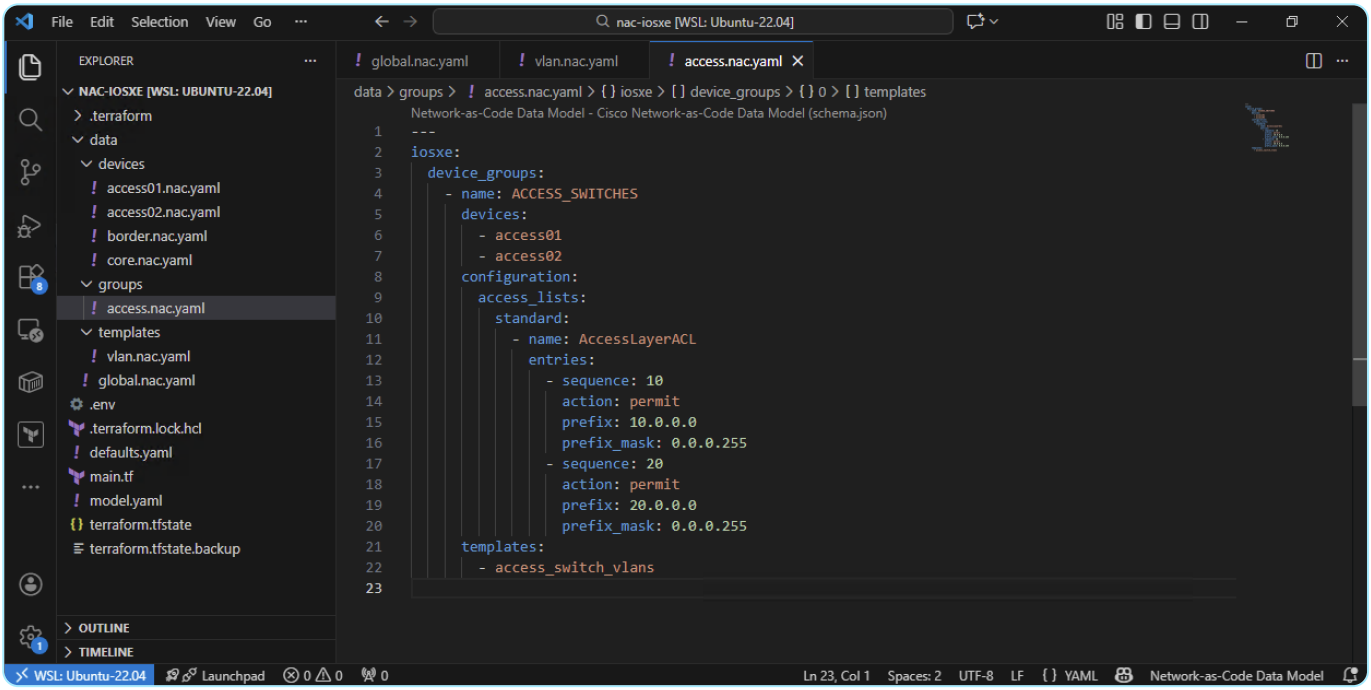
Now you need to apply the template to the access switches. Open the existing `data/groups/access.nac.yaml` file in VS Code (this file was created in Task04) and add the `templates:` section, at the end of the YAML configuration file:

FILE **data/groups/access.nac.yaml**

```yaml
---
iosxe:
  device_groups:
    - name: ACCESS_SWITCHES
      devices:
        - access01
        - access02
      configuration:
        access_lists:
          standard:
            - name: AccessLayerACL
              entries:
                - sequence: 10
                  action: permit
                  prefix: 10.0.0.0
                  prefix_mask: 0.0.0.255
                - sequence: 20
                  action: permit
```


Copyright © Cisco

```
                prefix: 20.0.0.0
                prefix_mask: 0.0.0.255
        templates:
          - access_switch_vlans
```



What was added

- `templates:` - New section to apply templates to all switches in the `ACCESS_SWITCHES` device group
- `access_switch_vlans` : Reference to the VLAN template defined in `templates/vlan.nac.yaml`

The template reference is added alongside the existing `AccessLayerACL` configuration. After running `terraform apply`, both the ACL and VLAN configuration will be present on **access01** and **access02**.

VERIFY PROJECT STRUCTURE

At this point, your `data/` folder should contain these files:

OUTPUT **Project layout**

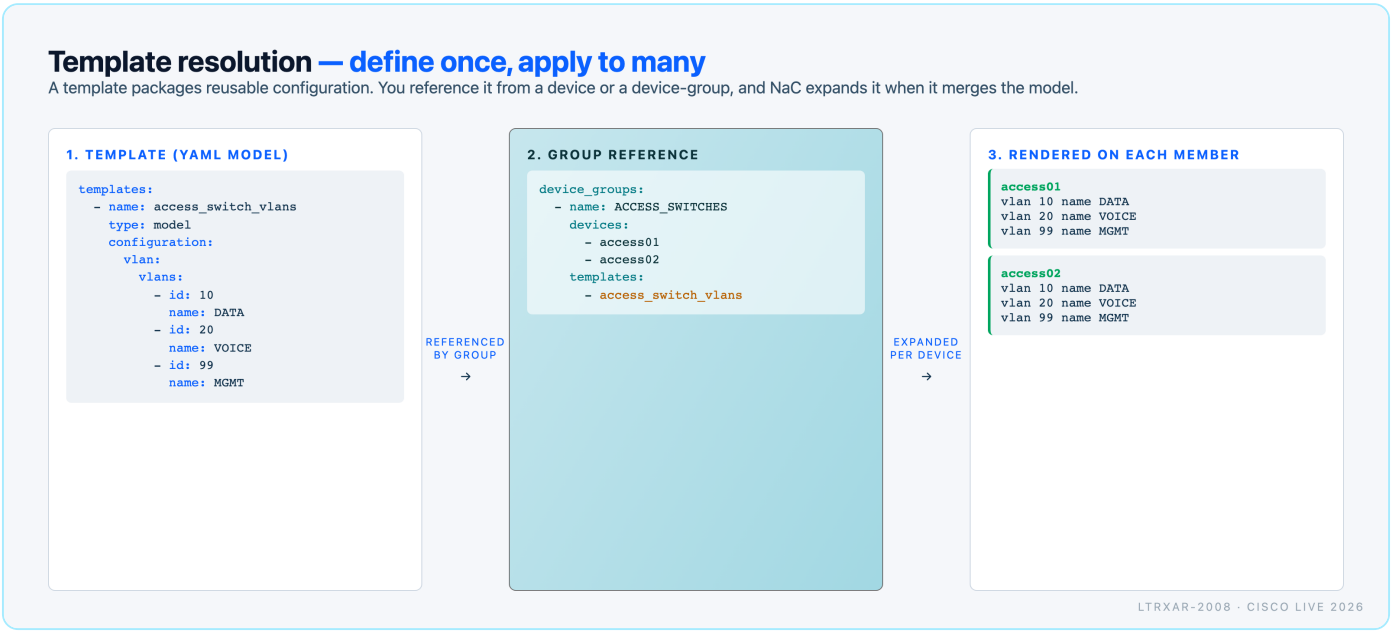
```
/home/cisco/nac-iosxe/
├── .env
├── main.tf
└── data/
    ├── devices/
    │   ├── access01.nac.yaml   # Task02: access01 registration
    │   ├── access02.nac.yaml   # Task02: access02 registration
    │   ├── border.nac.yaml     # Task02: border registration
    │   └── core.nac.yaml       # Task02 + Task05: core + Loopback0
    ├── groups/
    │   └── access.nac.yaml     # Task04 + Task07: ACL + VLAN template
    ├── templates/
    │   └── vlan.nac.yaml       # Task07: VLAN template (type: model)
    └── global.nac.yaml         # Task03: Global banner + hostname
```

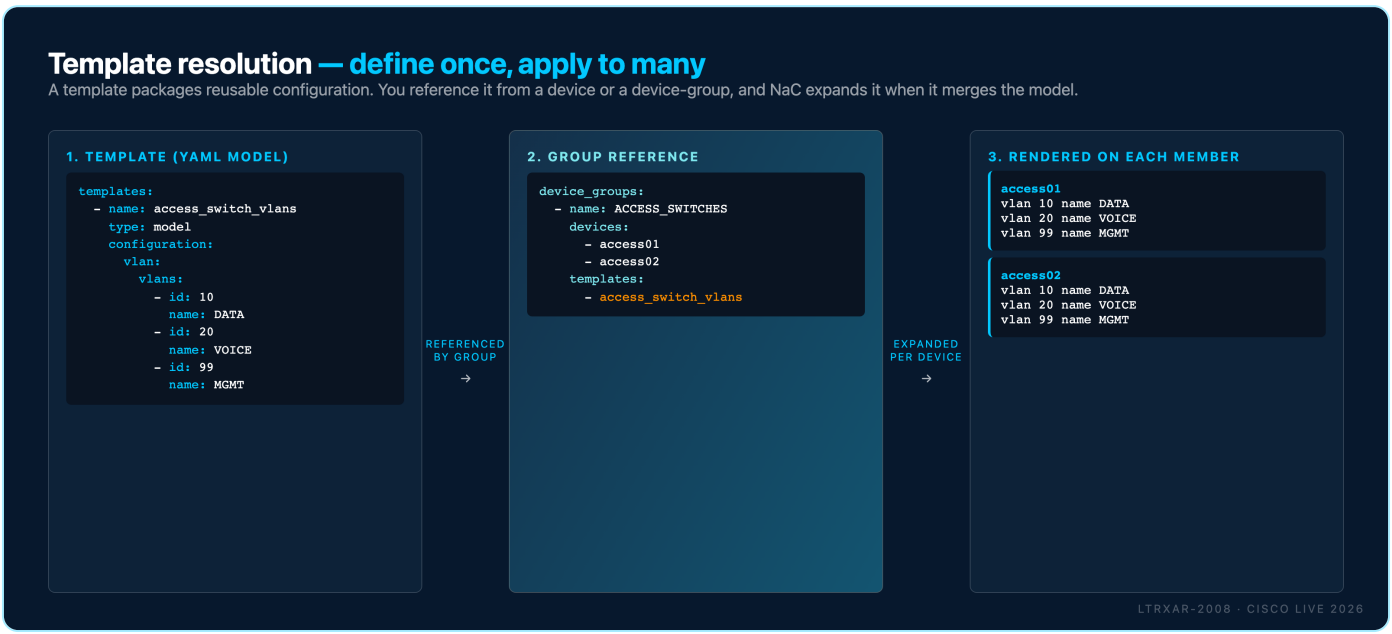

Copyright © Cisco

How templates work

When Network as Code processes your configuration:

1. **Template Resolution**: NaC reads the `templates/vlan.nac.yaml` file and loads the `access_switch_vlans` template, defined under `iosxe: templates`
2. **Device Group Processing**: NaC reads `groups/access.nac.yaml` and finds the `access_switch_vlans` template applied to the `ACCESS_SWITCHES` group
3. **Configuration Merge**: For **access01** and **access02** (members of the `ACCESS_SWITCHES` group), the template's configuration is merged with their settings
4. **Deployment**: VLANs are created on both **access01** and **access02** (but not on **core** or **border**)





Copyright © Cisco

Step 3: Apply template configuration

Open your WSL Ubuntu terminal and run the following steps:

Navigate to your project directory:

TERMINAL · WSL/Ubuntu cisco@wkst1:~$

```
cd ~/nac-iosxe
```

Optionally, preview the changes Terraform will make:

TERMINAL · WSL/Ubuntu cisco@wkst1:~/nac-iosxe$

```
terraform plan
```

Apply the configuration:

TERMINAL · WSL/Ubuntu cisco@wkst1:~/nac-iosxe$

```
terraform apply
```

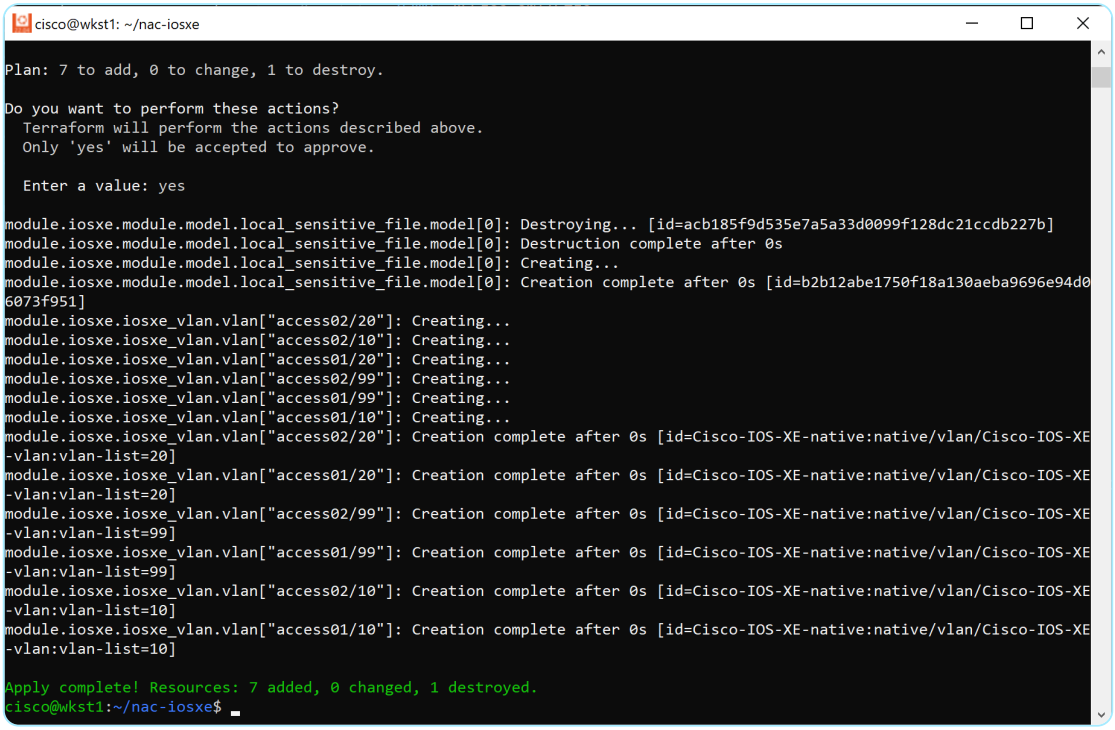
When prompted, type `yes` to confirm the deployment. Terraform will create the three VLANs on both **access01** and **access02** switches.

What to observe in the plan output:

- Terraform shows VLAN creation for **access01**
- Terraform shows VLAN creation for **access02**
- Both devices receive identical VLAN configuration

> **View the Merged Model**
>
> After running `terraform apply`, open the `model.yaml` file in VS Code to see how templates are rendered and merged with device configurations into a single data model. This is the same file used by Robot Framework for post-change validation in Task11.

Copyright © Cisco



Step 4: Verify template configuration

After successfully running `terraform apply`, verify that the VLANs were deployed to both access switches.

Use Solar-PuTTY to connect and verify:

1. Open **Solar-PuTTY** from your desktop
2. Connect to the **access01** switch
3. Run the verification command below
4. Disconnect and repeat for **access02** switch

Use the following command on both **access01** and **access02** switches to verify the VLANs:

DEVICE CLI · PuTTY **access01, access02**

```
show vlan brief
```

OUTPUT **Expected output on both switches**

```
access01#show vlan brief

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Gi1/0/1, Gi1/0/2, Gi1/0/3, Gi1/0/4,
                                                ...
                                                Gi1/0/21, Gi1/0/22, Gi1/0/23, Gi1/0/24
10   DATA                             active
20   VOICE                            active
99   MGMT                             active
1002 fddi-default                     act/unsup
1003 token-ring-default               act/unsup
1004 fddinet-default                  act/unsup
1005 trnet-default                    act/unsup
access01#
```

You should see all three VLANs (`10-DATA` , `20-VOICE` , `99-MGMT`) configured on both devices.

Copyright © Cisco

Templates vs other configuration methods

Here's a comparison of when to use templates versus other configuration approaches:

Method	Best For	Examples
Global	Settings that apply to ALL devices	Login banners, NTP, Syslog
Device Group	Role or location based settings for device groups	ACLs for access layer, routing for core, timezone for site
Device	Unique settings for one device	Management IP hosts, special features
Template	Reusable configurations across selected devices	Standard VLANs, interface templates

Templates give you fine-grained control - you choose exactly which devices get the template configuration by adding the template reference to each device.

Applying multiple templates

One of the most powerful features of templates is the ability to apply **multiple templates** to a device. This allows you to build modular, composable configurations where each template handles a specific aspect of the configuration.

For example, access switches might need:

- **VLAN configuration** (from `example_template_vlans`)
- **QoS policies** (from `example_template_qos`)
- **Security settings** (from `example_template_security`)

Using device groups (as shown in this task), you can apply multiple templates to all group members:

Example: Working with Multiple Templates

The config below is only an example, you do not need to add this in your lab.

FILE **example: applying multiple templates to a group (illustrative)**

```
---
iosxe:
  device_groups:
    - name: EXAMPLE_GROUP_SWITCHES
      devices:
        - example-switch01
        - example-switch02
        - example-switch03
      templates:
        - example_template_vlans
        - example_template_qos
        - example_template_security
```

Benefits of using templates

1. **Single Source of Truth**: VLAN definitions exist in one place
2. **Easy Updates**: Need to add `VLAN 30`? Update the template once, all devices get it
3. **Selective Application**: Not all devices need the same VLANs - only reference the template where needed
4. **Combine Multiple Templates**: A device can reference multiple templates for different configuration aspects
5. **Separation of concerns**: With multiple templates, each can handle one configuration domain

Copyright © Cisco

What you've accomplished

In this task, you have:

- Learned about templates and their benefits for Network as Code
- Created a reusable VLAN template (`access_switch_vlans`)
- Applied the template to multiple access switches
- Verified consistent VLAN deployment across devices
- Understood when to use templates vs global/group/device configurations

What's next

Task 08 introduces the `file` template type (external `.tftpl` files with Jinja-style loops and conditionals - the template you'd reach for when `model` templates aren't expressive enough). If you're only sampling one template type, skipping ahead to Task 10 - Schema validation is fine.

Task 06 - Variables

Task 08 - Templates 'file'

Copyright © Cisco

Task 08 - Templates type `file` (Optional)

Estimated Time to Complete: ~15 minutes

In this task you'll use **templates of type** `file` to reference external template files that use Terraform templating syntax (`.tftpl`). File templates are the right tool when you need loops, conditionals, or complex structures that plain YAML can't express cleanly.

For detailed documentation, see: IOS XE as Code Template Documentation

What you'll learn

By the end of this task you will have:

- Authored a `.tftpl` template file using Terraform templating syntax - loops (`%{ for }`), variable interpolation (`${ }`), whitespace stripping
- Referenced the file template from a device-level configuration on `border`
- Deployed an eBGP session to the pre-configured `isp` peer (AS 65001) and seen the route to `8.8.8.0/24` appear

File templates

File templates reference external `.tftpl` files that use **Terraform templating syntax**. This is ideal for:

- **Dynamic configurations**: Configurations with loops and conditionals
- **Complex structures**: Multi-line configurations that need programmatic generation

Terraform Templating Syntax:

Syntax	Purpose	Examples
`${ }`	Variable interpolation	`${BGP_AS_NUMBER}`
`%{ }`	Control structures	
`%{ for }`	For loop	`%{ for NEIGHBOR in BGP_NEIGHBORS }` , `%{ endfor }`
`%{ if }`	Conditional	`%{ if OPTION == "A" }` , `%{ else }` , `%{ endif }`
`~`	Whitespace stripping	`%{~ endfor ~}`

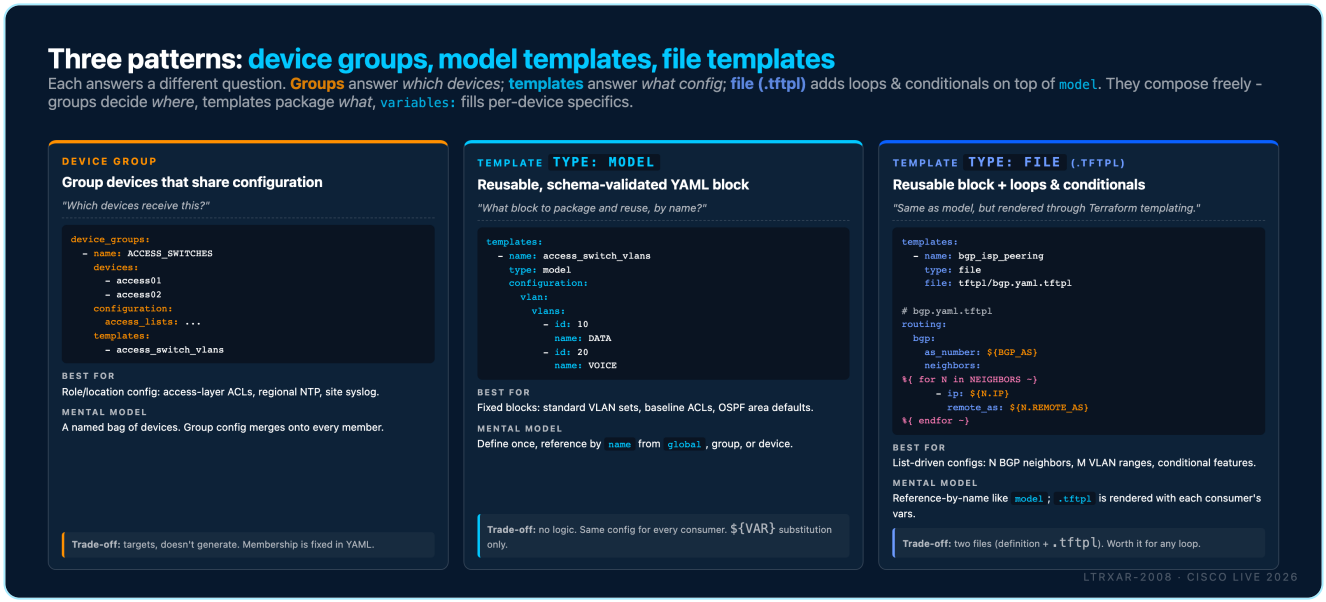
Template Types (reminder):

Type	Description	Use Case
model	YAML-based configuration template	Standard configurations (VLANs, ACLs, etc.) ← *Task07*
file	External `.tftpl` template files	Dynamic configs with variables ← *This task*
cli	Raw CLI commands	IOS XE features not in the IOS XE as Code data model ← *Task09*

Copyright © Cisco

Three patterns side-by-side: device groups, `model` templates, `file` (.tftpl) templates

By Task 08 you've now seen three distinct mechanisms for reusing configuration across devices: the **device groups** from Task 04, the `model` **templates** from Task 07, and the `file` **(.tftpl) templates** introduced in this task. They look superficially similar - all three reduce duplication - but they answer different questions and compose differently. The diagram below summarizes the distinction; the prose underneath walks through each one.



Each pattern answers a different question.

- **Device group** - answers ***"Which devices receive this configuration?"*** A device group is a named bag of devices (in this lab, `ACCESS_SWITCHES = [access01, access02]`). Configuration written directly under the group's `configuration:` key merges onto every member at apply time. A group can also reference templates, which are then expanded for each member. *Best for:* role-based or location-based config - access-layer ACLs, regional NTP/timezone, site-wide syslog targets. *Trade-off:* membership is fixed in YAML; the group definition itself doesn't generate any configuration, it just **targets** it.
- **Template** `type: model` - answers ***"What block of config do I want to package and reuse, by name?"*** A model template is a chunk of standard NaC YAML wrapped in a name. You define it once (e.g. `access_switch_vlans` from Task 07), then reference it by name from any combination of `global`, a device group, or individual devices. Because it's plain YAML against the data model, the schema validates it (Task 10) just like inline configuration. *Best for:* fixed, repeatable blocks - standard VLAN sets, baseline ACLs, common interface profiles, OSPF area defaults. *Trade-off:* no logic. Every consumer of the template gets the same config (variables can substitute simple values like `${HOSTNAME}`, but there are no loops or conditionals).
- **Template** `type: file` **(.tftpl)** - answers the same ***"what block?"*** question as `model`, but renders the block through Terraform's templating engine first. The template definition lives in YAML (`templates: - name: ...; type: file; file: tftpl/bgp.yaml.tftpl`), and the actual configuration body lives in a separate `.tftpl` file that supports variable interpolation (`${VAR}`), loops (`%{ for x in LIST }`), and conditionals (`%{ if ... }`). At apply time, NaC reads each consumer's `variables:` block and renders the `.tftpl` against those values, then merges the rendered YAML into the data model. *Best for:* list-driven configs - N BGP neighbors per device, M VLAN ranges, conditional features keyed off device variables. *Trade-off:* two files instead of one (the YAML template definition plus the `.tftpl` body). The moment you need a loop or conditional, the extra file is worth it.

They compose, they don't compete. A device group can apply both a `model` and a `file` template to its members at the same time, and a single device can override what the group applies. The canonical pattern in production is: **groups decide where**, **templates decide what**, and **variables fill in the per-device specifics** (IP, AS number, neighbor list).

Copyright © Cisco

Quick mental rule

- Need to point a chunk of configuration at a *set of devices*? Reach for a **device group**.
- Need to *package and name* a chunk of configuration so multiple places can reference it? Reach for a `model` **template**.
- Need that same packaged chunk to *expand into a list* or *react to per-device variables with loops/conditionals*? Promote the `model` template to a `file` **template**.

Use case: BGP configuration on border switch

In this example, you'll configure BGP on the **border** switch for peering with ISP providers. The template will use variables to define BGP neighbors dynamically.

Lab Scenario

The **border** switch connects to two ISP providers:

- **isp** (`198.18.100.1`) - Currently active and pre-configured in the lab
- **isp-x** (`198.18.100.5`) - Placeholder for a future connection

When you verify the BGP configuration, the **isp neighbor will show as Established**, while **isp-x will show as Idle** (since the remote end is not yet configured).

Step 1: Create the template file

First, create the `tftpl` directory and the template file using your **WSL Ubuntu terminal**:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
mkdir -p ~/nac-iosxe/tftpl
```

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
touch ~/nac-iosxe/tftpl/bgp.yaml.tftpl
```

Then open `tftpl/bgp.yaml.tftpl` in VS Code and add the following content:

FILE **tftpl/bgp.yaml.tftpl**

```
routing:
  bgp:
    as_number: ${BGP_AS_NUMBER}
    log_neighbor_changes: true
    neighbors:
%{ for NEIGHBOR in BGP_NEIGHBORS ~}
      - ip: ${NEIGHBOR.IP}
        remote_as: ${NEIGHBOR.REMOTE_AS}
        description: "${NEIGHBOR.DESCRIPTION}"
        timers_keepalive: 60
        timers_holdtime: 180
        timers_minimum_neighbor_holdtime: 30
%{ endfor ~}
    address_family:
      ipv4_unicast:
        neighbors:
%{ for NEIGHBOR in BGP_NEIGHBORS ~}
          - ip: ${NEIGHBOR.IP}
            activate: true
%{ endfor ~}
```


Copyright © Cisco

This template uses:

- `routing: bgp:` - BGP configuration must be nested under `routing` per the IOS XE as Code data model
- `${BGP_AS_NUMBER}` - Variable for the local AS number
- `%{ for NEIGHBOR in BGP_NEIGHBORS }` - Loop through list of neighbors (twice, in two sections)
- `%{ endfor }` - End of the loops
- `~` - Whitespace stripping, cleans up extra spaces/newlines in the rendered output
- `${NEIGHBOR.IP}`, `${NEIGHBOR.REMOTE_AS}`, `${NEIGHBOR.DESCRIPTION}` - Access neighbor variables for attributes

Template Location

The file `bgp.yaml.tftpl` is located in the `tftpl/` folder, outside of the main `data/` folder. Even though the `tftpl/` folder is not included in the module configuration in `main.tf`, the template's content is still accessible because it is referenced in the template definition file that you'll create next.

Step 2: Create the template definition file

Create a new file `data/templates/bgp.nac.yaml` that defines the template:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
touch ~/nac-iosxe/data/templates/bgp.nac.yaml
```

Then open `data/templates/bgp.nac.yaml` in VS Code and add the following content:

FILE **data/templates/bgp.nac.yaml**

```
---
iosxe:
  templates:
    - name: bgp_isp_peering
      type: file
      file: tftpl/bgp.yaml.tftpl
```

This separates the template definition from the device configuration, making it easier to manage and reuse.

Step 3: Apply the template to border device

Now open the existing `data/devices/border.nac.yaml` file in VS Code (this was created as a placeholder in Task05) and add the template reference with variables:

FILE **data/devices/border.nac.yaml**

```
---
iosxe:
  devices:
    - name: border
      host: 198.18.130.20
      variables:
        HOSTNAME: border
        BGP_AS_NUMBER: 65000
        BGP_NEIGHBORS:
          - IP: 198.18.100.1
            REMOTE_AS: 65001
            DESCRIPTION: eBGP to isp - Production
          - IP: 198.18.100.5
            REMOTE_AS: 65002
            DESCRIPTION: eBGP to isp-x - Future Migration
      templates:
        - bgp_isp_peering
      configuration:
        interfaces:
          ethernets:
            - type: GigabitEthernet
```


Copyright © Cisco

```
              id: "1"
              description: "isp G0/1"
              shutdown: false
              ipv4:
                address: 198.18.100.2
                address_mask: 255.255.255.252
```

What's in this configuration:

- `devices:` - Device-specific configuration
- `name: border` - The **border** switch where BGP will be configured
- `variables:` - Variables that will be substituted into the template
- `templates:` - References the `bgp_isp_peering` template defined in `templates/bgp.nac.yaml`
- `configuration: interfaces` - Interface configuration for the ISP connection. Refer to the IOS XE as Code data model

Variable Breakdown:

- `BGP_AS_NUMBER: 65000` : **border** switch AS number
- `BGP_NEIGHBORS` : List of ISP neighbors:
  - **isp** (65001): Active production peer
  - **isp-x** (65002): Placeholder for future network migration
- `IP` , `REMOTE_AS` , `DESCRIPTION` : Attributes used in the template for each neighbor

> **Device-Level Templates**
>
> Templates can be applied directly to individual devices, as shown here. This is ideal when a template is specific to a single device. For templates shared across multiple devices, you can use device groups (as shown with VLANs in Task07).

Step 4: Deploy the configuration

Open your WSL Ubuntu terminal and run the following steps:

Step 1: Navigate to your project directory:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
cd ~/nac-iosxe
```

Step 2: Optionally, preview the changes Terraform will make:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform plan
```

Step 3: Apply the configuration:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform apply
```

When prompted, type `yes` to confirm the deployment.

Copyright © Cisco

View the Merged Model

After running `terraform apply`, open the `model.yaml` file in VS Code to see how the BGP template file is rendered with your variables and merged into the complete data model. This shows exactly what configuration will be applied to the device.

Step 5: Verify BGP configuration

Expected: Idle for 30-60 seconds, then Established

BGP sessions do not come up instantly. After `terraform apply` finishes, expect a 30-60 second window before the neighbor state transitions through `Idle` -> `Active` -> `OpenSent` -> `Established`. If your first `show ip bgp summary` looks idle, wait a minute and run it again - this is normal protocol behavior, not a failure.

Use **Solar-PuTTY** to connect to the **border** switch and verify the BGP configuration:

DEVICE CLI · PuTTY **border**

```
show ip bgp summary
```

OUTPUT **Expected Output**

```
border#show ip bgp summary
BGP router identifier 198.18.130.20, local AS number 65000
...

Neighbor        V           AS MsgRcvd MsgSent   TblVer  InQ OutQ Up/Down  State/PfxRcd
198.18.100.1    4        65001      23      23        2    0    0 00:16:54        1
198.18.100.5    4        65002       0       0        1    0    0 never    Idle
border#
```

Understanding the output

Neighbor	Status	Explanation
198.18.100.1 (isp)	**Established** (shows prefix count)	Active peer - **isp** is pre-configured in the lab
198.18.100.5 (isp-x)	**Idle**	Expected - **isp-x** is a placeholder for future migration

The **isp** neighbor shows as **Established** with 1 prefix received, while **isp-x** shows as **Idle** since the remote end is not configured. This is expected behavior in this lab. In real-world scenarios, you often pre-configure BGP neighbors before the remote end is ready. This allows for seamless cutover during network migrations.

Lab BGP configuration

You can find the pre-configured BGP settings on the **isp** device in Appendix IV of this lab guide.

ADVERTISED NETWORK

The **isp** neighbor is pre-configured in the lab to advertise the network `8.8.8.0/24` to the **border** switch, to simulate internet connectivity. `8.8.8.8` is a loopback address configured on the **isp** device.

You can verify the received route with (same 30-60 second convergence window applies):

DEVICE CLI · PuTTY **border**

Copyright © Cisco

```
show ip route
```

OUTPUT **Expected Output**

```
border#show ip route
Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP
...

Gateway of last resort is 198.18.128.1 to network 0.0.0.0

S*    0.0.0.0/0 [1/0] via 198.18.128.1
      8.0.0.0/24 is subnetted, 1 subnets
B        8.8.8.0 [20/0] via 198.18.100.1, 00:20:04
      198.18.100.0/24 is variably subnetted, 2 subnets, 2 masks
C        198.18.100.0/30 is directly connected, GigabitEthernet1
L        198.18.100.2/32 is directly connected, GigabitEthernet1
C     198.18.128.0/20 is directly connected, GigabitEthernet2
      198.18.130.0/32 is subnetted, 1 subnets
L        198.18.130.20 is directly connected, GigabitEthernet2
border#
```

Advanced challenge - for learners who finish early

Skip this if you're running short on time

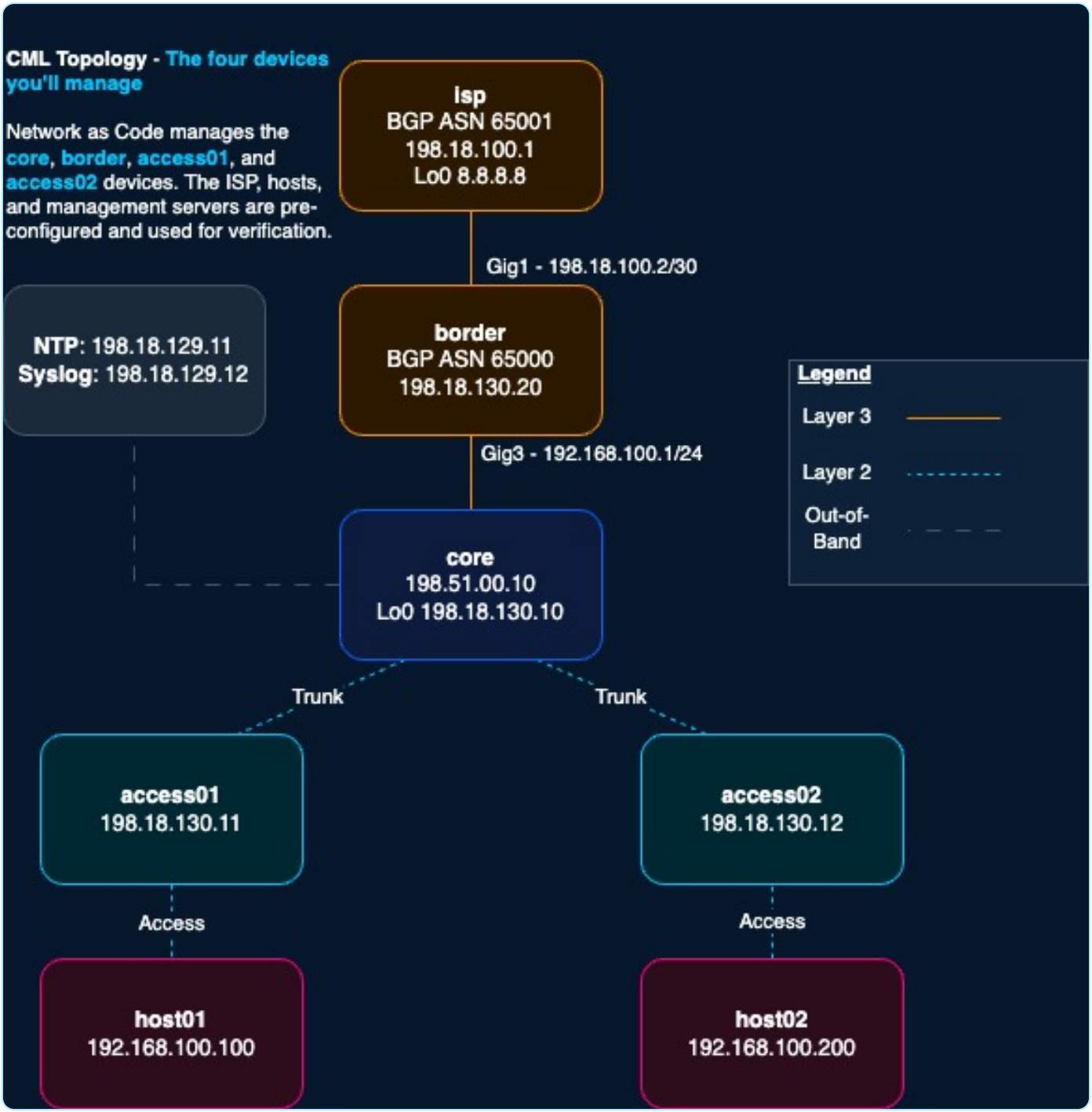
This is an optional extension inside an already-optional task. Come back to it if you've completed the recommended path and still have time left. If not, the same challenge is repeated in the **Conclusion** as something to try at home.

The goal: extend `border` so that traffic from `host01` / `host02` can actually reach the "internet" via the ISP BGP peering you just configured. Right now the BGP session is up and advertising `8.8.8.0/24`, but `border` has no interface facing the host subnet, so nothing from the hosts can get there.

Host	IP address	Default gateway
`host01`	192.168.100.100/24	192.168.100.1
`host02`	192.168.100.200/24	192.168.100.1

Your job: add an interface configuration on `border` so that `GigabitEthernet3` (facing the host network) has IP `192.168.100.1/24`. The hosts are already configured with that IP as their default gateway.

Copyright © Cisco



Primary verification (easy path - just SSH into `border`):

DEVICE CLI · PuTTY **border**

```
show ip interface brief | include GigabitEthernet3
show ip route 8.8.8.0
```

You should see `GigabitEthernet3` up with `192.168.100.1/24`, and `8.8.8.0/24` as a BGP-learned route via the ISP neighbor.

Copyright © Cisco

End-to-end verification via CML console (more work, more satisfying)

To prove actual end-to-end reachability, ping `8.8.8.8` from inside one of the hosts:

1. Open the CML web interface: https://198.18.130.34/login.
2. Log in with `guest` / `CiscoLive`.
3. Click on **NAC IOSXE-AS-CODE Topology**.
4. Right-click **host01** → **Console** → **Open Console**.
5. Log in with `cisco` / `cisco`.
6. Run: `ping 8.8.8.8`.
7. Repeat for `host02`.

This takes a minute to set up but is the only way to confirm the full end-to-end path from host → gateway → BGP → ISP loopback.

Try it yourself first. You've seen every Network as Code concept you need to solve this - per-device config, interface definitions, and the same `ethernets` list you just used for GigabitEthernet1. Browse the Ethernet data model docs if you get stuck.

Solution

Open `data/devices/border.nac.yaml` in VS Code and extend the `interfaces.ethernets` list with a second entry for `GigabitEthernet3`:

FILE **data/devices/border.nac.yaml**

```yaml
---
iosxe:
  devices:
    - name: border
      host: 198.18.130.20
      variables:
        HOSTNAME: border
        BGP_AS_NUMBER: 65000
        BGP_NEIGHBORS:
          - IP: 198.18.100.1
            REMOTE_AS: 65001
            DESCRIPTION: eBGP to isp - Production
          - IP: 198.18.100.5
            REMOTE_AS: 65002
            DESCRIPTION: eBGP to isp-x - Future Migration
      templates:
        - bgp_isp_peering
      configuration:
        interfaces:
          ethernets:
            - type: GigabitEthernet
              id: "1"
              description: "isp G0/1"
              shutdown: false
              ipv4:
                address: 198.18.100.2
                address_mask: 255.255.255.252
            - type: GigabitEthernet
              id: "3"
              description: "host network G1/0/3"
              shutdown: false
              ipv4:
                address: 192.168.100.1
                address_mask: 255.255.255.0
```

Run `terraform apply`, then verify with the `show` commands above. For bonus points, ping `8.8.8.8` from `host01` via CML console.

Copyright © Cisco

What you've accomplished

- Created a `.tftpl` template file with Terraform templating syntax in `tftpl/` folder
- Created a separate template definition file (`templates/bgp.nac.yaml`)
- Used variable interpolation (`${ }`) for dynamic values
- Used loops (`%{ for }`) for multiple BGP neighbors
- Applied templates at device level to the **border** switch
- Configured BGP peering on **border** switch for ISP connectivity

What's next

Task 09 covers the `cli` template type (raw CLI lines, last-resort for features not yet in the data model). If you've seen enough of the template story, skip ahead to Task 10 - Schema validation.

Task 07 - Templates 'model'

Task 09 - Templates 'cli'

Copyright © Cisco

Task 09 - Templates type `cli` (Optional)

Estimated Time to Complete: ~10 minutes

In this task you'll use **templates of type** `cli` to inject raw IOS XE CLI commands into your configuration. CLI templates are the escape hatch for features not yet modeled in the IOS XE as Code data model - reach for them rarely, but they're genuinely useful when you need them.

What you'll learn

By the end of this task you will have:

- Created a `cli` -type template containing a raw IOS XE command (`alias exec logs show logging`)
- Applied the template globally via `global.nac.yaml`
- Verified the alias appears on every device using `show alias`
- Understood **when** to reach for `cli` templates vs `model` / `file` (rarely - only for features outside the data model)

CLI Templates

The CLI template type allows you to include raw IOS XE CLI commands that are applied directly to devices. This can be used as a workaround for features not yet supported by the Network as Code IOS XE data model, as it accepts any valid CLI command.

Template Types (reminder):

Type	Description	Use Case
model	YAML-based configuration template	Standard configurations (VLANs, ACLs, etc.) ← *Task07*
file	External `.tftpl` template files	Large configurations stored separately ← *Task08*
cli	Raw CLI commands	IOS XE features not in the IOS XE as Code data model ← *This task*

When to use 'cli' templates:

- **Unsupported features**: Configurations not yet in the IOS XE as Code data model
- **Temporary workarounds**: Quick fixes before proper Network as Code support is added

> ⚠ **Use with Caution**
>
> Always prefer *model* and *file* templates when possible. They provide dependency handling, better validation, consistency, and they are easier to maintain.

Use case: custom alias for the `show logging` command

In this example, you'll configure a custom alias command `logs` that maps to the `show logging` command. This is a simple yet practical use case demonstrating how to use *cli* templates to add commands not directly supported by the IOS XE as Code data model.

Step 1: Create the CLI template

First, create the file using your **WSL Ubuntu terminal**:

`TERMINAL · WSL/Ubuntu` **cisco@wkst1:~/nac-iosxe$**

Copyright © Cisco

```
touch ~/nac-iosxe/data/templates/logging.nac.yaml
```

Then open `data/templates/logging.nac.yaml` in VS Code and add the following content:

FILE **data/templates/logging.nac.yaml**

```
---
iosxe:
  templates:
    - name: alias_logs
      type: cli
      content: alias exec logs show logging
```

This template:

- `type: cli` : Specifies raw CLI commands
- `content` : Contains the actual CLI command to create the alias

Step 2: Apply the template to the global configuration

Now add the alias template to your existing global configuration so that it applies to all devices.

Update `data/global.nac.yaml` to include the alias template:

FILE **data/global.nac.yaml**

```
---
iosxe:
  global:
    variables:
      SITE: "LTRXAR-2008 Lab"
    configuration:
      banner:
        login: |-
          ########################################
          #                                      #
          #   Welcome to the IOS XE as Code Lab!  #
          #                                      #
          ########################################
          Site:   ${SITE}
          Device: ${HOSTNAME}
      system:
        hostname: ${HOSTNAME}
    templates:
      - alias_logs
```

Step 3: Deploy the configuration

Open your WSL Ubuntu terminal and run the following steps:

Step 1: Navigate to your project directory:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
cd ~/nac-iosxe
```

Step 2: Optionally, preview the changes Terraform will make:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform plan
```

Step 3: Apply the configuration:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform apply
```


Copyright © Cisco

When prompted, type `yes` to confirm the deployment.

> **View the Merged Model**
>
> After running `terraform apply`, open the `model.yaml` file in VS Code to see how the alias_logs CLI template is merged with other configurations.

Step 4: Verify the configuration

Use **Solar-PuTTY** to connect to one of the configured devices (e.g., `core` switch) and verify that the alias command was successfully added.

After connecting to the device, run the following command to check if the alias was created:

DEVICE CLI · PuTTY **core**

```
show alias
```

OUTPUT **Expected Output**

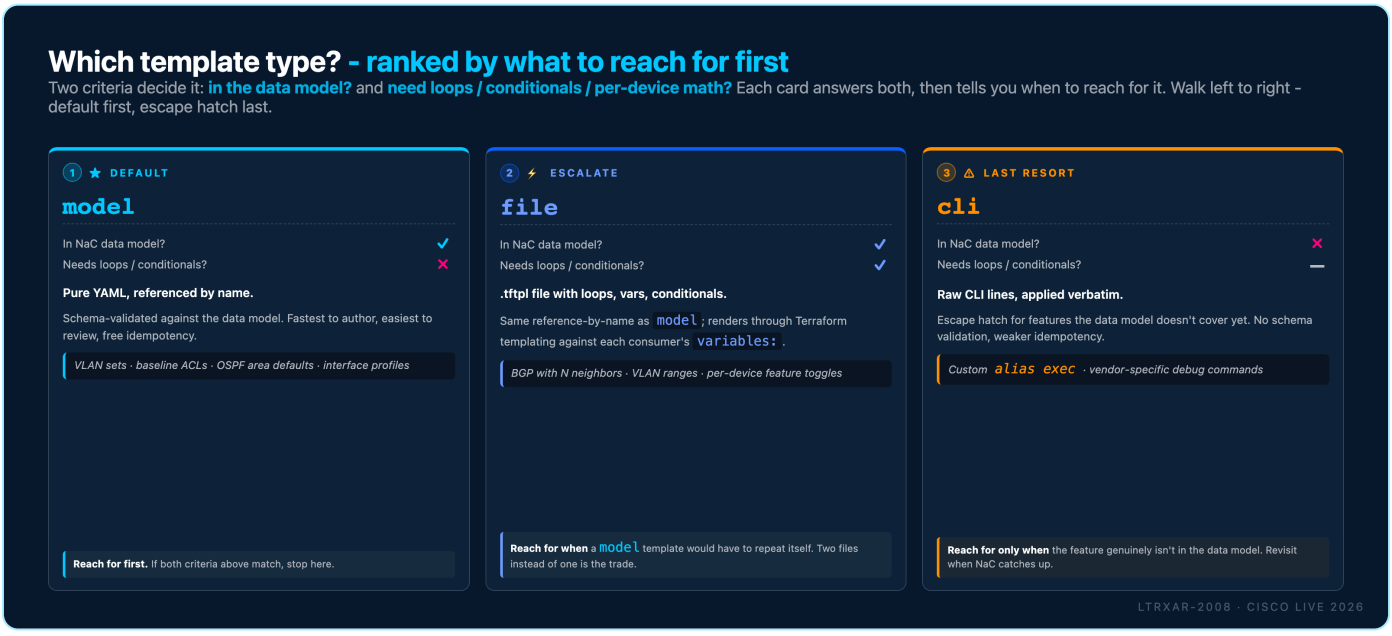
```
core#show alias
Exec mode aliases:
  h                   help
  lo                  logout
  ...
  logs                show logging

core#
```

You can also test the alias by running `logs` from exec mode.

When to use each template type

Now that you've seen all three template types (`model` in Task 07, `file` in Task 08, and `cli` in this task), here's the consolidated decision aid - two questions, three answers:



Copyright © Cisco

Rules of thumb from the tree:

- **Default to** `model` when the feature is in the data model and your config is straightforward. Fastest to write, gets free schema validation, reads as pure YAML.
- **Escalate to** `file` when you need Jinja-style loops or conditionals (e.g. "one BGP neighbor per item in this list") or when the config is long enough that a standalone `.tftpl` file is cleaner than inline YAML.
- **Reach for** `cli` **last** when the feature genuinely isn't in the data model yet. You lose schema validation and fine-grained idempotency in exchange for coverage of edge-case features.

Quick-reference scenario table:

Scenario	Recommended Type
Config that maps directly to the data model and only needs value templating	`model`
Complex config with loops, conditionals, or dynamic structures	`file`
Last resort for features not yet supported in the data model	`cli`

What you've accomplished

- Created a `cli` type template with a raw CLI command
- Applied the template to all devices via global configuration
- Deployed alias configuration

Task 08 - Templates type `file`

Task 10 - Schema validation

Copyright © Cisco

Task 10 - Pre-checks with schema validation

Estimated Time to Complete: ~15 minutes

You've been creating Network as Code YAML files and deploying them with Terraform. How do you ensure your YAML is correctly structured and contains valid data **before** hitting a device?

Pre-change validation catches errors at the YAML layer - before `terraform plan` ever opens a connection to a switch. It's the same principle as compiling code before running it: fail fast, fail cheap, fail with a clear error message.

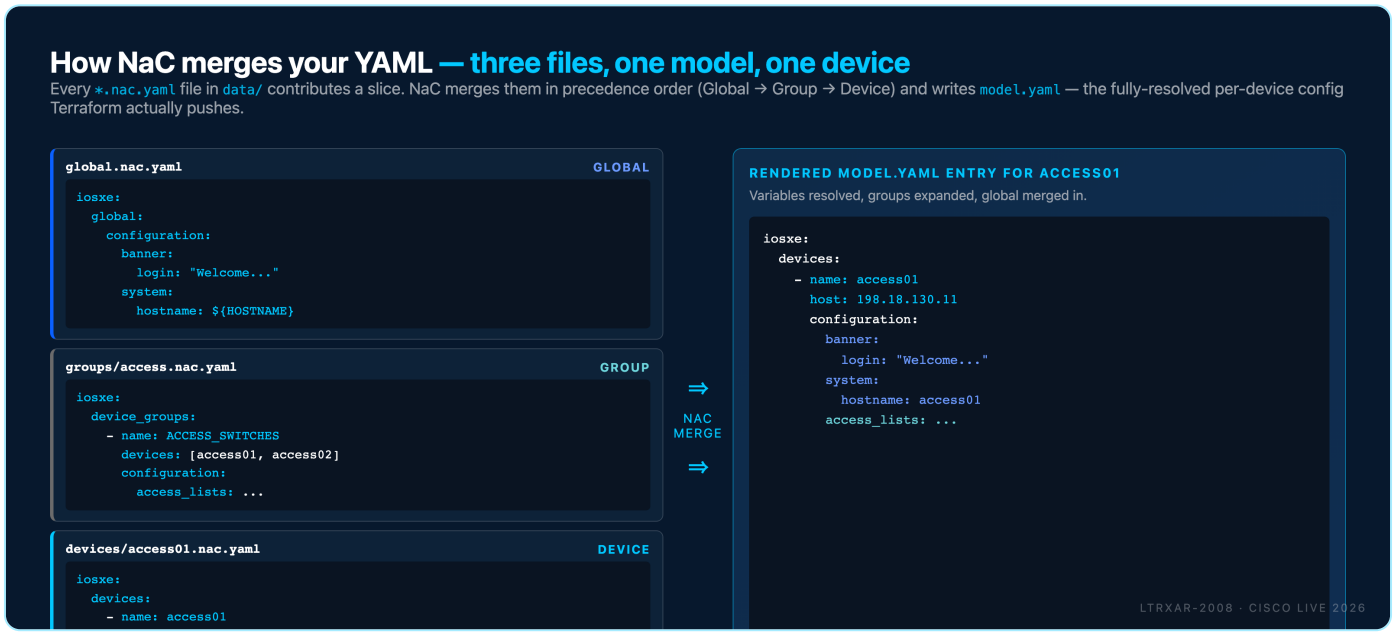
What you'll learn

By the end of this task you will have:

- Installed `nac-validate` and run it against your `data/` directory
- Read a schema-validation error message and mapped it back to the offending YAML key
- Understood the difference between **syntactic** validation (what `nac-validate` does by default) and **semantic** validation (custom rules)

First, what is Network as Code actually validating?

Before `nac-validate` runs, it's useful to see what it's validating *against*. Network as Code doesn't look at your individual YAML files in isolation - it builds a **merged data model** by combining every YAML in `data/` according to precedence (Global → Group → Device), resolving variables and template references, then producing one entry per device:



That merged model is what Terraform ultimately sends to each device (you've seen it written to `model.yaml` after each `terraform apply`). Schema validation checks the merged result - so an error might come from a bad key in a global file that only affects one group's rendered output. `nac-validate` tells you exactly which file and which path in the merged model caused the problem.

Copyright © Cisco

What is schema validation?

Schema validation verifies that your YAML configuration files:

- Follow the correct structure (proper indentation, key names, nesting)
- Contain valid data types (strings, integers, booleans)
- Use valid values (IP addresses in correct format, enums like `permit` / `deny`)
- Include all required fields
- Don't include unsupported attributes

This is similar to how a compiler checks code before running it, catching errors at "build time" rather than "run time".

The nac-validate tool

The **nac-validate** tool checks your YAML files against a schema definition. The schema acts as a contract that defines what attributes are allowed, what data types are expected, what values are valid, and which fields are mandatory vs. optional. This is called syntactic validation.

The tool can also perform semantic validation based on custom rules. These rules can check that the configuration is correct (e.g. by verifying references or conflicting IDs), enforce policies based on config best practices or check customer requirements (e.g. a custom naming convention). You'll exercise both modes in this task: schema-based syntactic validation first, then a custom semantic rule at the end.

The schema file

The complete schema for IOS XE Network as Code is documented on the Cisco NetAsCode website. For this lab, Appendix II contains only a subset of the schema relevant to the configurations you have deployed, including: global settings, devices, device groups, templates, banner, access lists, IP hosts, VLANs, BGP routing, and system settings. You can also find a copy of the schema in the WSL filesystem in the `~/schema/` directory.

Create the schema file in your project:

Copy the `~/schema/.schema.yaml` schema file to your `~/nac-iosxe/` working directory using the command below in the WSL terminal:

TERMINAL · WSL/Ubuntu cisco@wkst1:~$

```
cp ~/schema/.schema.yaml ~/nac-iosxe/.schema.yaml
```

Alternative: Manually Create the Schema File

If you need to, you can copy the schema content from Appendix II manually.

First, create the file using the `touch` command in your WSL Ubuntu terminal:

TERMINAL · WSL/Ubuntu cisco@wkst1:~$

```
cd ~/nac-iosxe
touch ~/nac-iosxe/.schema.yaml
```

Then use **VS Code** to copy-paste the schema content into the file.

1. Open VS Code and navigate to your `nac-iosxe` folder
2. Open the `.schema.yaml` file you just created
3. Copy the complete schema content from Appendix II
4. Paste it into the file and save

Copyright © Cisco

Your project structure should now include:

OUTPUT **Project layout (after copying schema)**

```
/home/cisco/nac-iosxe/
│
├── data/
│   ├── devices/access01.nac.yaml # Task02: access01 registration
│   ├── devices/access02.nac.yaml # Task02: access02 registration
│   ├── devices/border.nac.yaml   # Task02 + Task08 (optional): border + BGP
│   ├── devices/core.nac.yaml     # Task02 + Task05: core + Loopback0
│   ├── global.nac.yaml          # Task03 + Task06: banner + hostname
│   ├── groups/access.nac.yaml    # Task04 + Task07 (optional): ACL + VLAN template
│   ├── templates/bgp.nac.yaml           # Task08: BGP file template definition (optional)
│   ├── templates/logging.nac.yaml       # Task09: Logging alias CLI template (optional)
│   └── templates/vlan.nac.yaml          # Task07: VLAN model template (optional)
├── tftpl/
│   └── bgp.yaml.tftpl                # Task08: BGP template file (optional)
├── .env
├── .schema.yaml                       # ← New schema file
├── defaults.yaml                      # Generated by Terraform (default values)
├── main.tf
├── model.yaml                         # Generated by Terraform (merged config)
└── rules/                             # Created later in this task (semantic validation)
    └── no_permit_any.py              # Custom nac-validate rule
```

How defaults get applied - precedence order

`defaults.yaml` shows the *effective* defaults at merge time. Those values come from three possible sources, merged in this precedence order (later wins):

1. **Module built-in defaults** - shipped inside the Network as Code module itself. You don't see or edit them.
2. **Your own defaults file** - optional YAML you place in `data/` matching the `defaults:` schema root. Overrides module built-ins.
3. **Per-device / per-group / global values in your intent YAML** - explicit values in `config-*.nac.yaml` files. Override both default tiers.

The formal rule is documented at netascode.cisco.com/docs/guides/concepts/default_values/. Practical implication: if something in `model.yaml` isn't what you expected, `defaults.yaml` is the first place to look - a module built-in may be setting the value you thought was empty.

Install the nac-validate tool

First, install the **nac-validate** tool using pip in your **WSL Ubuntu terminal**:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
pip install nac-validate
```

Run schema validation

Navigate to your project directory:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
cd ~/nac-iosxe
```

Run validation:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
nac-validate -s .schema.yaml data/
```


Copyright © Cisco

Command breakdown:

- `nac-validate` - The validation tool
- `-s .schema.yaml` - Specifies the schema file to validate against
- `data/` - The directory containing your YAML configuration files

Successful validation

If your YAML files are correct, the command will return without any output - you'll just get your prompt back:

OUTPUT **Expected output (success)**

```
cisco@wkst1:~/nac-iosxe$ nac-validate -s .schema.yaml data/
cisco@wkst1:~/nac-iosxe$
```

No output means success. To confirm what `nac-validate` actually checked, re-run with verbose logging:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
nac-validate -s .schema.yaml -v DEBUG data/
```

OUTPUT **Expected output (verbose)**

```
cisco@wkst1:~/nac-iosxe$ nac-validate -s .schema.yaml -v DEBUG data/
DEBUG - Loading schema
DEBUG - Validate file: data/global.nac.yaml
DEBUG - Validate file: data/devices/core.nac.yaml
DEBUG - Validate file: data/devices/border.nac.yaml
DEBUG - Validate file: data/devices/access01.nac.yaml
DEBUG - Validate file: data/devices/access02.nac.yaml
DEBUG - Validate file: data/groups/access.nac.yaml
cisco@wkst1:~/nac-iosxe$
```

Each file in `data/` was loaded, parsed, and validated against the schema. This confirms that:

- Your YAML syntax is correct
- All attributes match the schema
- Data types are correct
- Values are valid (e.g., IP addresses are properly formatted)

Validation error examples

Let's intentionally introduce errors to see how validation catches them.

EXAMPLE 1: INVALID IP ADDRESS

If you accidentally typed an invalid IP like `198.51.100.1010` in `data/devices/core.nac.yaml` (the Loopback0 address from Task 05):

FILE **data/devices/core.nac.yaml (bad example)**

```
...
configuration:
  interfaces:
    loopbacks:
      - id: 0
        ipv4:
          address: 198.51.100.1010  # Invalid - octet > 255
          address_mask: 255.255.255.255
```


Copyright © Cisco

Running `nac-validate -s .schema.yaml data/` would produce:

OUTPUT **Validation error**

```
ERROR - Syntax error 'data/devices/core.nac.yaml':
iosxe.devices.[name=core].configuration.interfaces.loopbacks.[id=0].ipv4.address: '198.51.100.1010' is not a ip.
```

EXAMPLE 2: WRONG ATTRIBUTE NAME

If you misspelled an attribute like `banner` as `banners` in `data/global.nac.yaml`:

FILE **data/global.nac.yaml (bad example)**

```yaml
...
global:
  configuration:
    banners:  # Should be "banner" (singular)
      login: |
        ...
```

Running `nac-validate -s .schema.yaml data/` would produce:

OUTPUT **Validation error**

```
ERROR - Syntax error 'data/global.nac.yaml':
iosxe.global.configuration.banners: Unexpected element
```

EXAMPLE 3: INVALID ENUM VALUE

If you used an invalid action in an ACL in `data/groups/access.nac.yaml`:

FILE **data/groups/access.nac.yaml (bad example)**

```yaml
...
access_lists:
  standard:
    - name: AccessLayerACL
      entries:
        - sequence: 10
          action: allow  # Should be "permit" or "deny"
          prefix: 10.0.0.0
          prefix_mask: 0.0.0.255
        ...
```

Running `nac-validate -s .schema.yaml data/` would produce:

OUTPUT **Validation error**

```
ERROR - Syntax error 'data/groups/access.nac.yaml':
iosxe.device_groups.[name=ACCESS_SWITCHES].configuration.access_lists.standard.[name=AccessLayerACL].entries.[sequence=10].action: 'allow' not in ('deny',
'permit')
```

If everything is correct, you'll get your prompt back with no output. If there are errors, the tool will tell you exactly what's wrong and where.

IMPORTANT: Fix and Re-validate

Correct the introduced typos in your YAML files and run `nac-validate -s .schema.yaml data/` again to confirm that all files are now correct. Don't continue until you have fixed the typos.

Copyright © Cisco

Common validation errors and fixes

Error Message	Cause	Fix
`Required field missing`	Missing a mandatory attribute	Add the required field to your YAML
`Unexpected element`	Attribute name is misspelled or not supported	Check spelling against schema.yaml
`mapping values are not allowed here`	Potentially incorrect indentation	Correct the YAML syntax as per schema.yaml
`... not in enum(...)`	Using invalid value for a field	Use one of the allowed values from the error message
`... is not a regex match`	String does not match expected pattern	Ensure string format matches schema requirements
`... is not a int`	Value is not an integer	Change value to an integer
`... is not a list`	Expected a list but got something else	Use YAML list format (`[]` or `-` before each item)

Integrating validation into your workflow

Best practice workflow:

1. Edit your YAML files in VS Code
2. **Run** `nac-validate` to check for errors
3. Fix any errors reported
4. Run `terraform plan` to preview changes
5. Run `terraform apply` to deploy

By validating before running Terraform, you catch configuration errors immediately without attempting to connect to devices. This saves time and prevents partial deployments of invalid configurations.

> **CI/CD Integration**
>
> In Task13 - Run CI/CD Pipeline, you'll see how schema validation is automatically integrated into the GitLab CI/CD workflow. The pipeline runs `nac-validate` as part of the automated process, ensuring that every configuration change is validated before deployment - without manual intervention.

Semantic validation - write and run a custom rule

Schema validation (everything above) is **syntactic**: does the YAML match the shape of the data model? `nac-validate` can also do **semantic** validation - rules that answer "does this configuration make sense in context?".

Cisco CX ships a rule pack covering common semantic checks:

- Key attributes are unique (no duplicate device names, no overlapping ACL sequence numbers, etc.).
- IP routing is enabled when a routing protocol (BGP, OSPF, EIGRP, IS-IS) is configured.
- VLAN IDs referenced from an interface exist in the VLAN database.

You can also write your own. Rules are Python classes named `Rule` that implement a `match` classmethod receiving the merged data model (the `inventory` dict) and returning a list of error strings (empty list = pass). Let's write one and run it against your project.

Copyright © Cisco

STEP 1: CREATE THE `RULES/` DIRECTORY

In your **WSL Ubuntu terminal**, create a `rules/` folder under your project:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
mkdir -p ~/nac-iosxe/rules
```

`nac-validate` discovers rules by scanning a directory you point it at - any `.py` file containing a `Rule` class is loaded.

STEP 2: CREATE THE RULE FILE

Create the file in your WSL terminal:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
touch ~/nac-iosxe/rules/no_permit_any.py
```

Open `rules/no_permit_any.py` in **VS Code** and paste the following:

FILE **rules/no_permit_any.py**

```
class Rule:
    id = "101"
    description = "Standard ACLs must not have 'permit any' entries"
    severity = "HIGH"

    @classmethod
    def match(cls, inventory):
        results = []

        for scope_key in ("devices", "device_groups"):
            for item in inventory.get("iosxe", {}).get(scope_key, []):
                name = item.get("name", "unknown")
                acls = (
                    item.get("configuration", {})
                    .get("access_lists", {})
                    .get("standard", [])
                )
                for acl in acls:
                    for entry in acl.get("entries", []):
                        if entry.get("action") == "permit" and entry.get("any"):
                            results.append(
                                f"iosxe.{scope_key}[{name}].configuration"
                                f".access_lists.standard[{acl['name']}]"
                                f".entries[{entry['sequence']}] - "
                                f"'permit any' is a security risk"
                            )
        return results
```

What this rule does:

- `id`, `description`, `severity` - metadata `nac-validate` prints when the rule fails.
- `match()` - walks every device and device-group, looks at any standard ACL, and flags any entry whose `action` is `permit` AND whose `any` flag is `true`. Each flagged entry becomes one error string.

STEP 3: RUN THE RULE AGAINST YOUR CLEAN YAML

Back in your WSL terminal, run `nac-validate` with the `-r` flag pointing at `rules/`:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
nac-validate -s .schema.yaml -r rules/ data/
```

Your existing `AccessLayerACL` only permits specific subnets (`10.0.0.0/24` and `20.0.0.0/24` from Task 04) - no `permit any` anywhere - so the rule passes silently:

OUTPUT **Expected output (clean)**

Copyright © Cisco

```
cisco@wkst1:~/nac-iosxe$ nac-validate -s .schema.yaml -r rules/ data/
cisco@wkst1:~/nac-iosxe$
```

Empty output = both syntactic and semantic checks passed. Good - but a rule that never fails isn't very satisfying. Let's make it fire.

STEP 4: TRIGGER A VIOLATION

Open `data/groups/access.nac.yaml` in VS Code and add a third entry to `AccessLayerACL` that the rule will flag:

FILE **data/groups/access.nac.yaml**

```
access_lists:
  standard:
    - name: AccessLayerACL
      entries:
        # ... your existing sequence 10 and 20 entries above ...
        - sequence: 30
          action: permit
          any: true
```

Save the file and re-run validation:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
nac-validate -s .schema.yaml -r rules/ data/
```

This time the rule fires:

OUTPUT **Expected output (violation)**

```
cisco@wkst1:~/nac-iosxe$ nac-validate -s .schema.yaml -r rules/ data/
ERROR - Semantic error: 101 - HIGH - iosxe.device_groups[ACCESS_SWITCHES].configuration.access_lists.standard[AccessLayerACL].entries[30] - 'permit any' is a
security risk
```

Reading the error left to right:

- `101` and `HIGH` are the `id` and `severity` you set on the Rule class.
- The path `iosxe.device_groups[ACCESS_SWITCHES].configuration.access_lists.standard[AccessLayerACL].entries[30]` points exactly at the offending YAML node, the same way schema errors do.
- The trailing message is the string your rule's `match()` method returned.

STEP 5: REVERT THE VIOLATION

Remove the three lines you just added (the `sequence: 30` entry) so your YAML matches the lab's intent again, save the file, and confirm validation is clean:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
nac-validate -s .schema.yaml -r rules/ data/
```

OUTPUT **Expected output (clean again)**

```
cisco@wkst1:~/nac-iosxe$ nac-validate -s .schema.yaml -r rules/ data/
cisco@wkst1:~/nac-iosxe$
```


Copyright © Cisco

Why this pattern matters

Schema validation tells you "this YAML is shaped correctly". Semantic rules tell you "this YAML is shaped correctly *and* doesn't violate our network policy". The two layers stack: in a real CI pipeline you'd run both on every merge request, with the semantic rule pack curated to your org's standards (no `permit any`, naming conventions, AS-number ranges, mandatory description fields, etc.).

Where to find the full schema

The lab's `.schema.yaml` is a curated subset covering just the data-model paths exercised across Tasks 03-09. The full schema - with every supported IOS XE configuration option - is published at netascode.cisco.com/docs/data_models/iosxe/overview/.

Need the schema subset used in this lab?

See Appendix II for the full `.schema.yaml` you copied in earlier. It's a useful reference if you want to know what keys are valid without running into them via error messages.

Reference

For more details on the `nac-validate` tool, see the official documentation here.

What you've accomplished

In this task, you have:

- Understood the importance of pre-change validation
- Learned about schema-based validation
- Installed and used the nac-validate tool
- Validated your YAML configuration files
- Understood common validation errors and how to fix them
- Written a custom semantic rule and seen it both pass and flag a real violation
- Integrated validation into your Network as Code workflow

You now have a safety check in place to catch configuration errors before they reach your network devices. This is a critical DevOps practice that prevents misconfigurations and improves reliability.

What's next

Task 11 is **optional** - it automates post-change verification with Robot Framework tests rendered from your intent. If you do Task 11, run it **before** Task 12 (Task 12's `terraform destroy` wipes the configs Task 11 checks). If you're skipping the post-checks demo, go straight to Task 12 - Cleanup.

Task 06 - Variables

Task 11 - Post-checks

Copyright © Cisco

Task 11 - Post-checks with `nac-test` (Optional)

Estimated Time to Complete: ~15 minutes

> **Before you start**
>
> This task depends on configs applied in Tasks 03-06 (banner, ACL, Loopback0) plus `.schema.yaml` from Task 10 - Schema Validation and a recent `terraform apply` so `model.yaml` reflects your current intent. **Do not run this task after Task 12** - `terraform destroy` wipes every config Task 11 expects to verify, and the tests will return empty/failing results through no fault of your setup.

After Terraform applies a change, how do you confirm the configuration actually landed correctly on the devices? For one device, `show running-config` works. For a fleet, manual spot-checks don't scale - and running-config can't even tell you about operational state (BGP neighbor established, interface line-protocol up, VLAN database populated, etc.).

`nac-test` automates post-change validation by rendering Robot Framework tests directly from your intent YAML, then running them against the live devices. The test cases describe *what you asked for*; the framework confirms *what actually happened*.

- **Robot Framework** - a keyword-driven test framework; test cases are readable and easy to extend.
- **Pabot** - a parallel executor for Robot that runs suites simultaneously. Fast enough to run against many devices during a CI pipeline.

The key insight: you don't write tests by hand. `nac-test` renders them from the merged `model.yaml` NaC already produces - so the tests always match the intent.

> **Short on time? The most valuable part of this task is Step 7**
>
> Steps 1-6 walk through the happy path: render the tests, run them, see them pass. Step 7 is where `nac-test` earns its keep - you'll deliberately desync intent from device state and watch the tests catch it. If you only have 10 minutes, skim Steps 1-4, skip the report walkthrough in Steps 5-6, and make sure you do Step 7.

What you'll learn

By the end of this task you will have:

- Installed `nac-test` and copied the pre-built `tests/` scaffolding into your project
- Rendered a Robot Framework test suite directly from your intent YAML and `model.yaml`
- Read the generated HTML + JUnit test reports

Use case: validating the ACL on access switches

In Task 4 you deployed `AccessLayerACL` to `access01` and `access02` via a device group. You'll validate that it landed correctly.

> **Lab scope vs production**
>
> Production Network as Code deployments ship 100+ Robot test templates covering every configuration type. This task demonstrates the workflow with one (`access_lists.robot`) - once you see the pattern, the rest is more of the same.

Copyright © Cisco

The deployed intent (from `data/groups/access.nac.yaml`):

FILE **data/groups/access.nac.yaml**

```yaml
---
iosxe:
  device_groups:
    - name: ACCESS_SWITCHES
      devices:
        - access01
        - access02
      configuration:
        access_lists:
          standard:
            - name: AccessLayerACL
              entries:
                - sequence: 10
                  action: permit
                  prefix: 10.0.0.0
                  prefix_mask: 0.0.0.255
                - sequence: 20
                  action: permit
                  prefix: 20.0.0.0
                  prefix_mask: 0.0.0.255
```

Step 1: Copy the test files into your project

The lab WSL image ships with a pre-built `tests/` directory under `~/tests/`. Copy it into your project:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
cp -r ~/tests ~/nac-iosxe/
```

Your project now has:

OUTPUT **Project layout (after copying tests/)**

```
~/nac-iosxe/
├── data/                              # (your YAML intent from earlier tasks)
├── tests/
│   ├── filters/
│   │   └── url_encode.py              # Jinja2 filter - encodes special characters
│   └── templates/
│       ├── iosxe_common.resource      # Robot keywords shared across suites
│       ├── lib/
│       │   └── UtilsLib.py            # NETCONF connection + XML assertion keywords
│       └── config/
│           └── access_lists.robot     # Jinja2 template → rendered Robot suite
├── model.yaml                         # (generated by `terraform apply`)
└── ...
```

The contents of `access_lists.robot` are shown in Appendix III for reference.

Step 2: Have the model file ready

You will use the `model.yaml` file generated in the previous tasks as input to `nac-test`.

Copyright © Cisco

If the `model.yaml` file is missing

Before running `nac-test`, you need to run Terraform to generate the merged model file (`model.yaml`) that `nac-test` uses for rendering tests. If you haven't run the terraform commands below yet, do so now in your **WSL Ubuntu terminal**:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
cd ~/nac-iosxe
```

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform plan
```

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform apply
```

When prompted, type `yes` to confirm the deployment.

This generates two files in your project directory:

- `model.yaml` - The merged YAML data model (all your configuration files combined)
- `defaults.yaml` - The default values used by the module

Step 3: Install nac-test

Install the **nac-test** tool using pip in your **WSL Ubuntu terminal**:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
pip install nac-test
```

Step 4: Run nac-test

Once you have the model files and test templates in place, run the `nac-test` command in your **WSL Ubuntu terminal**:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
nac-test \
  --data ./model.yaml \
  --data ./defaults.yaml \
  --templates ./tests/templates \
  --filters ./tests/filters \
  --output /mnt/c/Users/admin/Desktop/TestResults
```

What this command does:

1. **Loads data** - Reads the merged model and defaults generated by Terraform
2. **Loads filters** - Loads custom Jinja2 filters from `./tests/filters`
3. **Renders templates** - Each template in `./tests/templates` is rendered with your configuration data
4. **Executes tests** - Pabot runs all test suites in parallel
5. **Creates reports** - Generates reports in the specified output directory

Copyright © Cisco

Output location

The test results are saved to your Windows Desktop (`C:\Users\admin\Desktop\TestResults`) for easy access. You can open the HTML reports directly in your browser.

At home? The `/mnt/c/Users/admin/...` path is lab-specific

The `--output /mnt/c/Users/admin/Desktop/TestResults` path works because the dCloud Win10 VM's user is literally `admin` , WSL mounts `C:` under `/mnt/c` , and `Desktop` is in a fixed location. If you re-run this command at home, **substitute the path**:

- **On your own WSL setup** – replace `admin` with your Windows username: `/mnt/c/Users/<YOU>/Desktop/TestResults` . Run `cmd.exe /c echo %USERNAME%` from the WSL prompt if you're not sure.
- **On native Linux / macOS** – pick any regular Linux path, e.g. `~/nac-test-results` or `./tests/results` (the pattern Task 14 uses when wiring `nac-test` into the GitLab pipeline).

The tool itself doesn't care – `--output` just needs a writable directory.

Step 5: Review the generated Robot test

After running `nac-test` , check the generated test file:

OUTPUT **TestResults directory**

```
C:\Users\admin\Desktop\TestResults\
└── config/
    └── access_lists.robot
```

This is the **TestResults** folder on the Windows 10 desktop. You can open the file with VS Code, notepad, or view the generated test file using your **WSL Ubuntu terminal**:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
cat /mnt/c/Users/admin/Desktop/TestResults/config/access_lists.robot
```

The `access_lists.robot` file contains tests automatically generated from your intent configuration:

OUTPUT **config/access_lists.robot (generated)**

```
*** Settings ***
Documentation   Verify Access Lists Configuration
Suite Setup     Run Only Once   Get Configs
Resource        ../iosxe_common.resource
Default Tags    config   iosxe   access_lists

*** Test Cases ***

Get Config access01
    ${config}=   Get Parallel Value For Key   access01_config
    Set Suite Variable   ${config}

Verify Standard Access List AccessLayerACL Device access01
    ${acl}=   Set Variable   data/native/ip/access-list/standard[name='AccessLayerACL']
    Should Be Equal Value Xml String   ${config}   ${acl}/name   AccessLayerACL
    ${entry}=   Set Variable   ${acl}/access-list-seq-rule[sequence='10']
    Should Be Equal Value Xml String   ${config}   ${entry}/remark
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/ipv4-address-prefix   10.0.0.0
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/mask   0.0.0.255
    Should Be Equal Value Xml Bool   ${config}   ${entry}/permit/std-ace/any
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/host-address
    Should Be Equal Value Xml Bool   ${config}   ${entry}/permit/std-ace/log
    ${entry}=   Set Variable   ${acl}/access-list-seq-rule[sequence='20']
    Should Be Equal Value Xml String   ${config}   ${entry}/remark
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/ipv4-address-prefix   20.0.0.0
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/mask   0.0.0.255
    Should Be Equal Value Xml Bool   ${config}   ${entry}/permit/std-ace/any
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/host-address
```


Copyright © Cisco

```
    Should Be Equal Value Xml Bool   ${config}   ${entry}/permit/std-ace/log

Get Config access02
    ${config}=   Get Parallel Value For Key   access02_config
    Set Suite Variable   ${config}

Verify Standard Access List AccessLayerACL Device access02
    ...
```

Lines with no expected value

Assertions where the expected value is blank (e.g. `remark`, `host-address`) are no-ops - the keyword returns immediately when the value is empty. They appear because the Jinja2 template renders every possible field; only the fields you defined in YAML actually perform a check.

Step 6: Review the test results

The terminal output from `nac-test`, that you ran earlier in Step 4, shows the test execution:

OUTPUT **nac-test execution output**

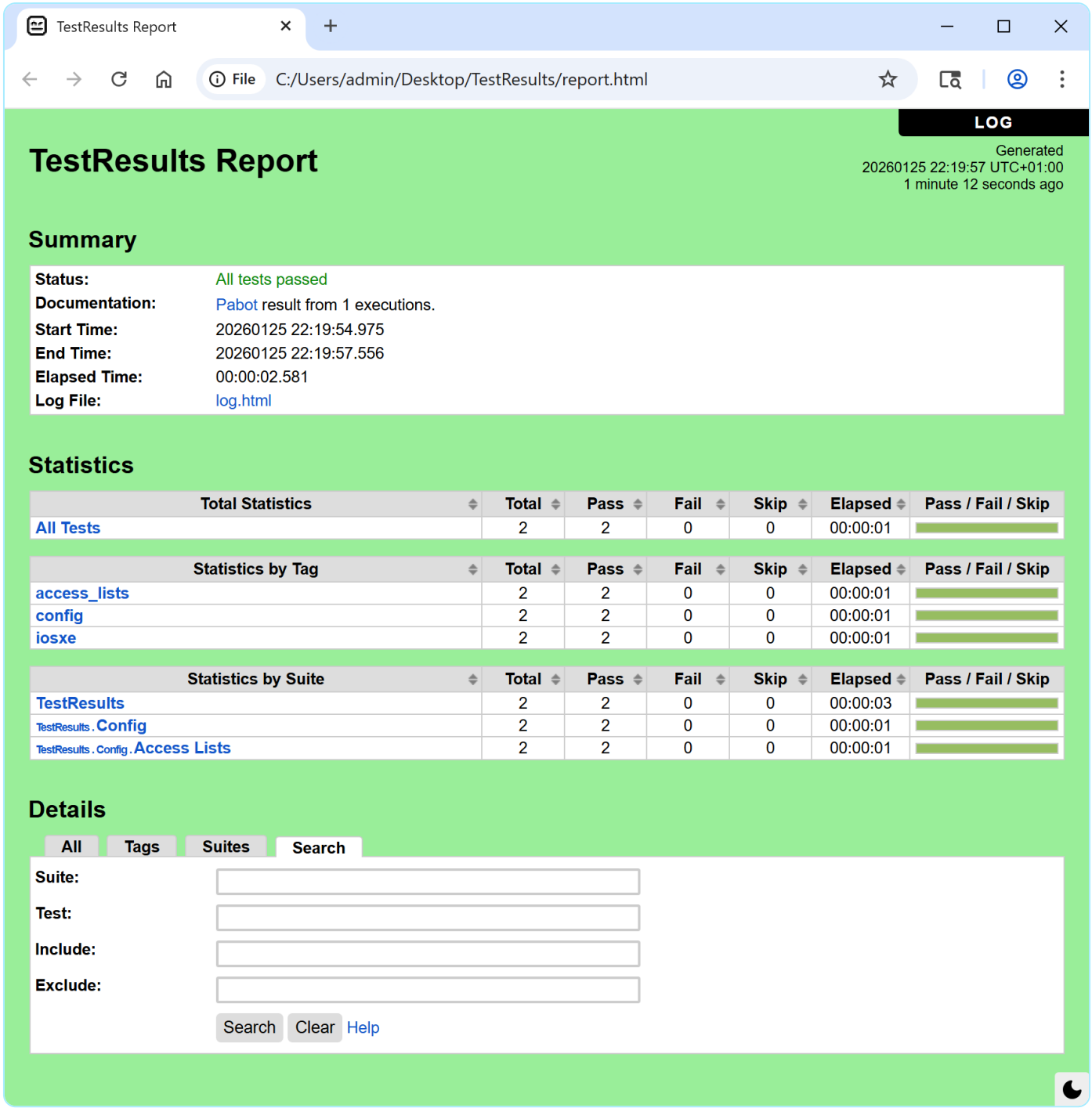
```
cisco@wkst1:~/nac-iosxe$ nac-test \
  --data ./model.yaml \
  --data ./defaults.yaml \
  --templates ./tests/templates \
  --filters ./tests/filters \
  --output /mnt/c/Users/admin/Desktop/TestResults
Robot Framework remote server at 127.0.0.1:36883 started.
Storing .pabotsuitenames file
2025-12-31 17:10:34.305712 [PID:25936] [0] [ID:0] EXECUTING TestResults.Config.Access Lists
2025-12-31 17:10:36.720046 [PID:25936] [0] [ID:0] PASSED TestResults.Config.Access Lists in 2.4 seconds
{'command': ['robot'], 'verbose': False, 'help': False, 'version': False, 'testlevelsplit': False, 'pabotlib': True, 'pabotlibhost': '127.0.0.1',
'pabotlibport': 0, 'processes': 4, 'processtimeout': None, 'artifacts': ['png'], 'artifactstimestamps': True, 'artifactsinsubfolders': False, 'shardindex':
0, 'shardcount': 1, 'chunk': False, 'no-rebot': False, 'argumentfiles': []}
2 tests, 2 passed, 0 failed, 0 skipped.
==================================================
Output:  /mnt/c/Users/admin/Desktop/TestResults/output.xml
XUnit:   /mnt/c/Users/admin/Desktop/TestResults/xunit.xml
Log:     /mnt/c/Users/admin/Desktop/TestResults/log.html
Report:  /mnt/c/Users/admin/Desktop/TestResults/report.html
Stopping PabotLib process
Robot Framework remote server at 127.0.0.1:36883 stopped.
PabotLib process stopped
Total testing: 2.40 seconds
Elapsed time:  3.41 seconds
cisco@wkst1:~/nac-iosxe$
```

Output artifacts:

- `output.xml` - Test results in Robot Framework format
- `xunit.xml` - Results in xUnit format for CI/CD integration
- `log.html` - Detailed execution log
- `report.html` - Human-readable summary report

Open the report in a browser to see the visual results. Navigate to your Desktop and open the `TestResults` folder, then double-click `report.html`:

Copyright © Cisco



Log Details

The `log.html` file contains detailed step-by-step execution information, including request and response data for each test case. This is useful for troubleshooting any test failures or understanding the test flow in depth.

Step 7: See a test fail (then fix it)

The previous run passed because the device state matched the intent. To see the real value of post-checks, you'll create a situation where they DON'T match - simulating **configuration drift**.

Copyright © Cisco

What is configuration drift?

Drift happens when a device's running config diverges from the declared intent. Common causes: a manual CLI change during troubleshooting, a device reboot that doesn't fully restore config, a NETCONF commit that silently drops a leaf, or another automation tool overwriting what Terraform pushed. Post-checks catch all of these.

Simulate drift by manually removing an ACL entry from `access01`:

Open Solar-PuTTY and connect to **access01** (`198.18.130.11`). Then remove sequence 10 from the ACL:

DEVICE CLI · PuTTY **access01**

```
configure terminal
ip access-list standard AccessLayerACL
no 10
end
```

The intent (`model.yaml`) still says sequence 10 should exist, but `access01` no longer has it. This is drift.

Run nac-test again:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
nac-test \
  --data ./model.yaml \
  --data ./defaults.yaml \
  --templates ./tests/templates \
  --filters ./tests/filters \
  --output /mnt/c/Users/admin/Desktop/TestResults
```

This time, the tests **fail**:

OUTPUT **nac-test output (drift detected)**

```
...
2 tests, 1 passed, 1 failed, 0 skipped.
=================================================
```

The test for `access01` expected sequence 10 (`prefix: 10.0.0.0`) to exist because it's in the intent - but the device no longer has it. The test for `access02` still passes because that device wasn't touched. This is exactly what nac-test catches: a mismatch between what you WANT on the device and what's ACTUALLY there.

Now fix the drift with Terraform and re-run:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform apply -auto-approve
```

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
nac-test \
  --data ./model.yaml \
  --data ./defaults.yaml \
  --templates ./tests/templates \
  --filters ./tests/filters \
  --output /mnt/c/Users/admin/Desktop/TestResults
```

OUTPUT **nac-test output (drift fixed)**

```
...
2 tests, 2 passed, 0 failed, 0 skipped.
=================================================
```

Tests pass again - `terraform apply` detected the missing ACE and re-pushed it to `access01` .

Copyright © Cisco

The real-world takeaway

In production, drift happens silently. A network engineer fixes a P1 outage with a manual CLI change, a device reboots and loses an uncommitted config, or a NETCONF commit ACKs but silently ignores a leaf. `nac-test` catches all of these by comparing what's actually on the wire against your source-of-truth intent. Run it after every `terraform apply`, and schedule it periodically to catch out-of-band drift.

What you've accomplished

- Copied the pre-built `tests/` directory into your project
- Ran `nac-test` to render and execute Robot Framework tests from your intent YAML
- Reviewed the auto-generated `access_lists.robot` suite
- Interpreted the HTML report and JUnit results
- Observed a test failure when device state didn't match intent, then resolved it

CI/CD integration

In Task 14 you'll wire `nac-test` into the GitLab pipeline so every deployment auto-verifies itself - no manual step.

Task 10 - Schema validation

Task 12 - Cleanup

Copyright © Cisco

Task 12 - Cleanup

Estimated Time to Complete: ~5 minutes

Before moving to CI/CD automation, clean up the manual work you've done so far. This ensures a clean starting point for the GitLab-based workflow.

What you'll learn

By the end of this task you will have:

- Run `terraform destroy` to remove all Terraform-managed configuration from the lab devices
- Understood **why** state separation matters - two Terraform environments fighting over the same devices produces unpredictable results
- Prepared a clean environment ready for the GitLab-driven workflow starting in Task 13

> **Important Note**
>
> Once you run the cleanup, you will no longer be able to complete the optional Tasks 07-09 (Templates), or Task 11 (Post-check Tests). The cleanup removes all previous configuration from your devices to prepare for the CI/CD tasks.
>
> Please take a moment and check your remaining time to decide whether to proceed to the CI/CD section or spend more time on the optional tasks.

Why clean up?

In the previous tasks, you:

- Created configuration files manually in VS Code
- Ran Terraform commands directly from WSL Ubuntu
- Deployed configurations to the IOS XE devices

Starting from the next task, you'll use **GitLab CI/CD pipelines** to automate all of this. The GitLab repository already contains a complete project setup, so you need to:

1. **Remove configurations from devices** - Undo the changes made during manual tasks
2. **(Optionally) Delete local files** - You may clean up the manually created project folder as it's no longer needed

Step 1: Destroy Terraform resources

First, remove all configurations that Terraform deployed to the IOS XE devices. Open your WSL Ubuntu terminal.

Navigate to the project folder:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
cd ~/nac-iosxe
```

Run Terraform destroy:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
terraform destroy
```


Copyright © Cisco

When prompted, type `yes` to confirm. Terraform will:

- Connect to each IOS XE device
- Remove all configurations it previously applied (banners, ACLs, VLANs, etc.)
- Update the state file to reflect the clean state

The destroy process may take a few minutes

Wait until you see "Destroy complete!" before proceeding.

Terraform destroy Warning

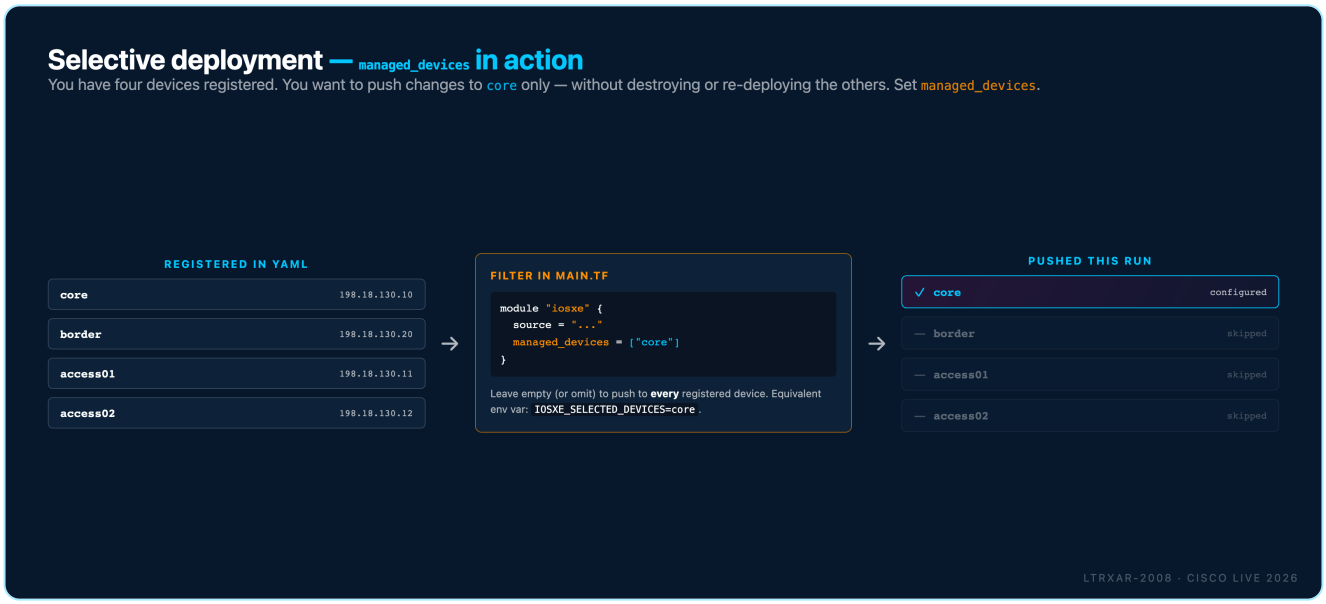
The `terraform destroy` command removes all configurations that were applied by Terraform. Use it with caution! In a production environment, do not run this command unless you intend to completely remove the deployed infrastructure.

If you made any manual changes directly on the devices outside of Terraform, those changes will remain.

Copyright © Cisco

Need to clean up just *one* device? Use `managed_devices`

`terraform destroy` unconfigures every device the module manages. If you only want to touch a subset, the Network as Code module exposes a filter variable - `managed_devices` - that restricts Terraform's scope to a specific list of device names. Every other registered device is left alone.



To use it, edit `main.tf`:

FILE **main.tf**

```
module "iosxe" {
  source              = "git::https://github.com/netascode/terraform-iosxe-nac-iosxe.git"
  yaml_directories    = ["data/"]
  managed_devices     = ["core"]   # only core is in scope this run
}
```

The equivalent environment variable, useful for CI jobs without editing `main.tf`, is `IOSXE_SELECTED_DEVICES=core` (comma-separated for multiple).

`managed_devices = []` (the default, or omitting the variable) means "all registered devices" - which is the behavior you're using in this lab.

Step 2: Verify Network as Code configurations are removed (optional)

You can use **Solar-PuTTY** to connect to one of the devices and verify the configurations have been removed. As you did in Task 01, double-click on the **core** switch to connect.

Once connected, check that the banner, hostname and other configurations you applied are no longer present. You can also check the running configuration.

DEVICE CLI · PuTTY **core**

```
show running-config
```


Copyright © Cisco

Default Hostnames

If you completed Task 06 - Variables, the hostnames revert to default (for example, `Switch` or `Router`). Running `terraform destroy` removes all changes made during the lab, reverting to defaults even if they were pre-configured manually before Terraform. Don't worry, the hostnames will be re-applied in the next task.

Step 3: Delete the local project folder (optional)

If you wish, you may also remove the manually created project folder, `nac-iosxe`, as it's no longer needed.

Remove Project Folder (Optional)

To delete the project folder, run the following commands in your WSL Ubuntu terminal:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
cd ~
```

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
rm -rf ~/nac-iosxe
```

This deletes:

- `.env` - Credentials file
- `main.tf` - Terraform configuration
- `data/` - All YAML configuration files
- `tftpl/` - Template files (.tftpl)
- `tests/` - Robot Framework test files
- `.terraform/` - Downloaded modules and providers
- `terraform.tfstate` - State file

You can verify the folder is deleted by listing the home directory contents:

TERMINAL · WSL/Ubuntu **cisco@wkst1:~$**

```
ls -la ~/
```

You should no longer see the `nac-iosxe` folder listed.

Step 4: Close VS Code and WSL terminal

In the next tasks, you will no longer need VS Code and the WSL Ubuntu terminal. You can close both applications now.

What you've accomplished

- Removed all Terraform-deployed configurations from IOS XE devices
- (Optionally) deleted the manually created project folder
- Prepared a clean environment for CI/CD automation

Copyright © Cisco

Next steps

In the next task, you'll work with **GitLab** to:

- Use an existing project with pre-configured CI/CD pipelines
- Make changes through the GitLab Web IDE
- Let the pipeline automatically validate, plan, and deploy configurations

While manual Terraform commands are useful for learning and small-scale changes, in production environments you will typically use automated CI/CD pipelines to implement the Network as Code workflow.

Task 10 - Schema validation

Task 13 - Run a CI/CD pipeline

Copyright © Cisco

Task 13 - Run a CI/CD pipeline

Estimated Time to Complete: ~20 minutes

> ⚠ **You must finish Task 12 before starting Task 13**
>
> The GitLab pipeline you're about to trigger manages the same lab devices Terraform has been managing locally. If both environments hold state for those devices at the same time, every pipeline run fights your local `terraform apply` (and vice versa), producing symptoms like "I deployed a banner but it reverted 30 seconds later." `terraform destroy` in Task 12 removes the local state so GitLab becomes the single source of truth from here on.
>
> **Self-check** - run this in your WSL terminal from `~/nac-iosxe/` :
>
> TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**
>
> ```
> terraform state list
> ```
>
> - **No output** - perfect, local state is empty, proceed.
> - **Any output at all** - Task 12 hasn't been run (or didn't finish). Run `terraform destroy` and answer `yes` before you continue.

Up until now you've been driving Terraform manually: `init` , `plan` , `apply` . That's fine for learning, but production teams run the same workflow through CI/CD so every change is validated, previewed, applied, and tested the same way - without a human typing commands. In this task you'll do exactly that using a pre-configured **GitLab pipeline**.

What you'll learn

By the end of this task you will have:

- Navigated the `netascode/nac-iosxe-terraform` GitLab project and read its `.gitlab-ci.yml`
- Committed a change via the GitLab Web IDE and watched the pipeline trigger automatically
- Observed a complete pipeline run (validate → plan → deploy → notify) and verified the deployed change on the lab devices

CI/CD for Network as Code

The lab ships with a pre-configured GitLab repository containing your Network as Code project and a ready-to-run CI/CD pipeline. Every push or merge request triggers a pipeline that runs the same five stages you just performed by hand - plus a few you didn't:

Copyright © Cisco



Two important behaviors to notice before we run one:

- **Merge-request pipelines only preview** – they run `validate` + `plan` and post the plan diff as an MR comment. They never touch the devices. This is the guardrail that makes code review meaningful.
- **Main-branch pipelines actually deploy** – they run all five stages (`validate` → `plan` → `deploy` → `test` → `notify`). Merging to `main` is the act that commits a change to the real network.

Same pipeline definition, different behavior per branch – a simple pattern that gives you a preview environment for free.

Step 1: Access GitLab

> ⚠ **Use the browser inside your RDP session**
>
> GitLab is only reachable from within the lab environment. Open the URL below in **Chrome inside your RDP session** (the Windows 10 VM) – it will not load from your local machine.

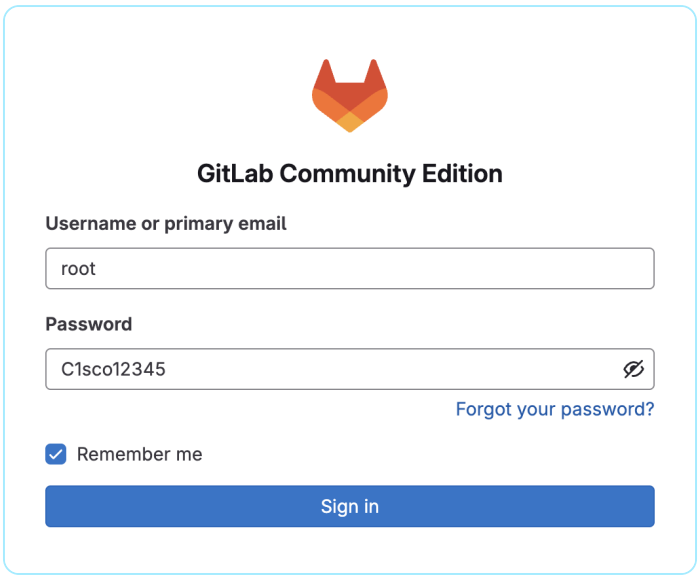
In **Chrome**, navigate to GitLab – open the following link in a new tab:

https://198.18.133.101

> ⚠ **Certificate Warning**
>
> You may see a security warning because the lab uses a self-signed certificate. Click **Advanced** and then **Proceed to 198.18.133.101** to continue.

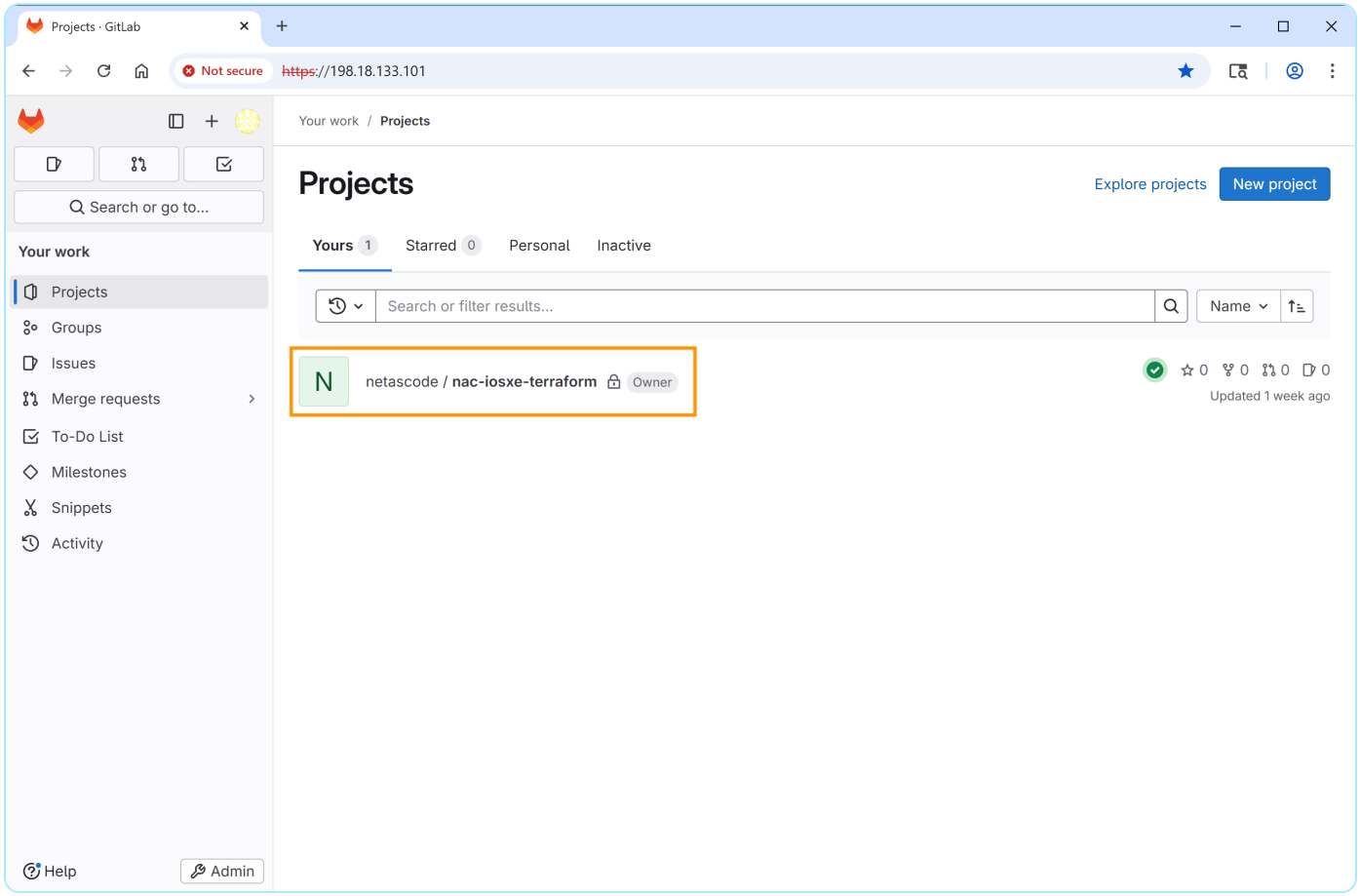
If prompted, log in with credentials: **Username:** `root` / **Password:** `C1sco12345`

Copyright © Cisco



Step 2: Navigate to the NaC-IOSXE project

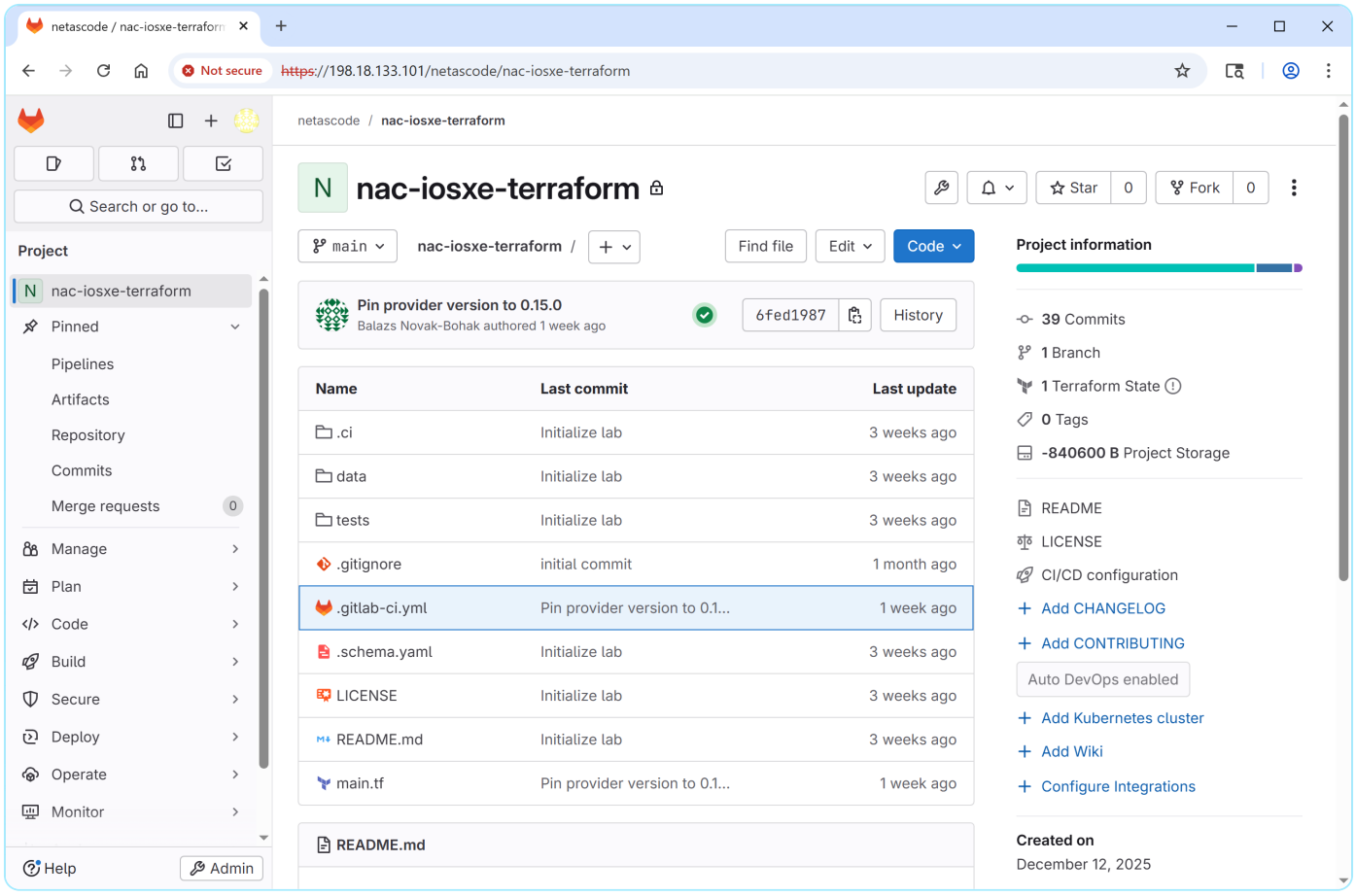
After logging in, you'll see the GitLab dashboard. Click on the **netascode/nac-iosxe-terraform** project to open it.



Copyright © Cisco

The project page shows your repository files, including:

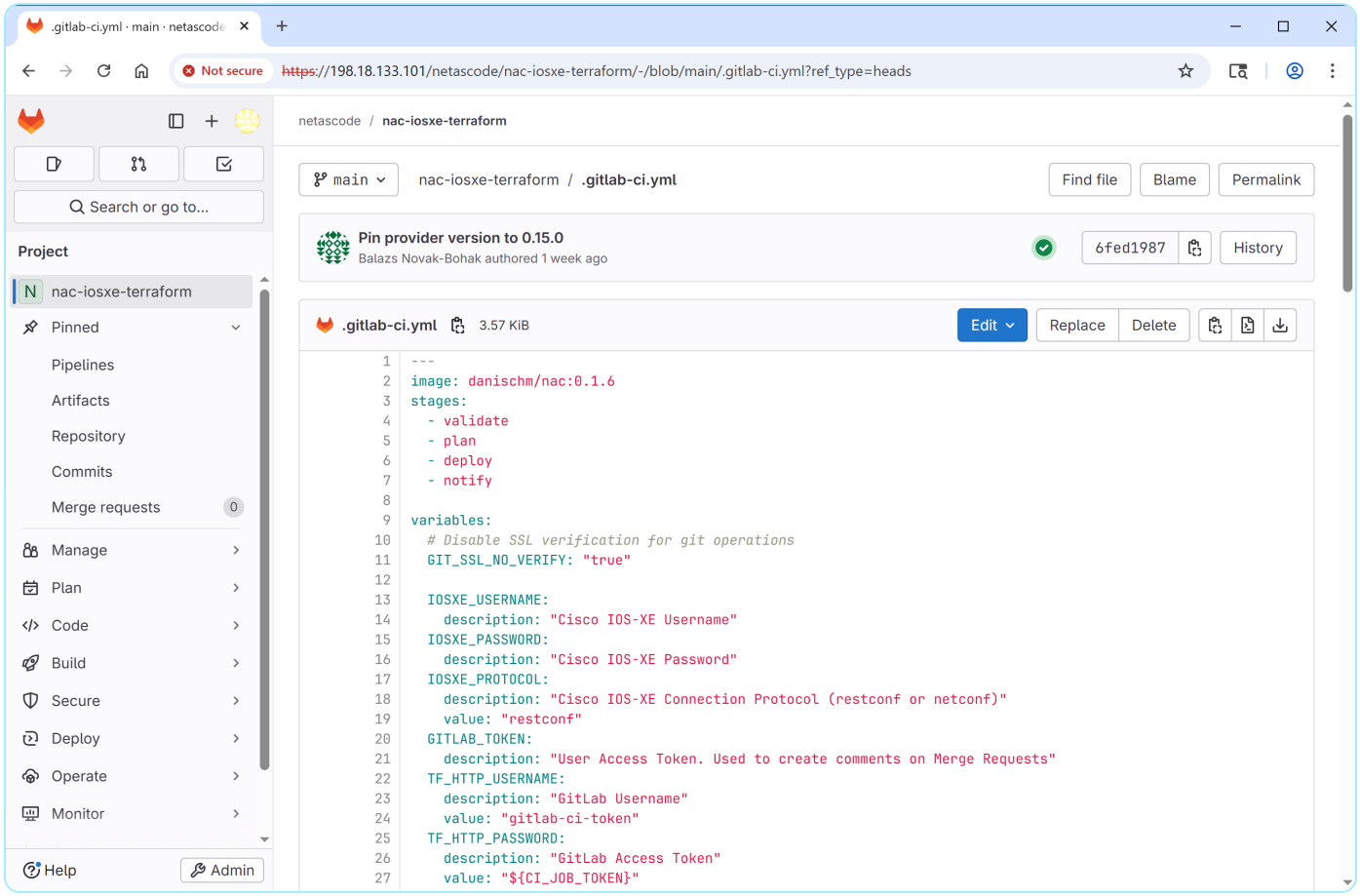
- `data/` folder with your YAML configurations - Same as you created in Task 02
- `tests/` folder with your ROBOT tests - Same as the ACL tests from optional Task 11
- `main.tf` - Terraform configuration - Same as you created in Task 02
- `.schema.yaml` - The Network as Code for IOS XE schema - Same as you used in Task 10
- `.gitlab-ci.yml` - CI/CD pipeline definition file - This is new!



CI/CD pipeline configuration

Before running the pipeline, understand how it's configured first. Click on `.gitlab-ci.yml` to view the pipeline definition:

Copyright © Cisco



A CI/CD pipeline is an automated workflow of tasks that enables changes to be deployed in a consistent and repeatable manner. This lab leverages pipelines to automate the tasks you previously performed manually. This pipeline example includes the following stages: `validate`, `plan`, `deploy`, and `notify`.

FILE **.gitlab-ci.yml Snippets**

```
image: danischm/nac:0.1.6
stages:
  - validate
  - plan
  - deploy
  - notify

variables:
  IOSXE_USERNAME:
    description: "Cisco IOS XE Username"
  IOSXE_PASSWORD:
    description: "Cisco IOS XE Password"
  # ... additional variables for GitLab tokens, Terraform state, Webex notifications ...

cache:
  key: terraform_modules_and_lock
  paths:
    - .terraform
    - .terraform.lock.hcl
    - defaults.yaml
    - model.yaml

validate:
  stage: validate
  script:
    - set -o pipefail && terraform fmt -check |& tee fmt_output.txt
    - set -o pipefail && nac-validate ./data/ |& tee validate_output.txt
  # ... artifacts and rules configuration ...

plan:
  stage: plan
  script:
    - terraform init -upgrade -input=false
    - terraform plan -out=plan.tfplan -input=false
```


Copyright © Cisco

```
    - terraform show -no-color plan.tfplan > plan.txt
    # ... additional plan output processing ...
  # ... artifacts, dependencies, and branch rules ...

deploy:
  stage: deploy
  script:
    - terraform init -input=false
    - terraform apply -input=false -auto-approve plan.tfplan
  # ... runs only on main branch ...

failure:
  stage: notify
  script:
    - python3 .ci/webex-notification-gitlab.py -f
  when: on_failure

success:
  stage: notify
  script:
    - python3 .ci/webex-notification-gitlab.py -s
  when: on_success
```

Abbreviated View

The YAML above shows the key structure of `.gitlab-ci.yml`. The actual file (~150 lines) includes additional configuration for variables, artifacts, caching, and branch rules. You can view the complete file in the GitLab repository.

Key concepts:

- **image** - Uses a pre-built Docker container with Terraform, `nac-validate`, and other tools
- **stages** - Define the order of execution (validate → plan → deploy → notify)
- **variables** - Pipeline variables for credentials (IOS XE, GitLab), entered at runtime
- **cache** - Preserves Terraform modules and state between pipeline runs
- **validate** - Runs `terraform fmt` check and `nac-validate` schema validation
- **plan** - Creates the Terraform execution plan and generates reports
- **deploy** - Applies the configuration automatically (runs on main branch only)
- **failure/success** - Send Webex notifications based on pipeline outcome

Webex Notifications

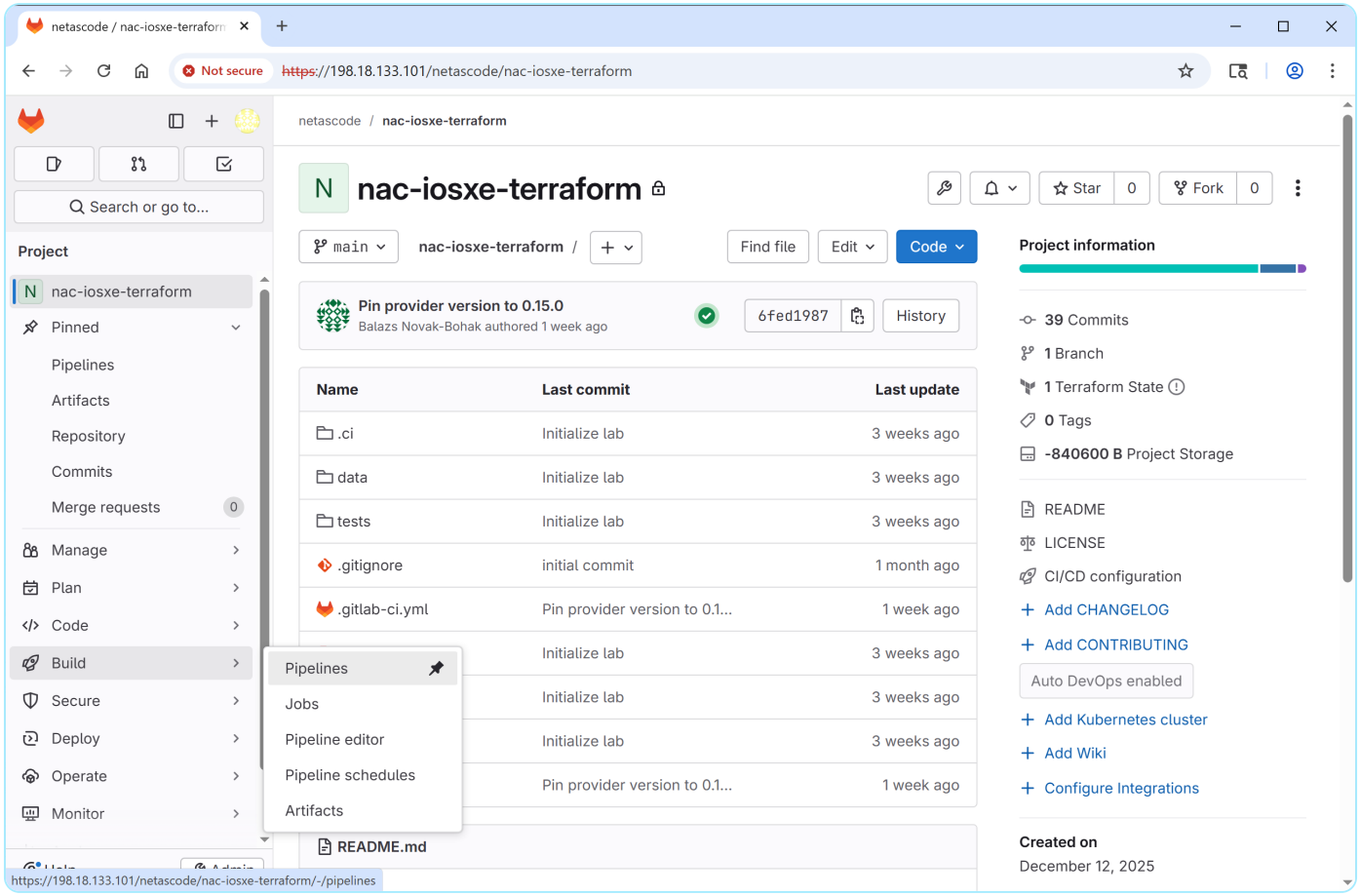
The **notify** job typically use a Python script to send messages to a Webex room. In this lab, this functionality is intentionally omitted, the notify stage is only included as a placeholder for demonstration purposes.

If you're interested in implementing notifications, reach out to your instructors for guidance.

View existing pipelines

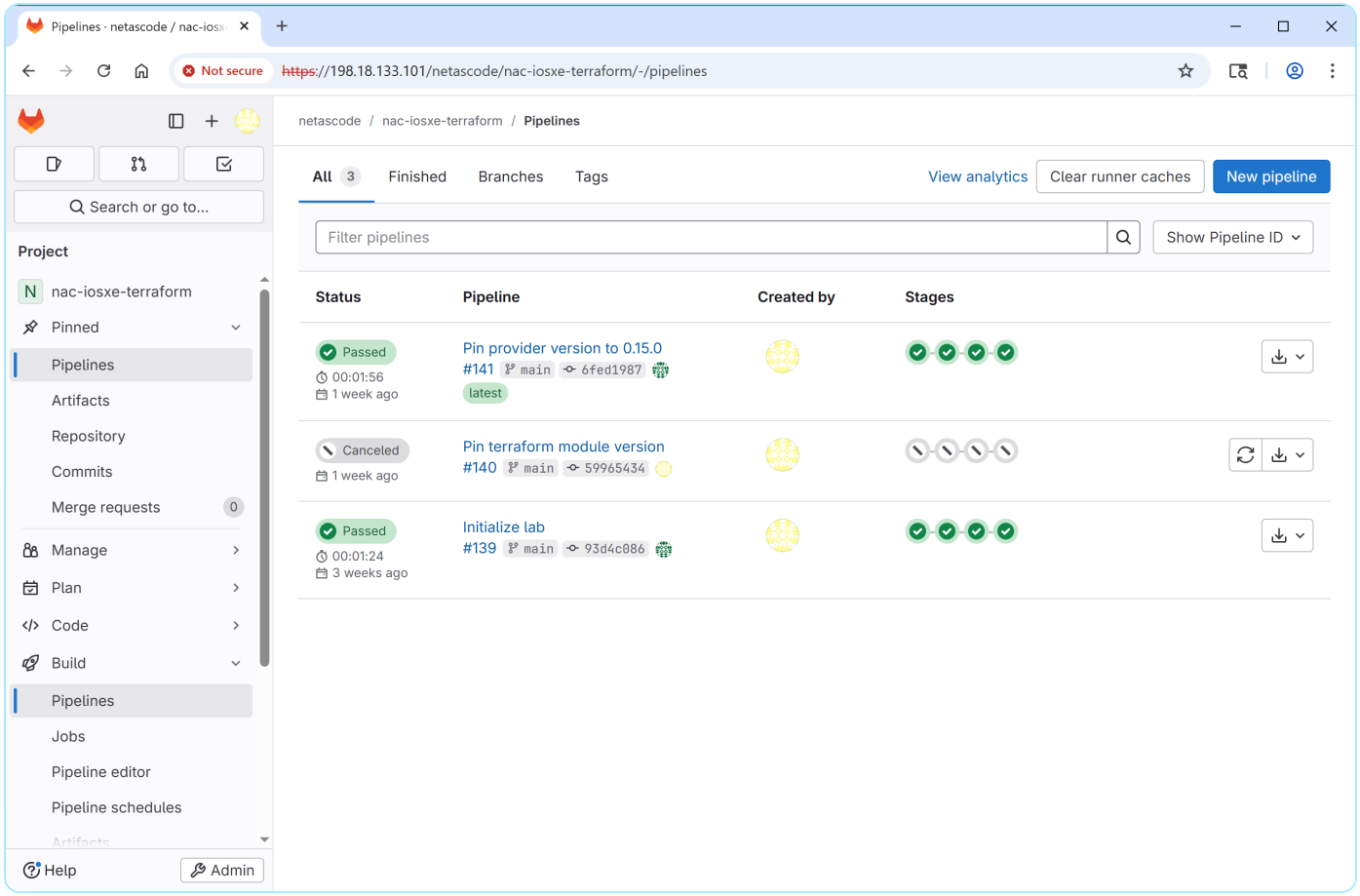
To see the pipeline history, navigate to **Build** → **Pipelines** in the left sidebar.

Copyright © Cisco



You'll see a list of past pipeline runs with their status (passed, failed, running).

Copyright © Cisco



State file management

Navigate to Home (GitLab icon in the top left), select **netascode/nac-iosxe-terraform**, then open the `main.tf` file.

The pipeline uses GitLab's **http backend** to store the Terraform state file on the GitLab server. This allows multiple pipeline runs to share the same state, ensuring consistency across deployments.

It is configured in the `main.tf` file with the following `backend` block under the `terraform` section:

FILE **main.tf (excerpt)**

```
backend "http" {
  skip_cert_verification = true
}
```

Certificate Verification

The `skip_cert_verification = true` setting is used here because the lab environment uses self-signed certificates. In production, you should use valid certificates.

For more details on how this is set up, refer to the GitLab Docs.

Copyright © Cisco

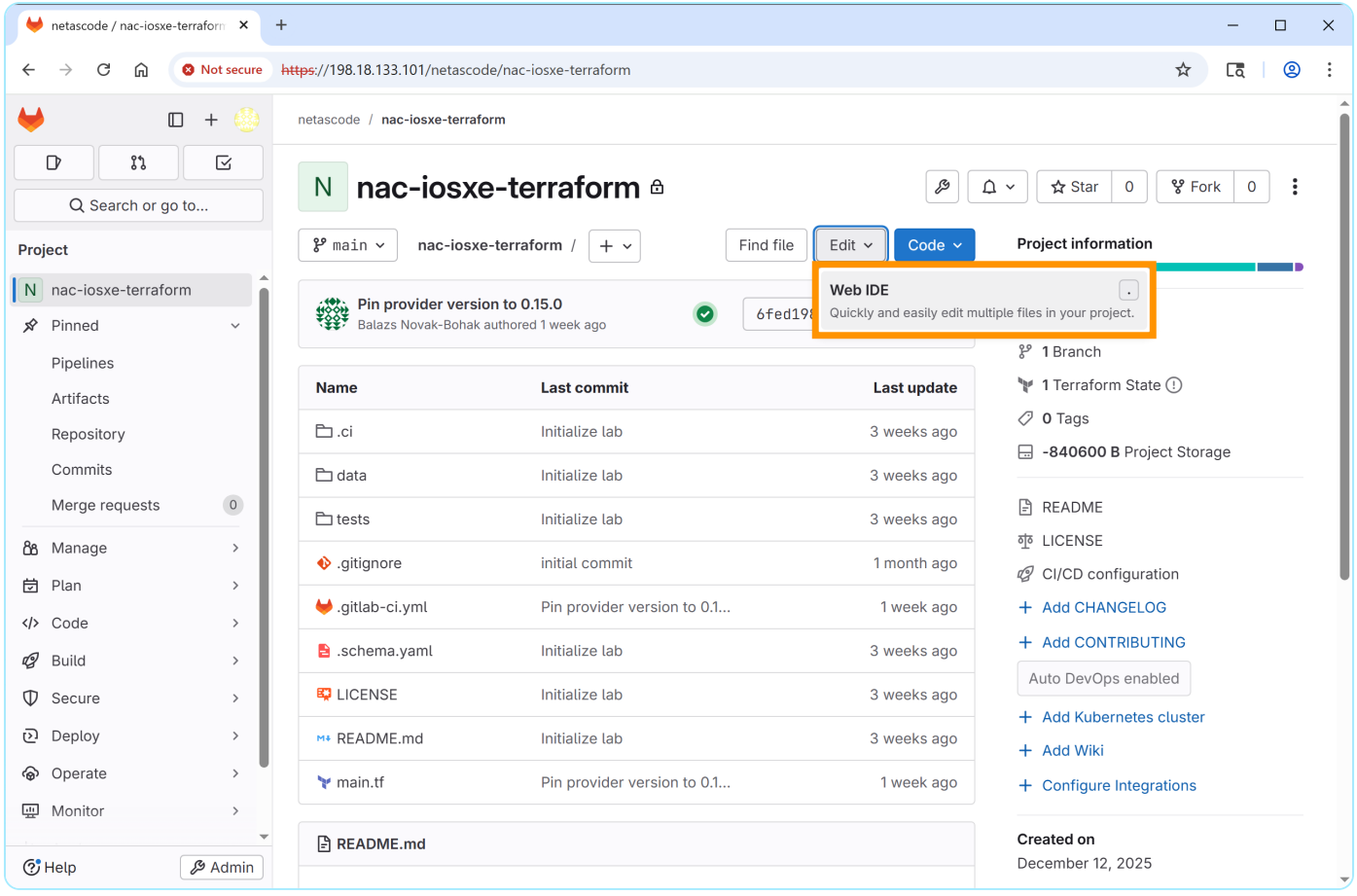
Step 3: Make a change

> **This step deploys real configuration to the lab devices**
>
> The moment you commit to `main`, the pipeline's **deploy** stage runs `terraform apply` against the CML devices over NETCONF - exactly like you did by hand in Task 03. The first pipeline run after Task 12 will re-push global + per-device config to all four switches. That's expected here (the whole point of Task 13) but worth naming explicitly: **you are about to trigger a real device change from a web UI**. In production, this is exactly the power commit-to-main protected branches are meant to gate (Task 15 covers that).

The best way to see the CI/CD pipeline in action is to make a configuration change. You'll add the global configuration (from Task 03 - Global Configuration and Task 06 - Variables). This includes the login banner and hostnames. To edit the configuration files, you'll use GitLab's built-in **Web IDE** - an editor similar to VS Code that runs directly in your browser.

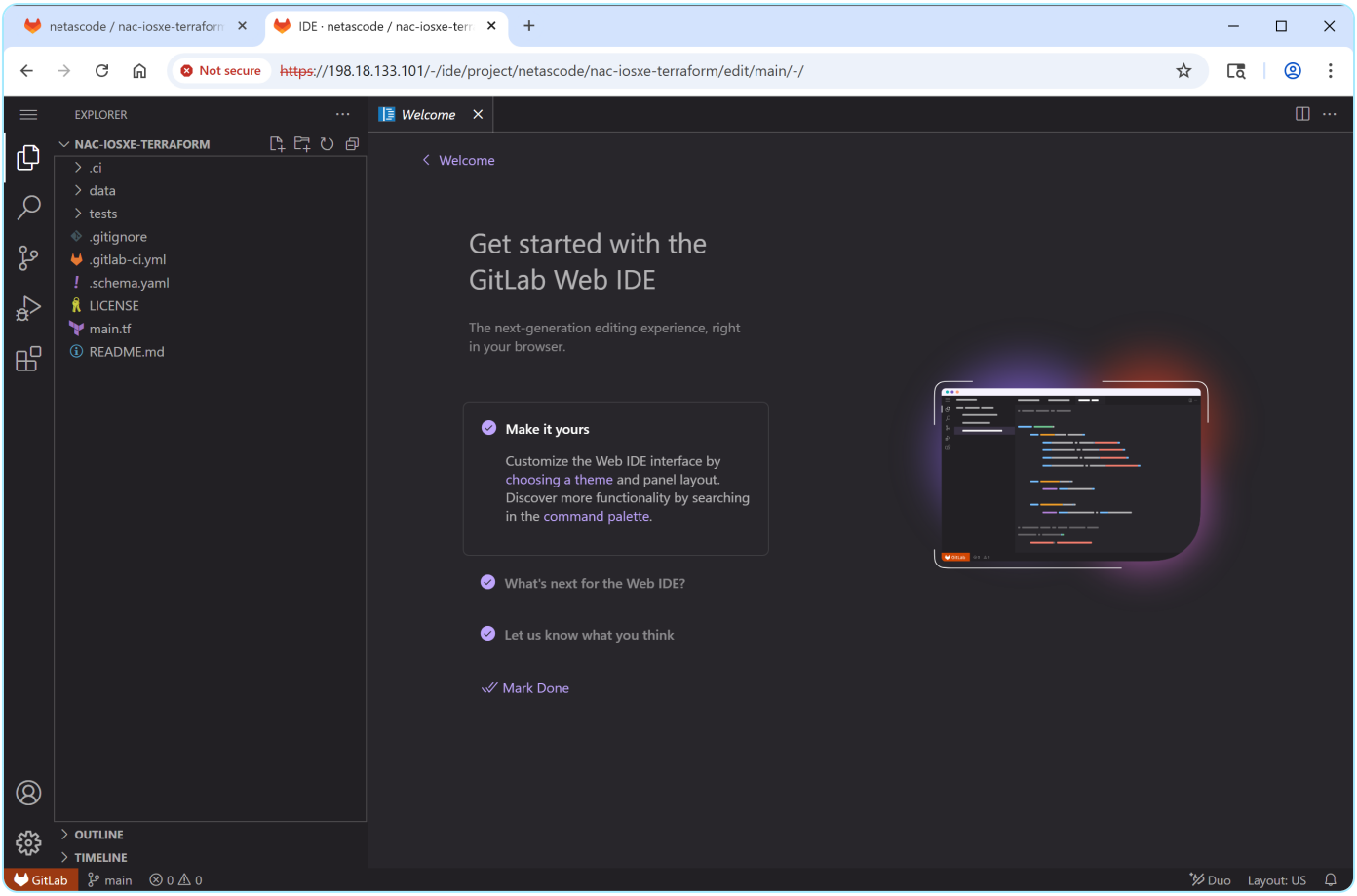
OPEN THE WEB IDE

1. From the project page, click the **Edit** dropdown button (with a pencil icon)
2. Select **Web IDE**



The Web IDE opens with a familiar VS Code-like interface:

Copyright © Cisco



LAB CONFIGURATION FILES

Take a look at the `data/` folder in the file explorer (left panel). This folder contains the same configuration files used in Tasks 2-6. However, some file extensions have been changed from `.yaml` to `.yaml_` (with an underscore at the end). This was done intentionally to ignore these files for now.

> **`.yaml_` vs. `.yaml` files**
>
> The Network as Code framework only uses `.yaml` files from the `yaml_directories` defined in `main.tf` (in our case, the `data/` folder). Files with other extensions (like `.yaml_`) are ignored.

For this task, you'll rename one file (the global config from Tasks 03 and 06) to enable it, and let the pipeline do the rest.

ADD GLOBAL CONFIGURATION

1. In the file explorer (left panel), navigate to the **data** folder
2. Find the file `global.nac.yaml_` (note the underscore at the end)
3. Right-click on the file and select **Rename**
4. **Rename the file** from `global.nac.yaml_` to `global.nac.yaml` (remove the underscore)
5. Click on the file to open it and inspect the banner content:

FILE **global.nac.yaml**

```
---
iosxe:
```


Copyright © Cisco

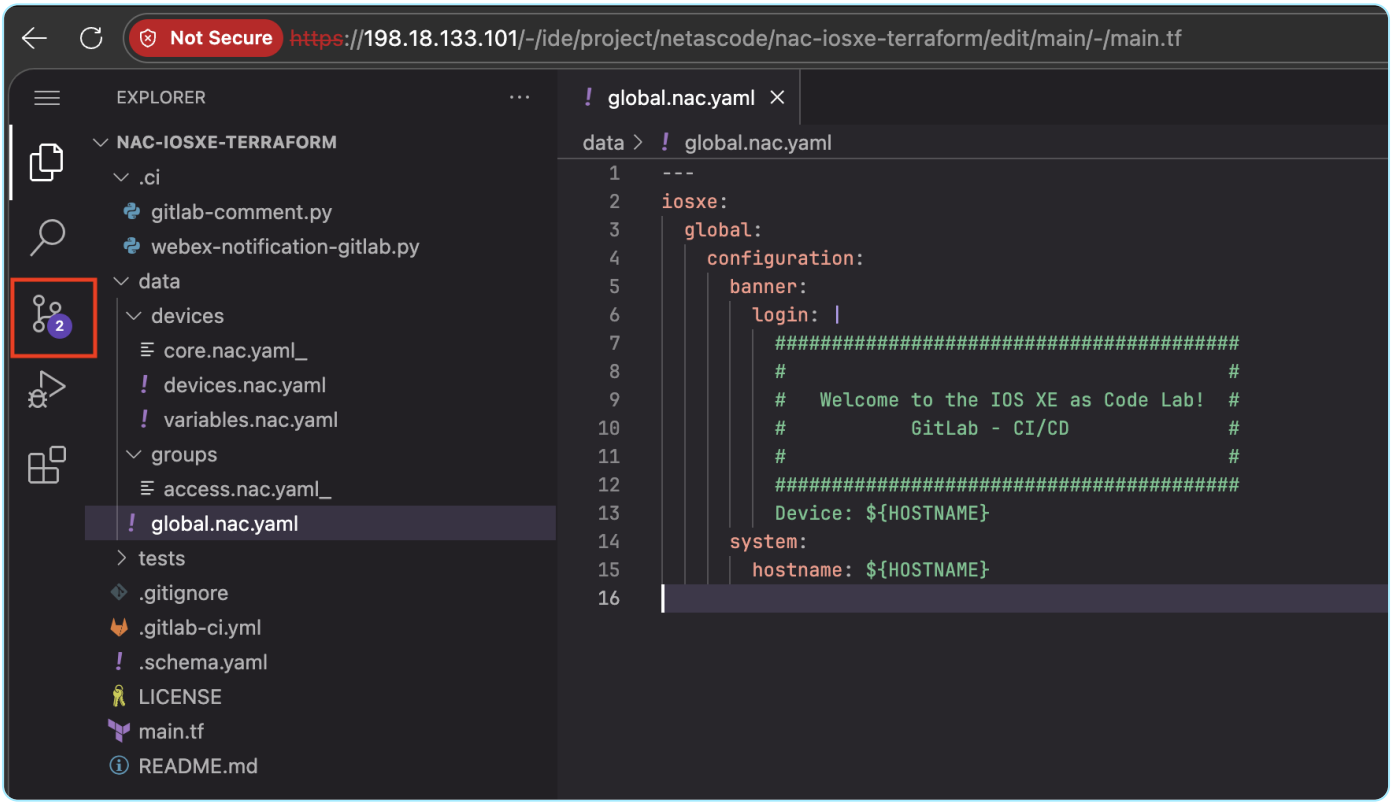
```
global:
  configuration:
    banner:
      login: |
        ########################################
        #                                      #
        #   Welcome to the IOS XE as Code Lab!  #
        #           GitLab - CI/CD             #
        #                                      #
        ########################################
        Device: ${HOSTNAME}
    system:
      hostname: ${HOSTNAME}
```

Banner Modification

Note that the banner text has been modified to indicate that GitLab CI/CD is now being used.

Optionally, you can also change the banner text to something new, if you'd like.

Step 4: Commit change to trigger pipeline

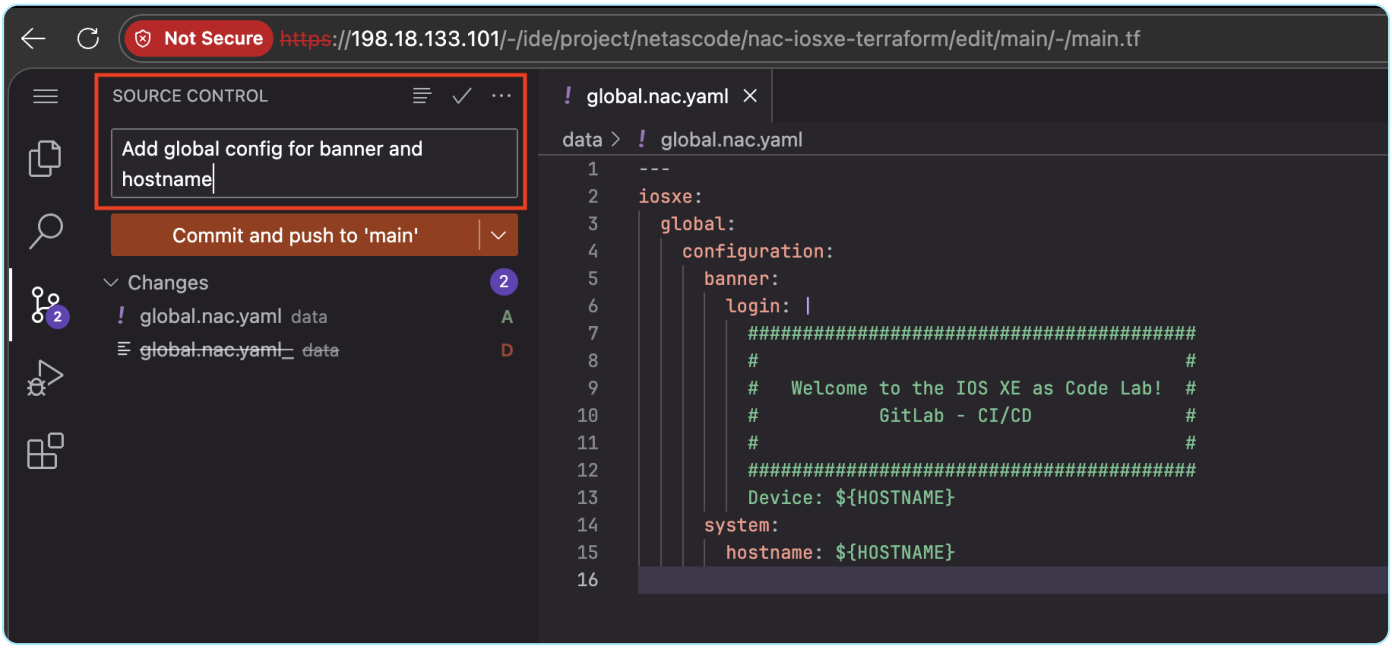


1. Click on **Source Control** icon in the left sidebar
2. You'll see your modified file listed
3. Enter a commit message:

```
Add global config for banner and hostname
```

4. Click **Commit and push to 'main'**
5. If prompted with “You're committing your changes to the default branch. Do you want to continue?”. Select **Continue**.

Copyright © Cisco



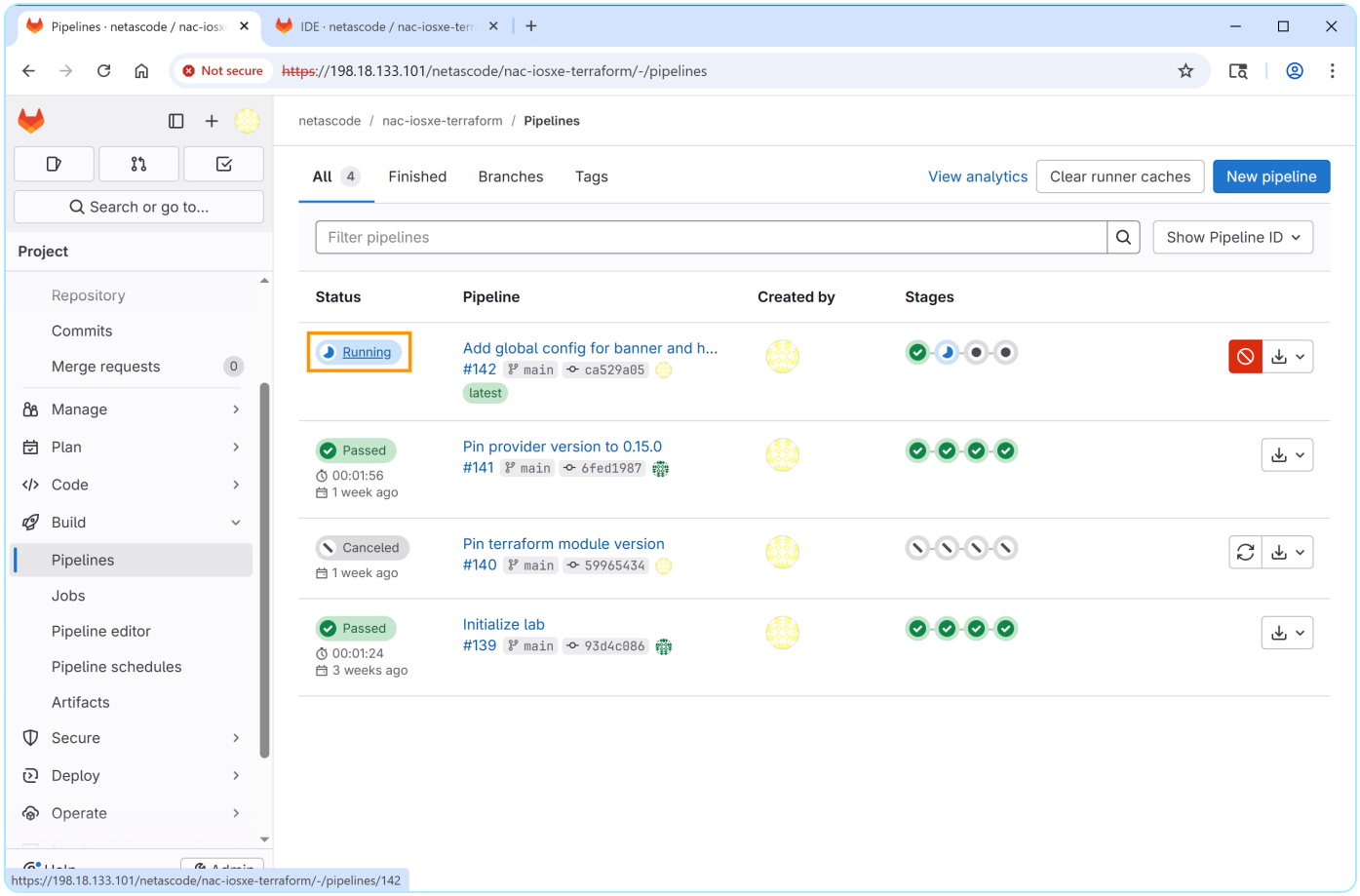
Pipeline Auto-Trigger

When you commit to the `main` branch, GitLab automatically triggers the CI/CD pipeline. No manual action required!

Step 5: Monitor pipeline execution

To view the pipeline progress, navigate to **Build** → **Pipelines** in the left sidebar, then click on the top pipeline showing **running** status.

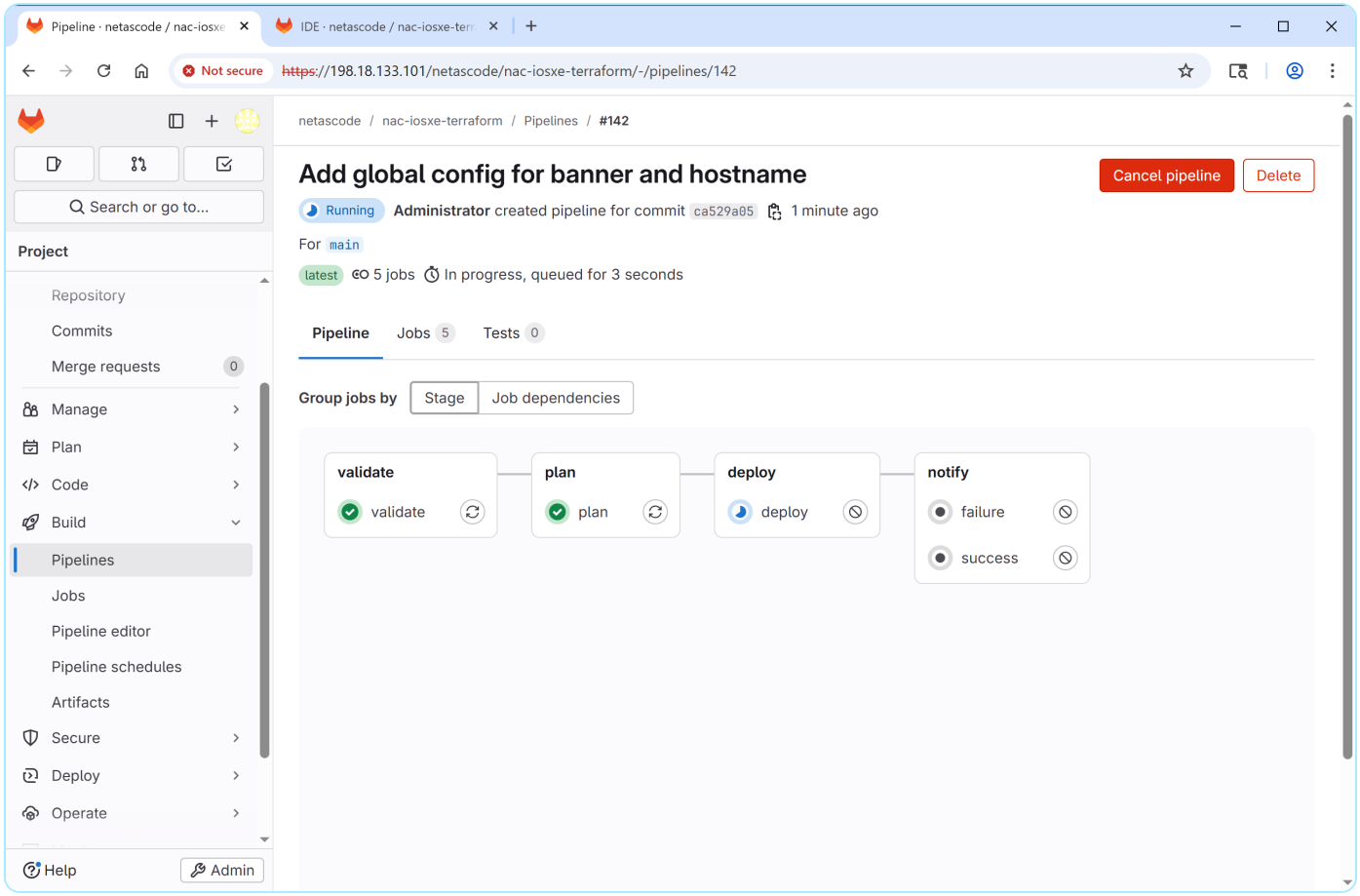
Copyright © Cisco



Tip

You need to click on the pipeline status icon or the pipeline ID.

Copyright © Cisco



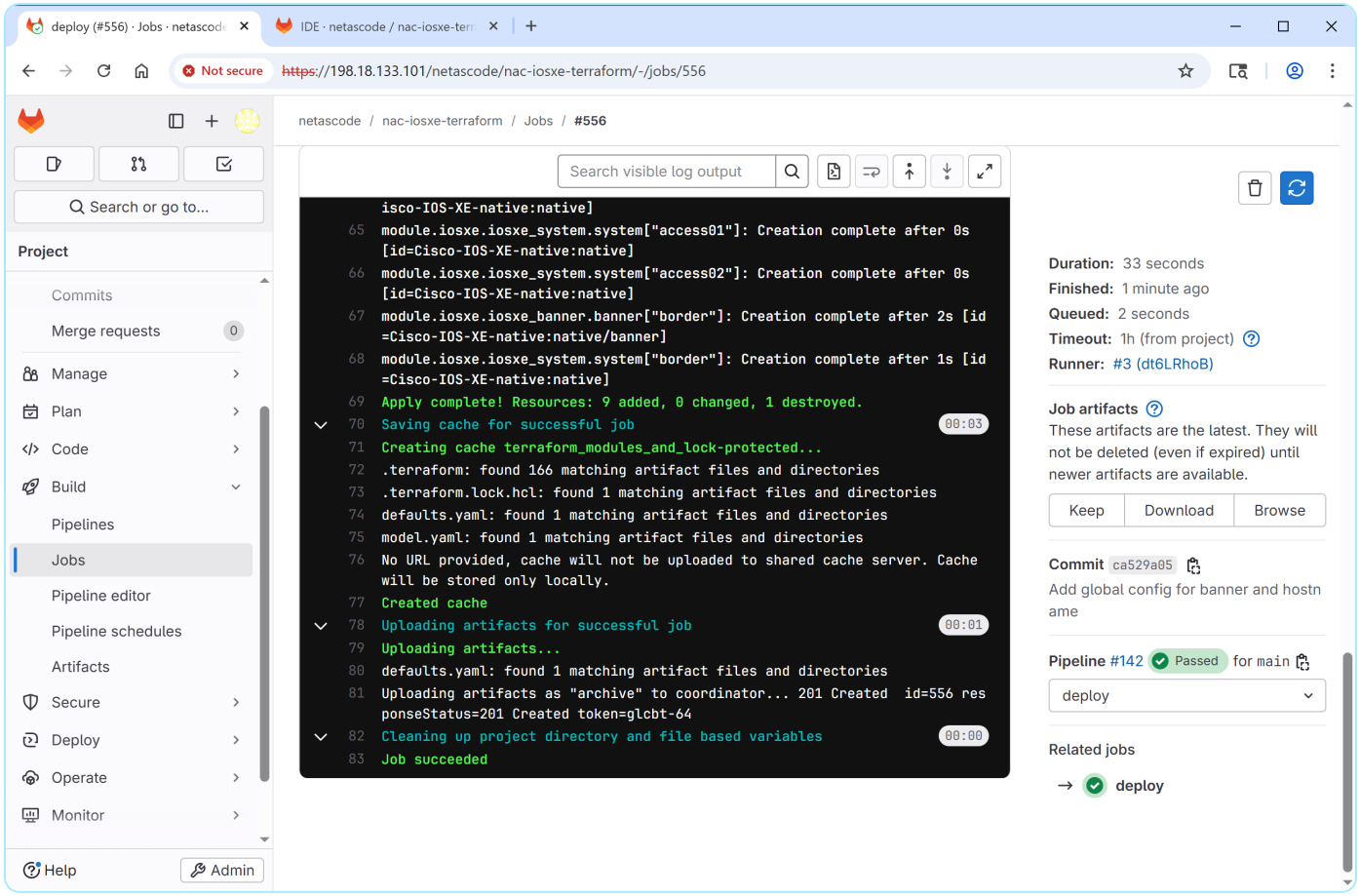
You'll see each stage progressing:

1. **validate** - green checkmark when YAML validation passes. *Usually <10s.*
2. **plan** - generates the Terraform plan against the live devices. *~30-60s.*
3. **deploy** - applies the configuration to the four CML devices over NETCONF. *~60-90s.*
4. **notify** - sends success or failure notifications (not used in this lab). *<5s.*

End-to-end a healthy pipeline run completes in about **2-3 minutes**. If the `deploy` stage sits on "running" for more than 4-5 minutes, one of the CML devices is probably unreachable - click into the job to see the NETCONF error. (The troubleshooting section below covers the most common causes.)

Click on any stage to view its detailed logs.

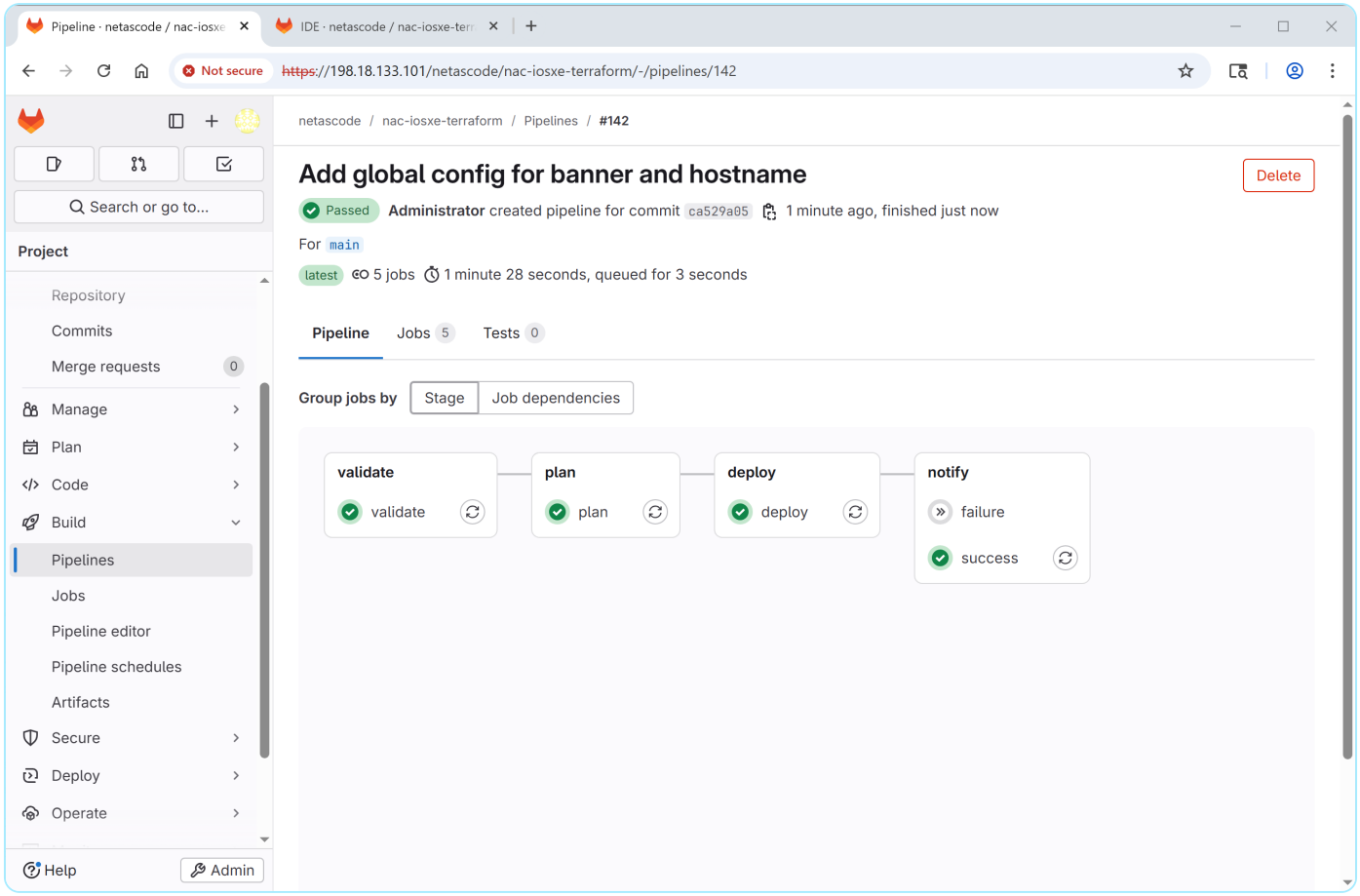
Copyright © Cisco



Step 6: Verify pipeline success

When all stages complete successfully, the pipeline shows a green **passed** status.

Copyright © Cisco



VERIFY THE DEPLOY JOB ACTUALLY CHANGED THE DEVICES

A green pipeline only tells you the stages *finished without error* - it doesn't by itself prove the devices changed. A no-op deploy (running on a state that already matches intent) also shows green. To confirm Terraform actually pushed something, click into the **deploy** job and scroll to the end of the log. You're looking for a line like:

OUTPUT **deploy job tail**

```
Apply complete! Resources: 12 added, 0 changed, 0 destroyed.
```

- **Resources > 0** in `added` or `changed` - the devices were modified.
- **All zeros** - the pipeline ran against a state that already matched; nothing was pushed. That's fine, but if you expected a change, verify your commit made it into `main` and re-trigger the pipeline.

This log-based check is more reliable than looking at device output because it comes straight from the Terraform engine's own accounting.

SPOT-CHECK A DEVICE (OPTIONAL BUT SATISFYING)

1. Open **Solar-PuTTY** from your desktop
2. Connect to one of the devices (e.g., **core** switch)
3. Check the banner and hostname - The banner will now include `GitLab - CI/CD`

Copyright © Cisco



Validation test (optional)

Additionally, you can test the validation stage by introducing an error in the configuration, just like in Task 10 - Schema Validation.

If the validation fails, the pipeline will stop, and you'll see a red **failed** status.

Troubleshooting common pipeline failures

If your pipeline shows a red **failed** status, click into the failing job to see the logs. Here are the failure modes you're most likely to hit during the lab:

`nac-validate` reports a schema error

Symptom: The `validate` stage fails with an `ERROR - Syntax error 'data/…'` message.

Fix: Open the file named in the error and fix the reported issue. Schema errors in the pipeline are the same errors you'd see running `nac-validate` locally - same tool, same rules. Refer back to Task 10 - Schema Validation for the full error catalog and fix patterns.

`terraform plan` or `apply` can't reach the devices

Symptom: `deploy` stage fails with an error mentioning `dial tcp`, `connection refused`, or `Error: Connection to host failed`.

Likely causes (in order of frequency):

1. `IOSXE_USERNAME` / `IOSXE_PASSWORD` variables weren't supplied when you ran the pipeline. Re-run with **Run pipeline** and fill in the credentials.
2. The CML lab devices are down. Check CML UI and restart the stopped node.
3. Wrong `IOSXE_PROTOCOL` value. Should be `netconf` for this lab (or `restconf` if you opted out). Anything else, and the provider won't know how to talk to the device.

Copyright © Cisco

`terraform apply` fails with `Error acquiring the state lock`

Symptom: `deploy` stage fails with `Error acquiring the state lock` and a lock ID.

Fix: An earlier pipeline run died mid-apply and left the HTTP backend state locked. Unlock it from the command line:

TERMINAL · WSL/Ubuntu cisco@wkst1:~/nac-iosxe$

```
terraform force-unlock <lock-id>
```

Or, from GitLab: **Operate → Terraform states → nac-iosxe-terraform → Remove lock**.

Then re-run the pipeline. Only do this if you're confident no other apply is actually in progress - force-unlocking a live apply will corrupt state.

Pipeline blocked: "You cannot push to this protected branch"

Symptom: The commit is rejected before the pipeline even starts.

Fix: The target branch is protected (see Task 15). Create a feature branch instead, push to it, open a merge request. This is expected behavior once protected-branch rules are in place - it's the system telling you to use the MR workflow.

Need more signal? Turn on Terraform's debug log

When a pipeline job logs a generic "apply failed" without naming the offender, re-run the job with Terraform's debug output. Add two env vars to the pipeline run (**Run pipeline** → custom variables):

```
TF_LOG=DEBUG
TF_LOG_PATH=terraform-debug.log
```

`TF_LOG=DEBUG` prints every RPC the provider makes, including the NETCONF XML payloads. `TF_LOG_PATH` writes the same output to a file you can attach as a CI job artifact - much more usable than scrolling through the job log. Other levels: `TRACE` (noisier), `INFO`, `WARN`, `ERROR`.

Locally, the same env vars work in your shell: `TF_LOG=DEBUG terraform plan`. Turn it off (`unset TF_LOG`) once you've found the problem - debug output is verbose.

What you've accomplished

- Accessed GitLab and navigated to the NaC-IOSXE project
- Understood the CI/CD pipeline configuration
- Triggered a pipeline by committing a change
- Monitored pipeline execution through all stages

What's next

Tasks 14 and 15 are **optional** - Task 14 extends the pipeline with integration and idempotency tests; Task 15 introduces merge-request workflows with protected branches. If you're at the end of your time box, skip straight to the Conclusion.

Copyright © Cisco

Task 12 - Cleanup

Task 14 - Extend CI/CD pipeline

Copyright © Cisco

Task 14 - Extend the pipeline with automated tests (Optional)

Estimated Time to Complete: ~15 minutes

In Task 13 you ran a CI/CD pipeline with validate, plan, and deploy stages. In this task you'll extend it with a **test stage** that automatically validates deployments after they're applied - the same thing you ran manually in Task 11, now on every merge.

What you'll learn

By the end of this task you will have:

- Edited `.gitlab-ci.yml` to add a new `test` stage between `deploy` and `notify`
- Added a `test-integration` job that runs `nac-test` against the live devices
- Added a `test-idempotency` job that runs `terraform plan -detailed-exitcode` to catch configuration drift
- Read the test results as JUnit reports attached to the pipeline

Test stage

Adding automated testing to your CI/CD pipeline ensures that:

- Configurations are correctly applied to devices
- The deployment is idempotent (running it again produces no changes)
- Any issues are detected immediately after deployment

You'll add two test jobs:

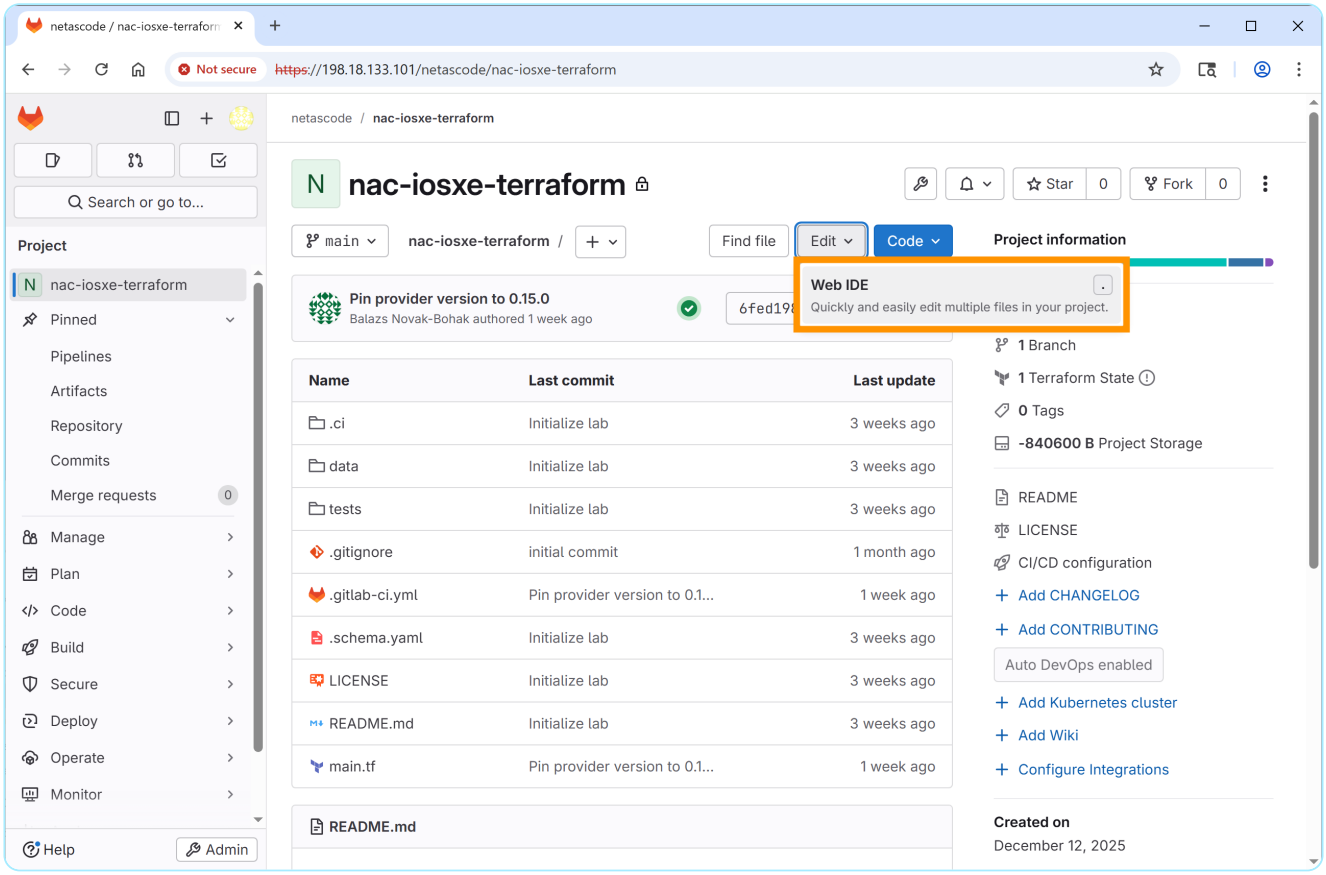
- `test-integration` - Runs `nac-test` to verify configurations match expected state
- `test-idempotency` - Runs `terraform plan` again to confirm no drift

Copyright © Cisco

Step 1: Open pipeline definition file

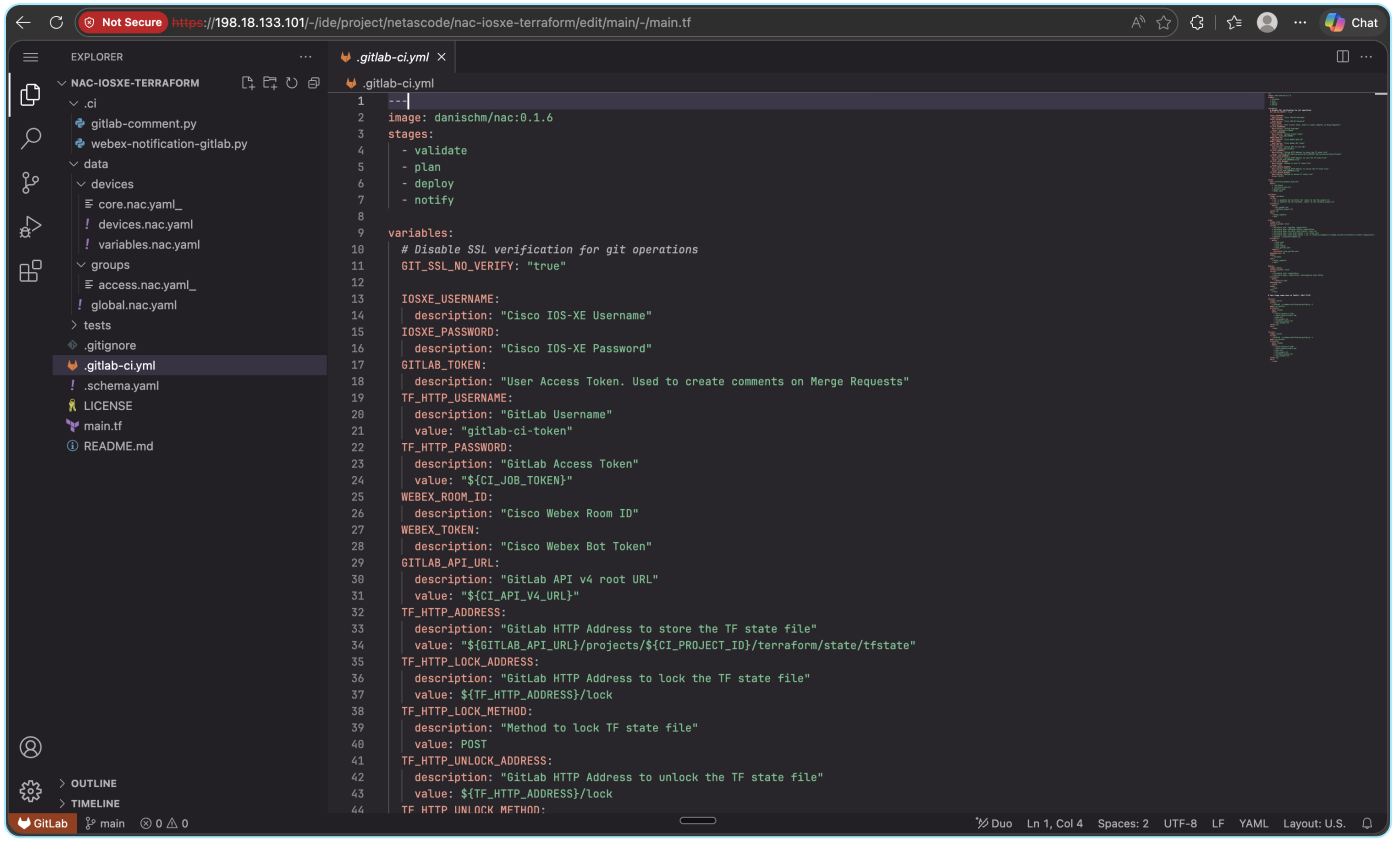
How to Open Web IDE

1. In **Chrome inside your RDP session** (Windows 10 VM), open GitLab in a new tab: https://198.18.133.101
2. Navigate to the **netascode/nac-iosxe-terraform** project
3. From the project page, click the **Edit** dropdown button (with a pencil icon)
4. Select **Web IDE**



In the Web IDE, click on `.gitlab-ci.yml` in the Explorer panel to open it for editing.

Copyright © Cisco



Step 2: Add the test stage

Find the `stages` section at the top of the file. You need to add `test` between `deploy` and `notify`.

Find this section:

FILE **.gitlab-ci.yml**

```
stages:
  - validate
  - plan
  - deploy
  - notify
```

Add `test` so it looks like this:

FILE **.gitlab-ci.yml**

```
stages:
  - validate
  - plan
  - deploy
  - test
  - notify
```

Step 3: Add the test-integration job

After the `deploy` job section (around line 115), add the `test-integration` job. This job runs `nac-test` to verify your configurations.

Add this new job after the `deploy:` section:

FILE **.gitlab-ci.yml (add this job)**

Copyright © Cisco

```
test-integration:
  stage: test
  script:
    - echo "=== PWD and tree ==="; pwd; ls -la
    - echo "=== Inputs nac-test needs ==="; ls -la model.yaml defaults.yaml || true
    - echo "=== tests/ scaffolding ==="; ls -la tests/ tests/templates tests/filters || true
    - echo "=== nac-test version ==="; nac-test --version || true
    - mkdir -p tests/results
    -
set -o pipefail && nac-test --data ./model.yaml --data ./defaults.yaml --templates ./tests/templates --filters ./tests/filters --output ./tests/results |&
tee test_output.txt
  after_script:
    - echo "=== tests/results contents (post-run) ==="; ls -la tests/results || true
    - echo "=== last 50 lines of test_output.txt ==="; tail -n 50 test_output.txt || true
  artifacts:
    when: always
    paths:
      - tests/results/
      - test_output.txt
    reports:
      junit: tests/results/xunit.xml
  dependencies:
    - deploy
  needs:
    - deploy
  only:
    - main
```

What this job does:

- The pre- `nac-test` `echo` / `ls` lines print the working directory, the data files (`model.yaml` , `defaults.yaml`), and the `tests/` scaffolding so a missing input is obvious in the log instead of being inferred from a downstream failure.
- `mkdir -p tests/results` guarantees the output directory exists before `nac-test` writes to it, so the artifact glob never trips on a missing parent.
- `script` : Runs `nac-test` with your data files and test templates.
- `after_script` : Always runs (even on failure) and prints the contents of `tests/results/` plus the tail of `test_output.txt` , so you see what `nac-test` actually produced.
- `artifacts` : Uploads the entire `tests/results/` directory (HTML reports, log, output.xml, xunit.xml) so a partial run is still inspectable.
- `reports: junit` : Integrates test results into GitLab's test reporting UI.
- `dependencies` and `needs` : Ensure this job runs after `deploy` completes.
- `only: main` : Only runs on the main branch (not merge requests).

Reading the log when this job fails

Scroll to the lines just above `Uploading artifacts ...` in the job log:

- A missing `model.yaml` or `defaults.yaml` in the first `ls` block means the file wasn't passed to this job - check the `deploy` job's artifacts and the top-level `cache:` block.
- A missing `tests/templates` or `tests/filters` means the `tests/` scaffolding isn't committed on `main` .
- `Suite '...' contains no tests or tasks` from Robot Framework means the templates rendered an empty suite - usually because no input data matched (for example, `data/groups/access.nac.yaml_` was never renamed in Step 5, so `model.yaml` has no ACLs to test).

Step 4: Add the test-idempotency job

Add another test job that verifies idempotency - running Terraform again should show no changes if the deployment was successful.

Add this job right after the `test-integration:` job block:

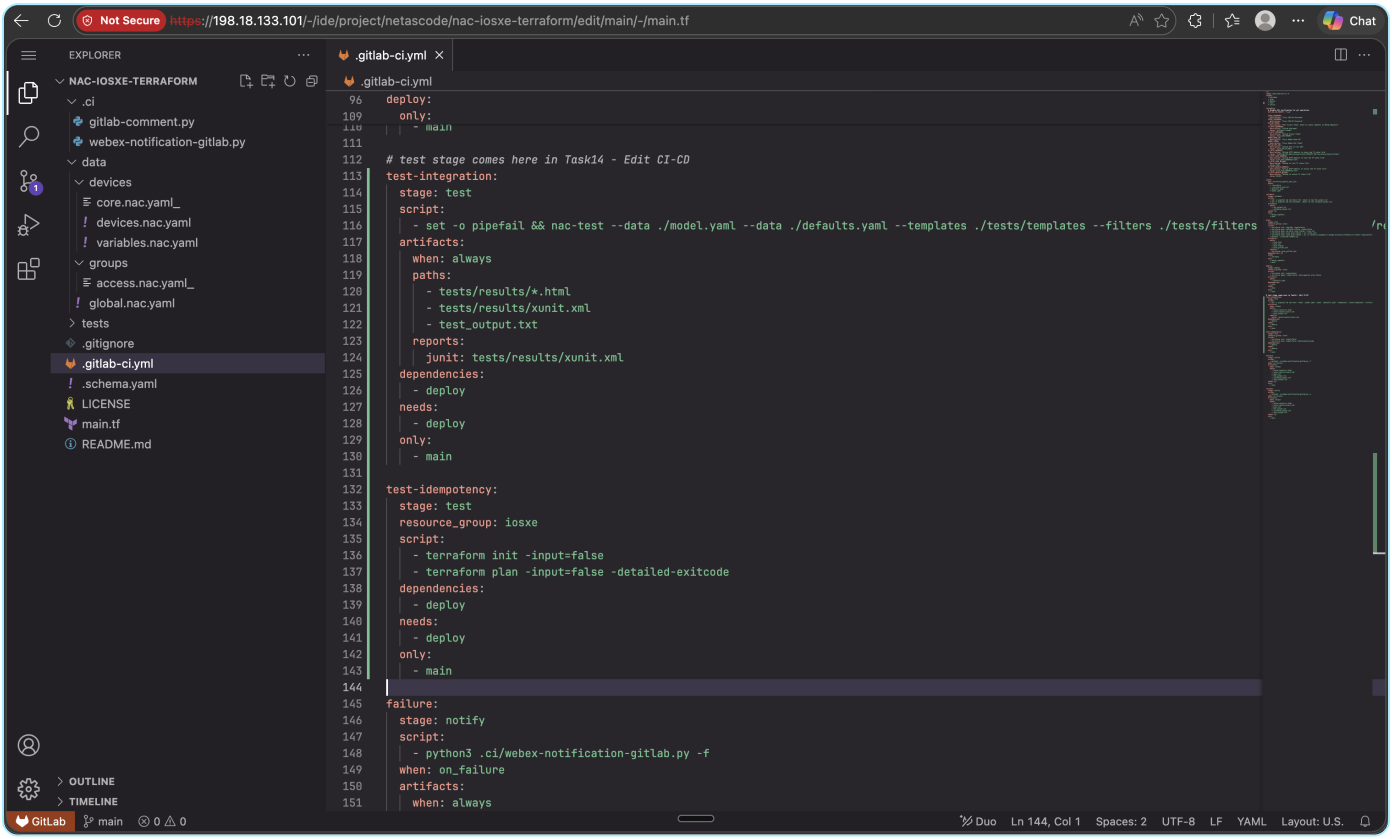
FILE **.gitlab-ci.yml (add this job)**

```
test-idempotency:
  stage: test
  resource_group: iosxe
```


Copyright © Cisco

```
  script:
    - terraform init -input=false
    - terraform plan -input=false -detailed-exitcode
  dependencies:
    - deploy
  needs:
    - deploy
  only:
    - main
```

In the end, your test stage section should look like this:



What the `test-idempotency` job does:

- `terraform plan -detailed-exitcode` : Returns exit code 2 if there are changes, failing the job
- `resource_group: iosxe` : Prevents concurrent access to devices
- If this job passes, it confirms your deployment is idempotent

> **Complete Pipeline Reference**
>
> You can refer to Appendix I for the complete `.gitlab-ci.yml` file with all changes included.

Copyright © Cisco

Step 5: Add ACL configuration

Just as in Task 11 - Post-checks, you will add the ACL configuration to test the pipeline.

1. In the Web IDE file explorer, navigate to `data/` and rename `groups/access.nac.yaml_` to `groups/access.nac.yaml` (remove the trailing underscore).
2. You may review the ACL configuration by opening the file - it defines the standard ACL named `AccessLayerACL` that you configured in Task 4 - Device group configuration. You will notice that additional entries have been added to the Access List.

Step 6: Commit your changes

After making all the changes:

1. In your Web IDE, click on **Source Control** icon in the left sidebar (as you did in Task 13)
2. You'll see the modified files listed: `.gitlab-ci.yml` and `data/groups/access.nac.yaml`
3. Enter a commit message:

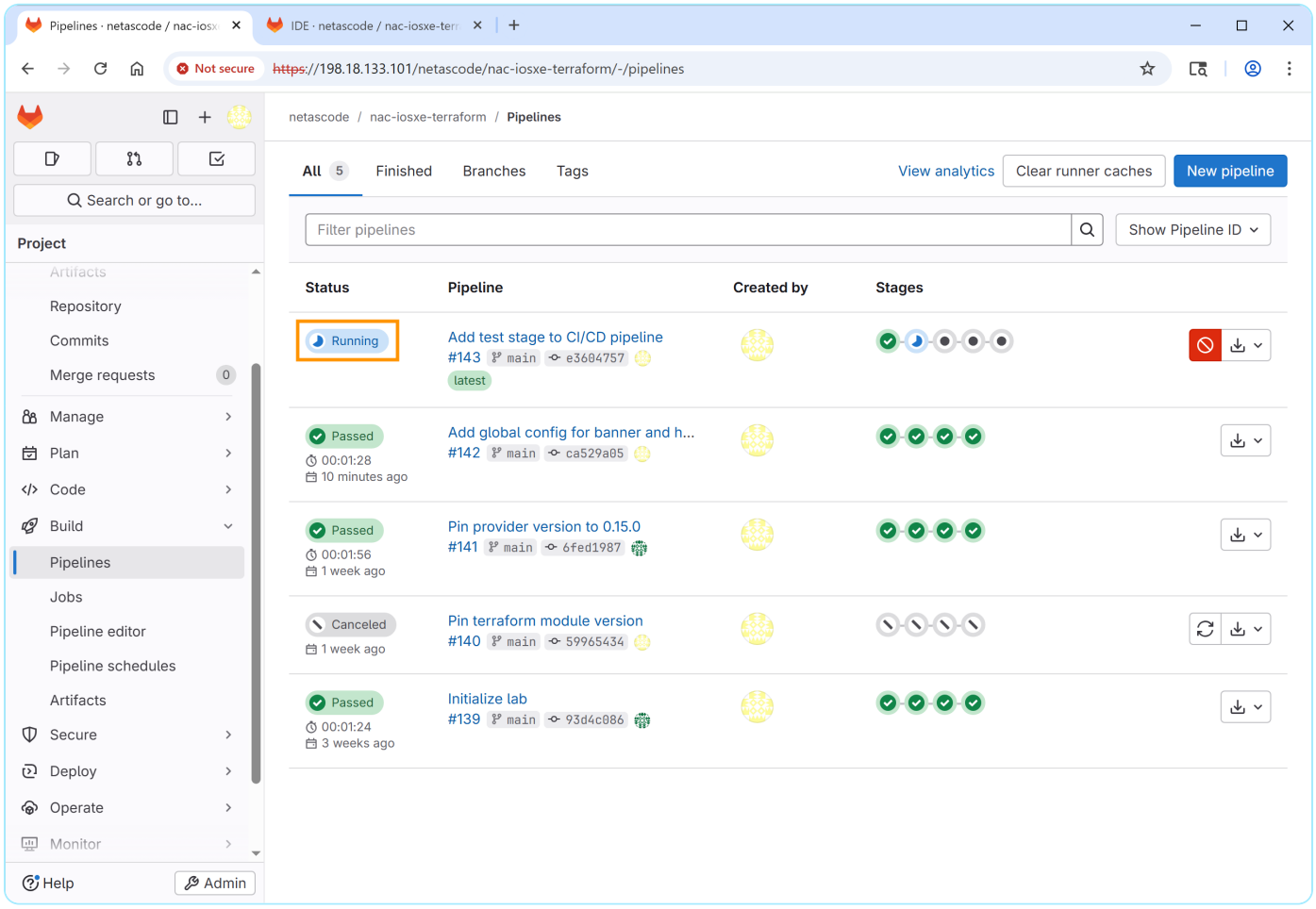
```
Add test stage to CI/CD pipeline
```

4. Click **Commit and push to 'main'**

Step 7: Verify the pipeline

After committing, a new pipeline will automatically start. Navigate to **Build** → **Pipelines** and click on the pipeline showing **running** status to watch its progress.

Copyright © Cisco

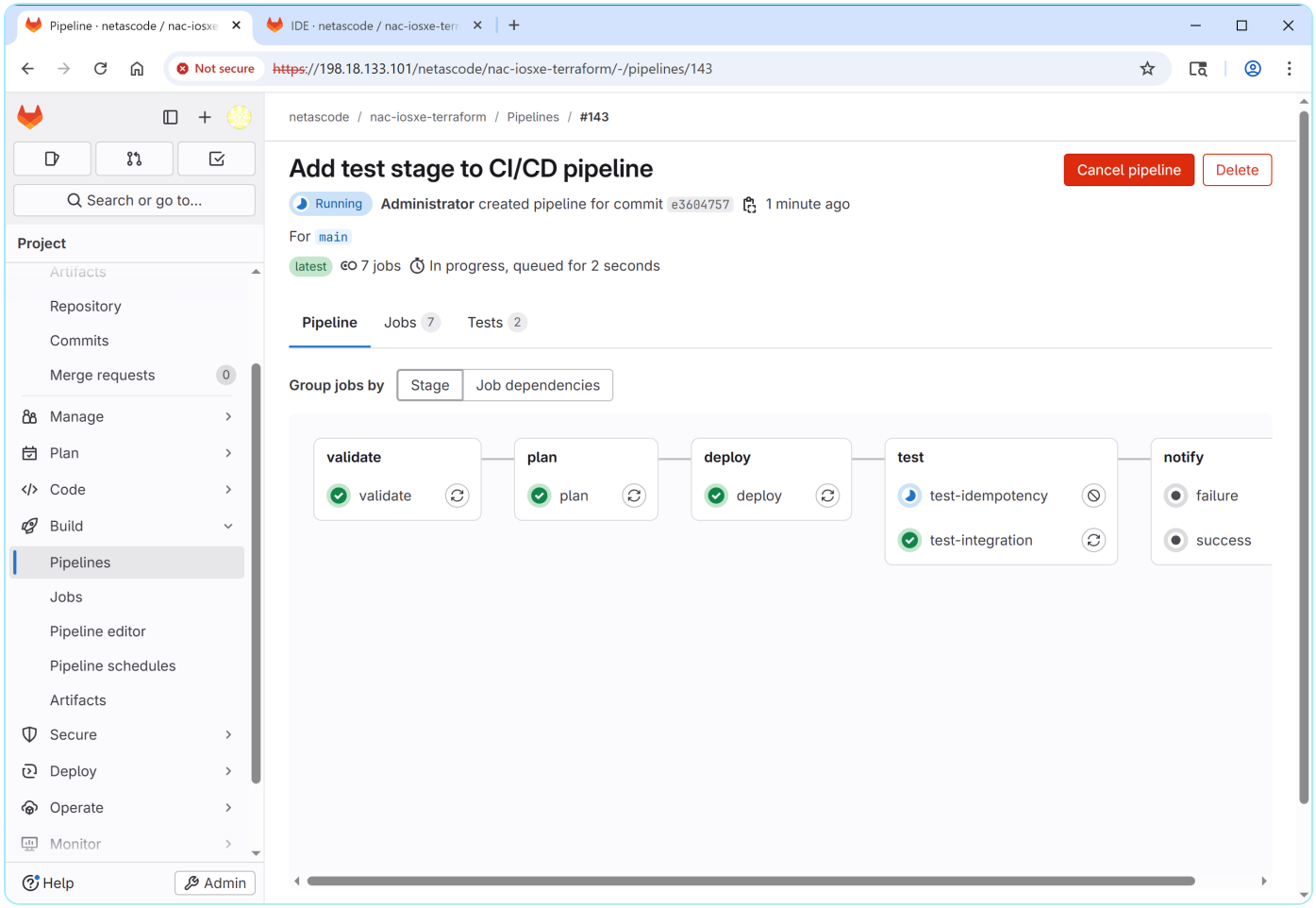


You should now see **5 stages** in the pipeline:

1. **validate** - schema and format validation. *Usually <10s.*
2. **plan** - Terraform planning. *~30-60s.*
3. **deploy** - apply configuration. *~60-90s.*
4. **test** - integration (`nac-test`) + idempotency (`terraform plan -detailed-exitcode`). *~45-60s - nac-test does the heaviest work here.*
5. **notify** - success/failure notifications. *<5s.*

End-to-end with the test stage, expect about **3-4 minutes** total. If `test` sits on running for more than 3 minutes, it's probably waiting on a device that didn't respond - click into the job to see which assertion timed out.

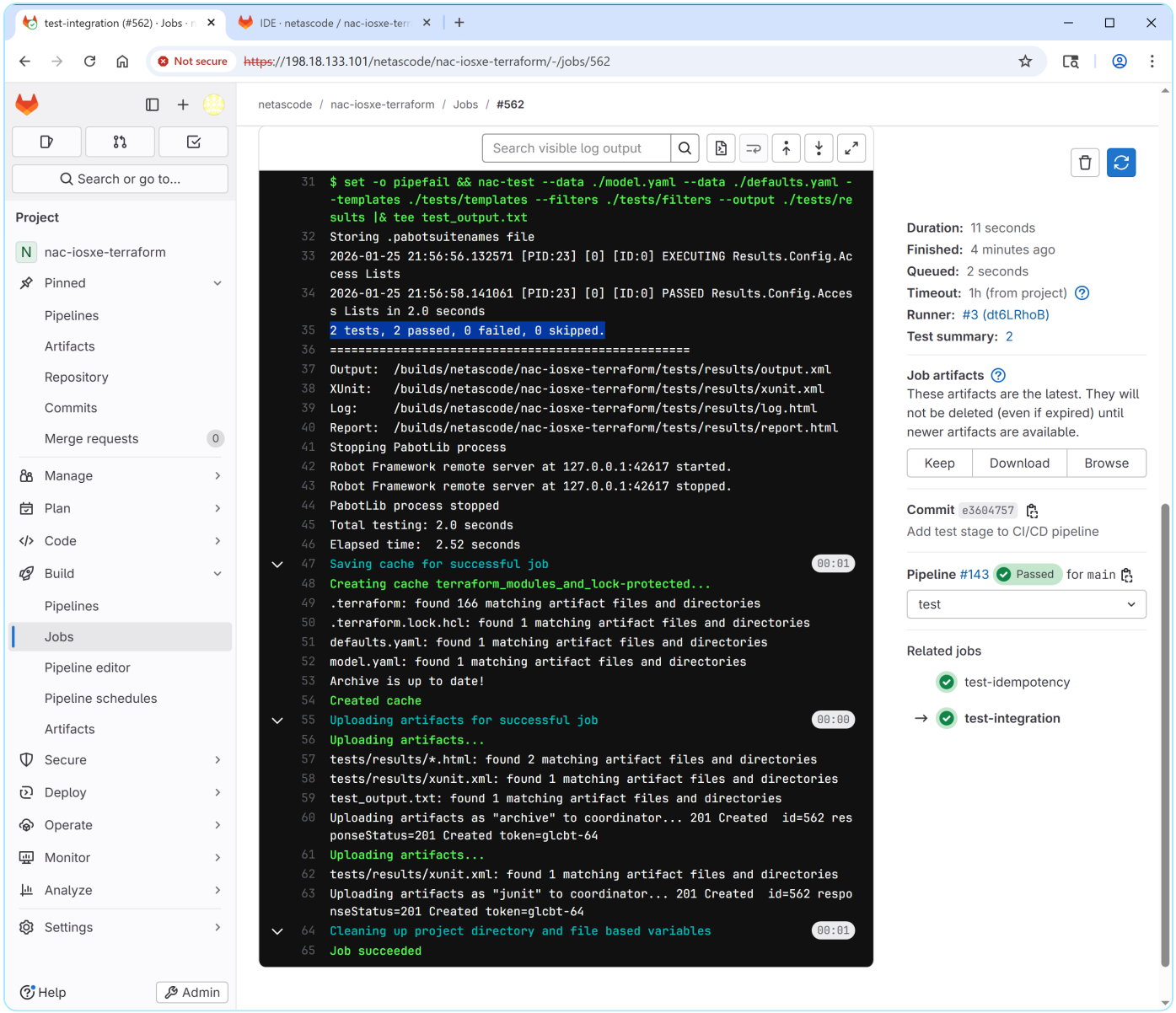
Copyright © Cisco



Step 8: Review test results

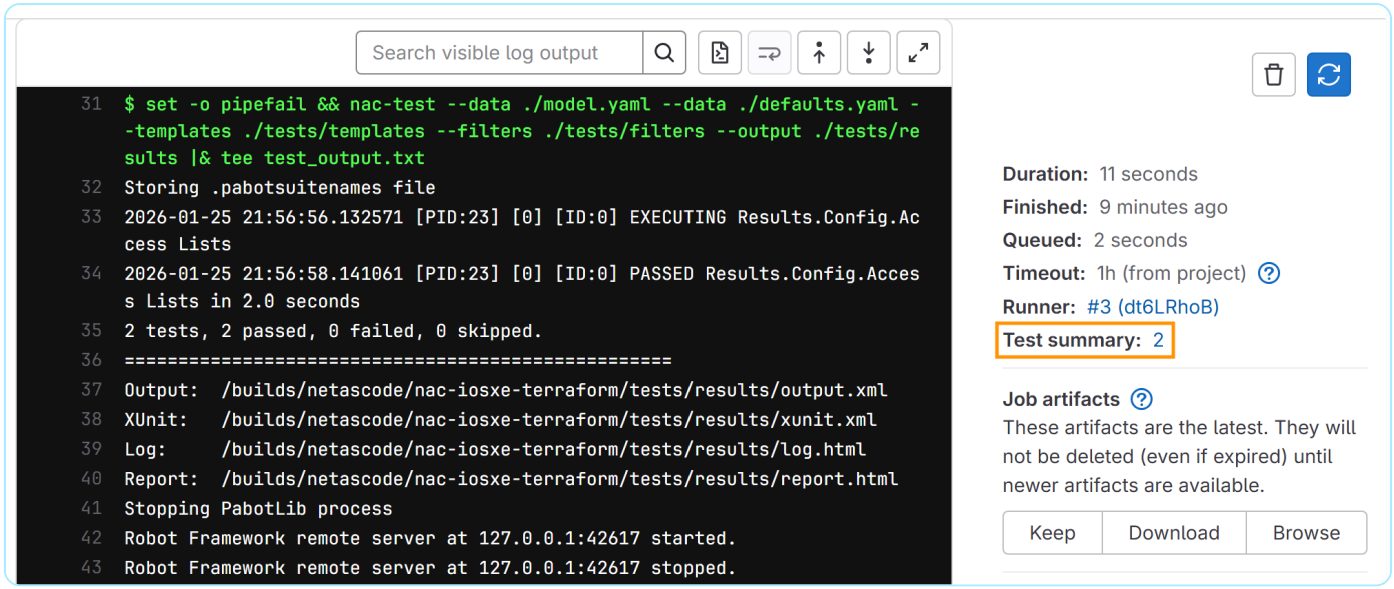
- After the pipeline completes, click on the `test-integration` job to view the test results.
- Review the logs to see the output of `nac-test`. You should see that all tests have passed successfully:
  `2 tests, 2 passed, 0 failed, 0 skipped.`

Copyright © Cisco

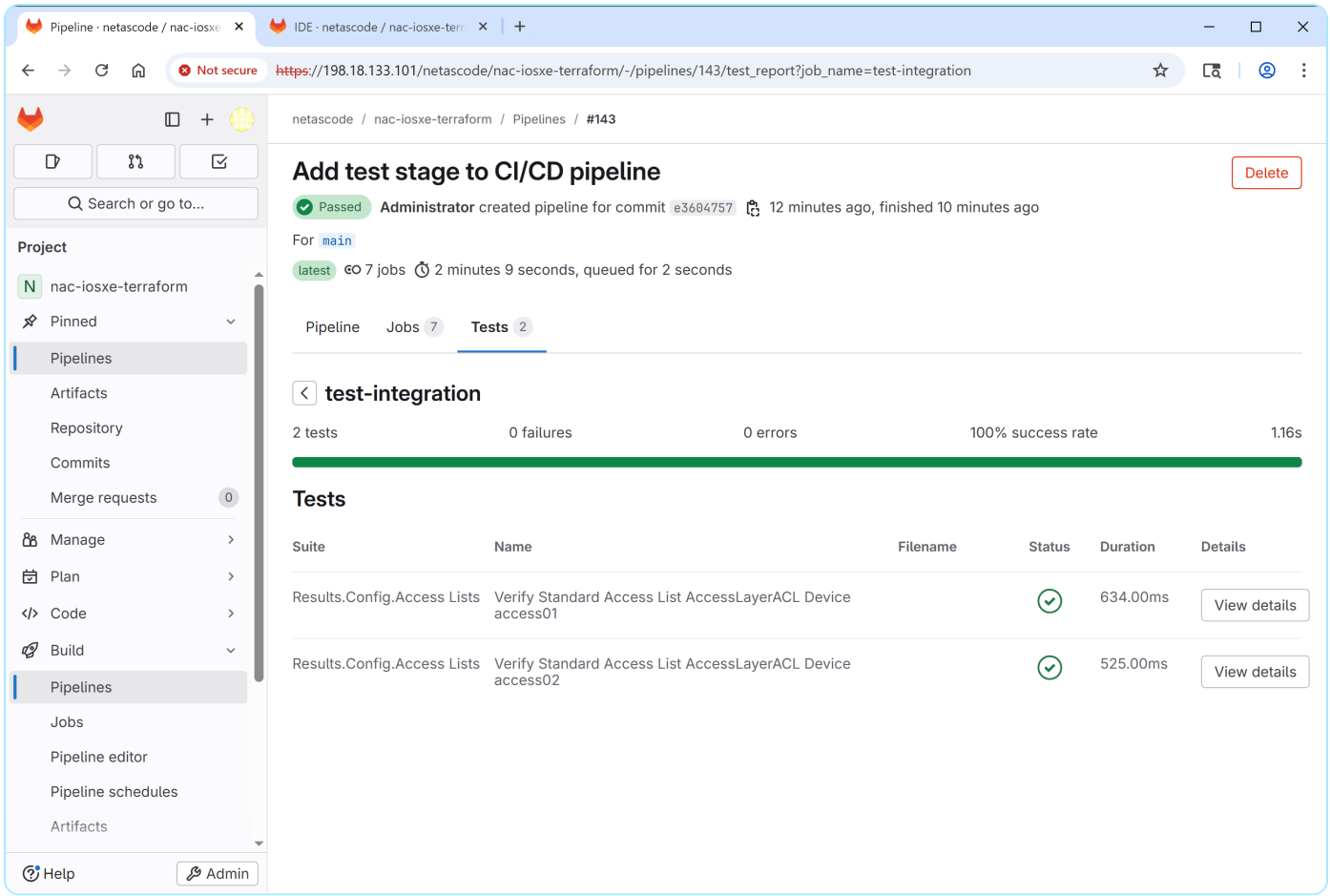


- Click on **Test Summary** on the right sidebar.

Copyright © Cisco

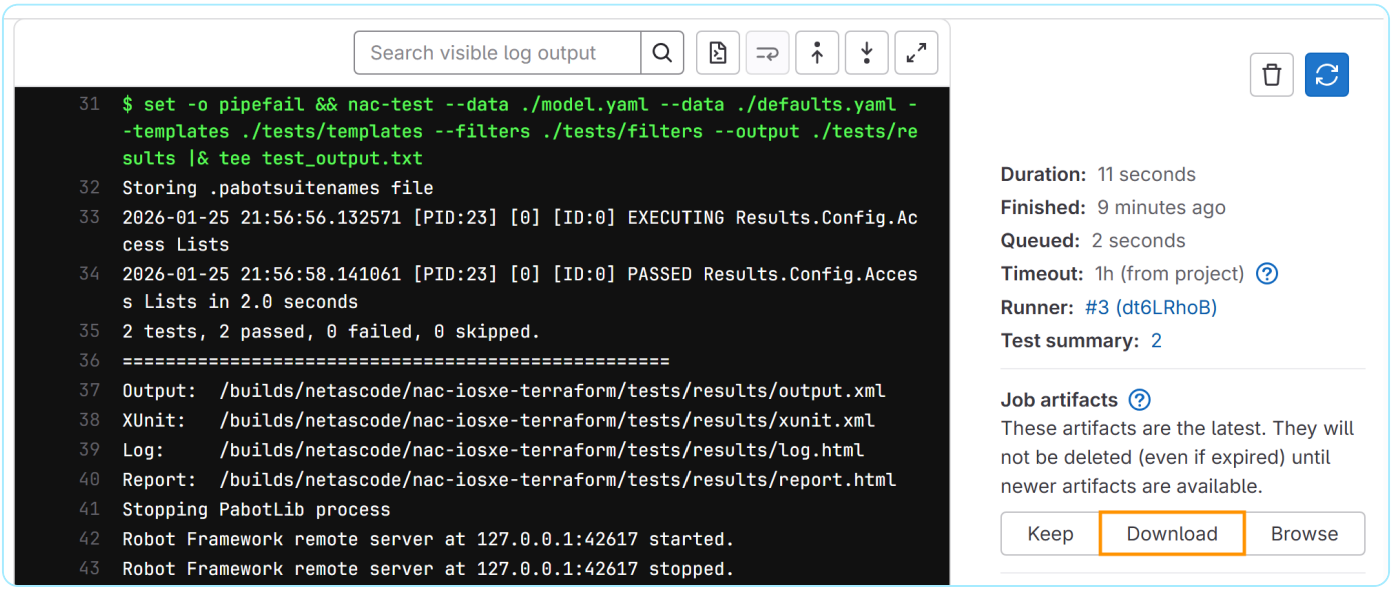


This is where GitLab displays the test results from the JUnit report. You should see a summary indicating all tests passed.



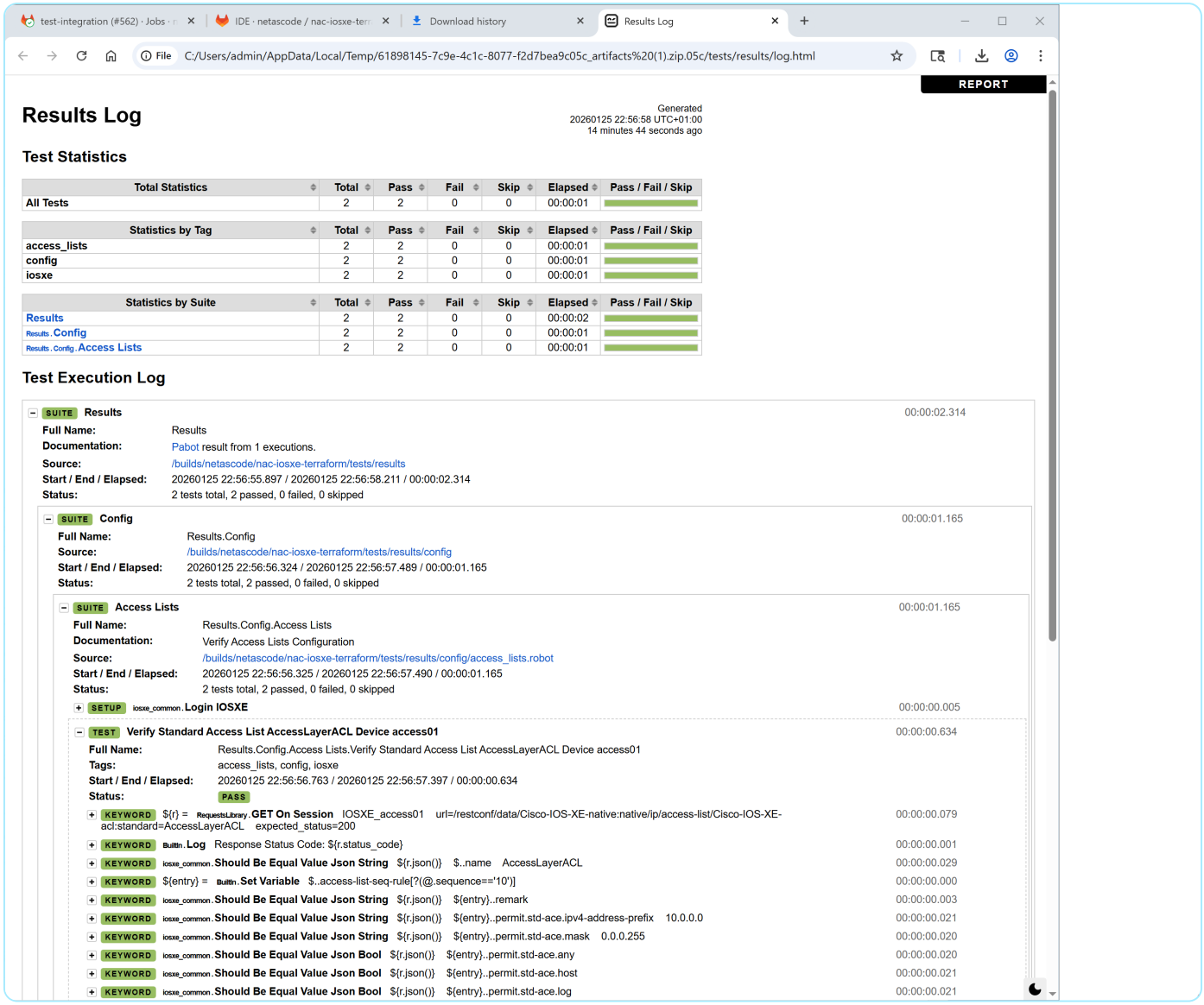
- Go back to the job page and find the **Job artifacts** section on the right sidebar. Click on **Download** to download the test report and log HTML files.

Copyright © Cisco



- Open and inspect the `report.html` and `log.html` files in your web browser. The report will show the same tests as in Task 11 - Post-checks, confirming the configurations were applied correctly.

Copyright © Cisco



What you've accomplished

- Added a test stage to the CI/CD pipeline
- Configured integration tests with `nac-test`
- Added idempotency verification
- Verified the enhanced pipeline runs successfully

What's next

Task 15 is the last **optional** task - it introduces the merge request workflow with protected branches, so changes land on `main` only after review. If you're at the end of your time box, jump to the Conclusion.

Task 13 - Run a CI/CD pipeline

Task 15 - Merge request workflow

Copyright © Cisco

Task 15 - Branch and merge-request workflow (Optional)

Estimated Time to Complete: ~20 minutes

> **Prerequisite**
>
> Basic familiarity with **Git** (branches, commits) and **GitLab** (merge requests, protected branches). If you're new to these, skim Git's Basic Branching guide first.

What you'll learn

By the end of this task you will have:

- Enabled protected-branch rules on `main` so direct commits are blocked
- Created a feature branch, committed to it, and opened a merge request
- Observed two pipelines - the MR preview (validate + plan only) and the post-merge deploy (all five stages)
- Approved and merged a change, seeing GitOps end-to-end: code review → plan preview → deploy

Why this matters

In Tasks 13 and 14, you triggered CI/CD pipelines by committing directly to the `main` branch. While this approach works for learning purposes, **it's not how production environments operate**.

In Network as Code, the `main` branch represents the live production network configuration. In real-world scenarios, many engineers collaborate on the same codebase. To ensure stability, reliability, and accountability, changes must go through a structured review process before being applied to production.

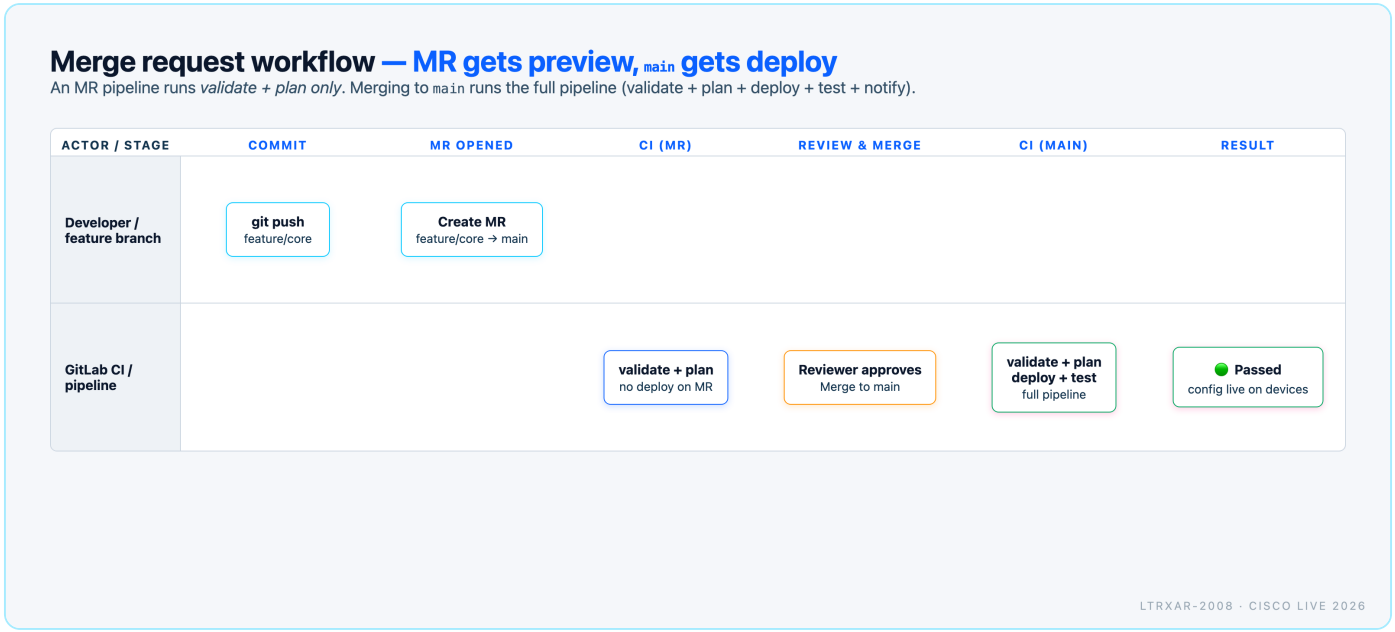
> **Best Practice**
>
> Committing directly to the `main` branch (as done in earlier tasks) is generally **not recommended** in production environments. This task teaches you the proper way to manage changes using branches and merge requests.

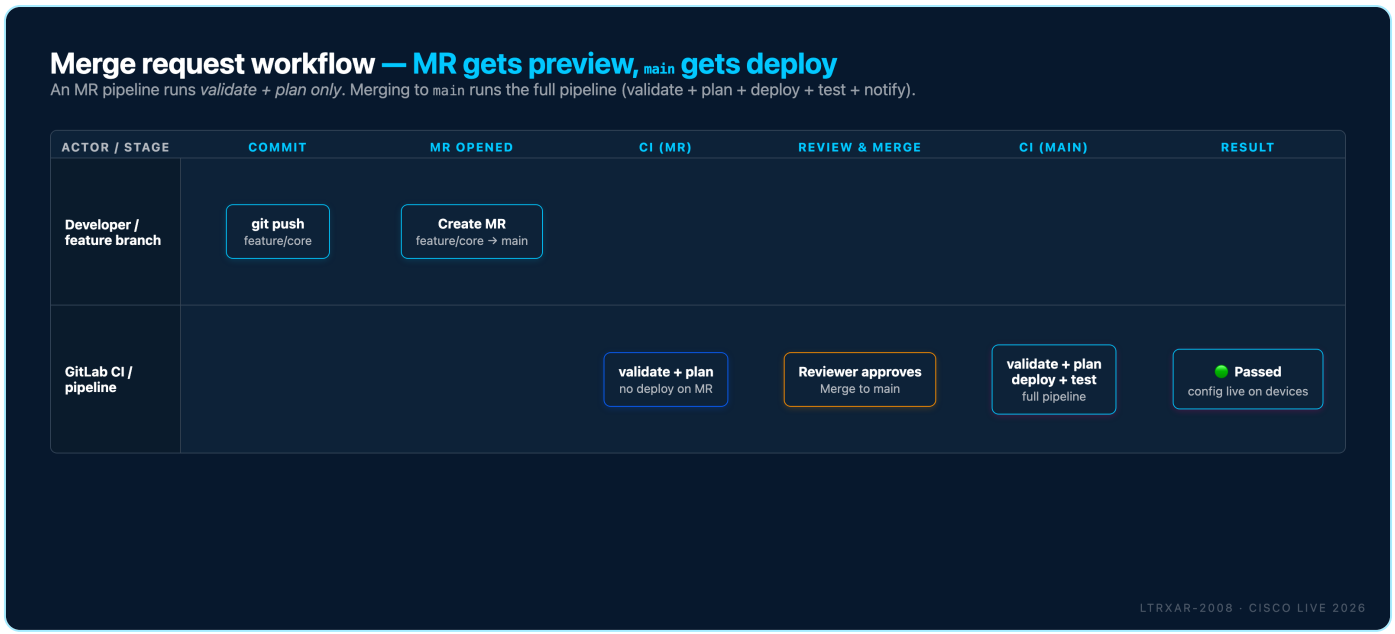
Lab exercise: complete branch and merge request workflow

In this exercise, you will:

1. Protect the `main` branch to require merge requests
2. Create a feature branch for your changes: `feature/core`
3. Add the **core** switch configuration with host IP host entries, and commit the changes to the `feature/core` branch
4. Create a merge request
5. Observe the first pipeline (validate + plan)
6. Approve and merge the request
7. Observe the second pipeline (validate + plan + deploy + test + notify)

Copyright © Cisco





> **Merge Request (MR) vs. Pull Request (PR)**
>
> Different version control platforms use different terminology. In GitLab, the term is **Merge Request (MR)**, while in GitHub, it's called a **Pull Request (PR)**. Both serve the same purpose: to propose changes from one branch to another and facilitate code review.

Step 1: Protect the main branch

First, you'll set up branch protection of the `main` branch. This means that you can no longer push changes directly to `main`; instead, all changes must go through a merge request (MR) process.

Copyright © Cisco

WHAT IS A PROTECTED BRANCH?

A **protected branch** is a branch with restrictions that prevent accidental or unauthorized changes. When you protect the `main` branch:

- **Direct commits are blocked**: No one can push changes directly to main
- **Force push is not allowed**: History cannot be rewritten
- **Merge requests are required**: All changes must go through the review/approval process

CONFIGURE MAIN BRANCH PROTECTION

Access GitLab

1. Open **Chrome** on the Windows 10 VM and navigate to GitLab: https://198.18.133.101
2. Log in with credentials: **Username:** `root` / **Password:** `C1sco12345`
3. Navigate to the **netascode/nac-iosxe-terraform** project.

Open Branch Protection Settings

1. In your project, click on **Settings** in the left sidebar
2. Select **Repository**
3. Scroll down to find the **Protected branches** section
4. Expand section to see the settings

Protect the Main Branch

1. In the **Protected branches** section, you should already see `main (default)` listed
2. Configure the following settings:

Setting	Value	Explanation
Allowed to merge	Maintainers	Only maintainers can merge approved changes
Allowed to push and merge	No one	Prevents direct commits to main

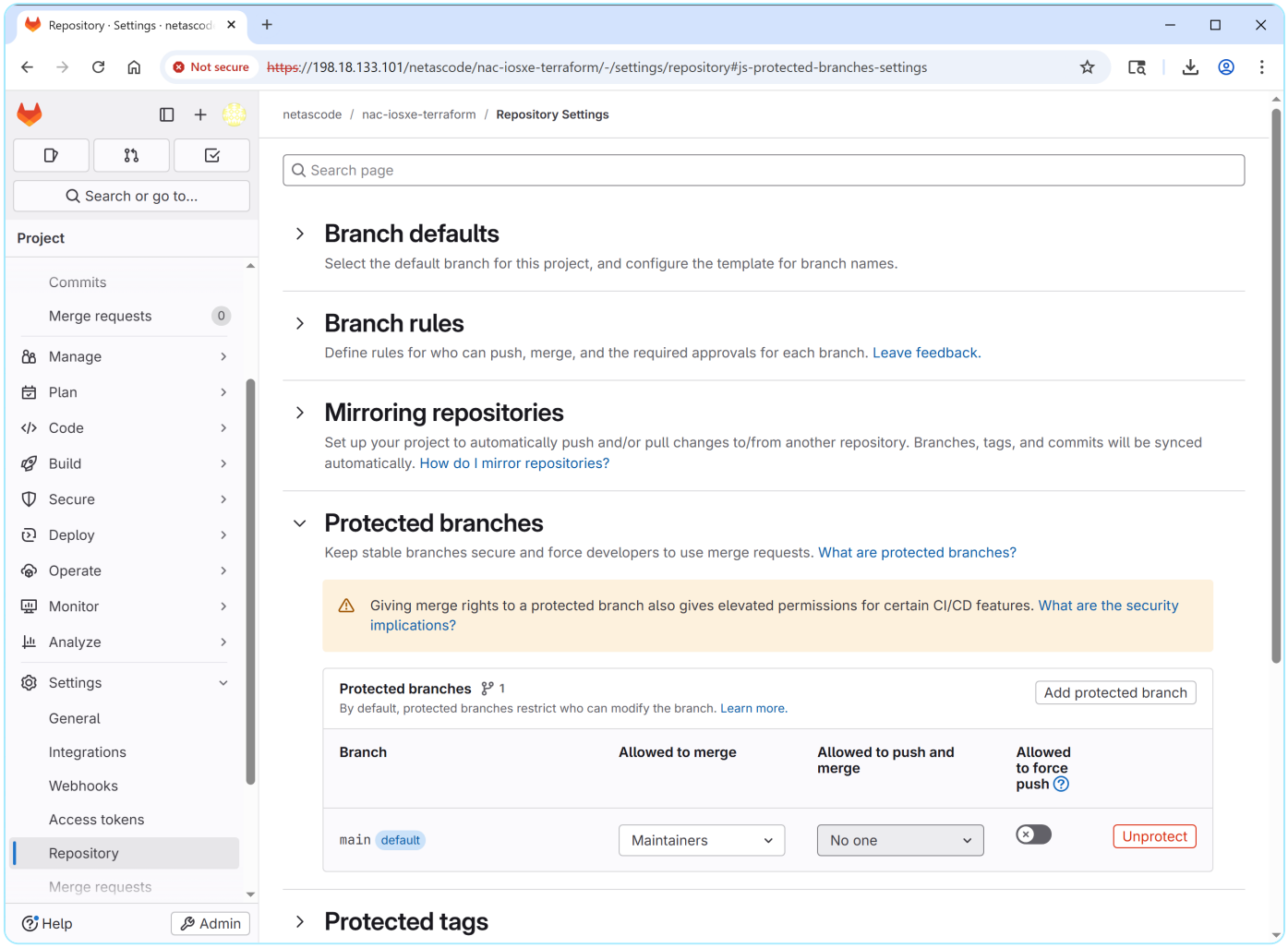
Why two separate knobs?

GitLab protects a branch along **two dimensions** - who can *merge an approved MR* and who can *push directly*. Those are different capabilities:

- **Merge** is the gated, reviewed path: an MR has to exist, pipelines have to pass, reviewers have to approve.
- **Push** is the ungated path: `git push origin main` bypasses the MR flow entirely.

A role can have one without the other. For a production repo you typically want **Maintainers: allowed to merge, No one: allowed to push** - maintainers can approve and click the button, but nobody (not even maintainers) can force-push around the review process. That's the configuration this task sets up. You'll see the effect in Step 2 when GitLab rejects your direct commit to main.

Copyright © Cisco



Default Branch Protection in GitLab

By default, GitLab protects the `main` branch, but as you already saw, direct pushes are still allowed from maintainers. In this lab, the user `root` is a maintainer, so you need to explicitly block direct pushes to enforce the MR workflow.

VERIFY BRANCH PROTECTION

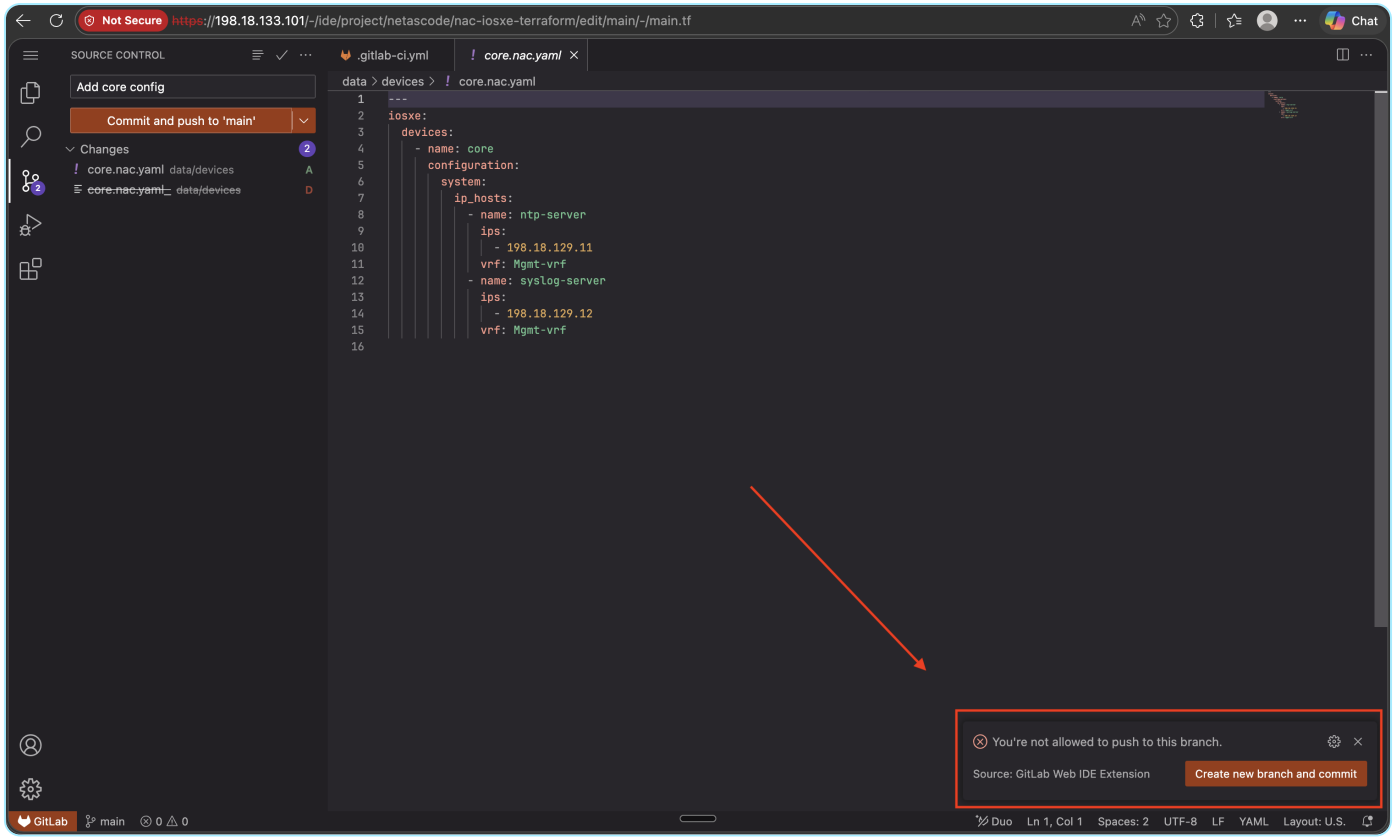
To confirm that the `main` branch is protected:

1. Open the **Web IDE** from the project page
2. Add the **core** device configuration: rename `data/devices/core.nac.yaml_` to `data/devices/core.nac.yaml` (remove the trailing underscore)
3. Click on the **Source Control** icon in the left sidebar
4. Enter a commit message:

```
Add core config
```

5. Attempt to commit the change directly to `main`: click **Commit and push to 'main'**

Copyright © Cisco



> **You can't push to the main branch**
>
> You should see an error message indicating that pushing to the protected branch is not allowed. This confirms that the `main` branch is successfully protected.

Step 2: Create a feature branch and commit changes

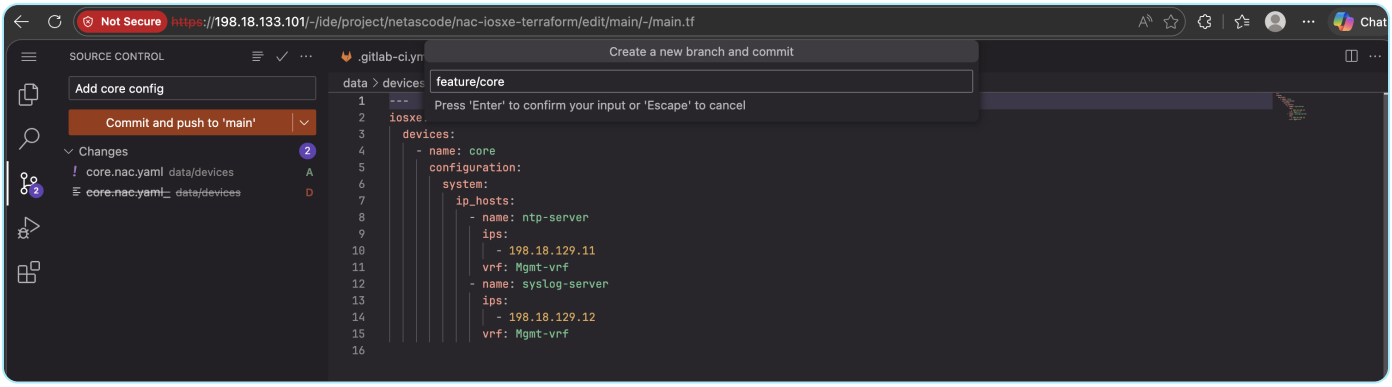
In the previous step, the Web IDE already prompted you to create a new branch when you tried to commit to `main`.

1. Press the **Create new branch** button
2. Enter the branch name:

```
feature/core
```

3. Press Enter to create the branch

Copyright © Cisco



Branch Created

You have successfully created a new branch named `feature/core`. This branch is now your isolated workspace to make changes without affecting the main production branch.

You can also see `feature/core` displayed in the bottom-left corner, indicating you're working on your new branch. The commit `Add core config` is already pushed to it.

Branch Naming Conventions

Use descriptive branch names that indicate the purpose of the changes:

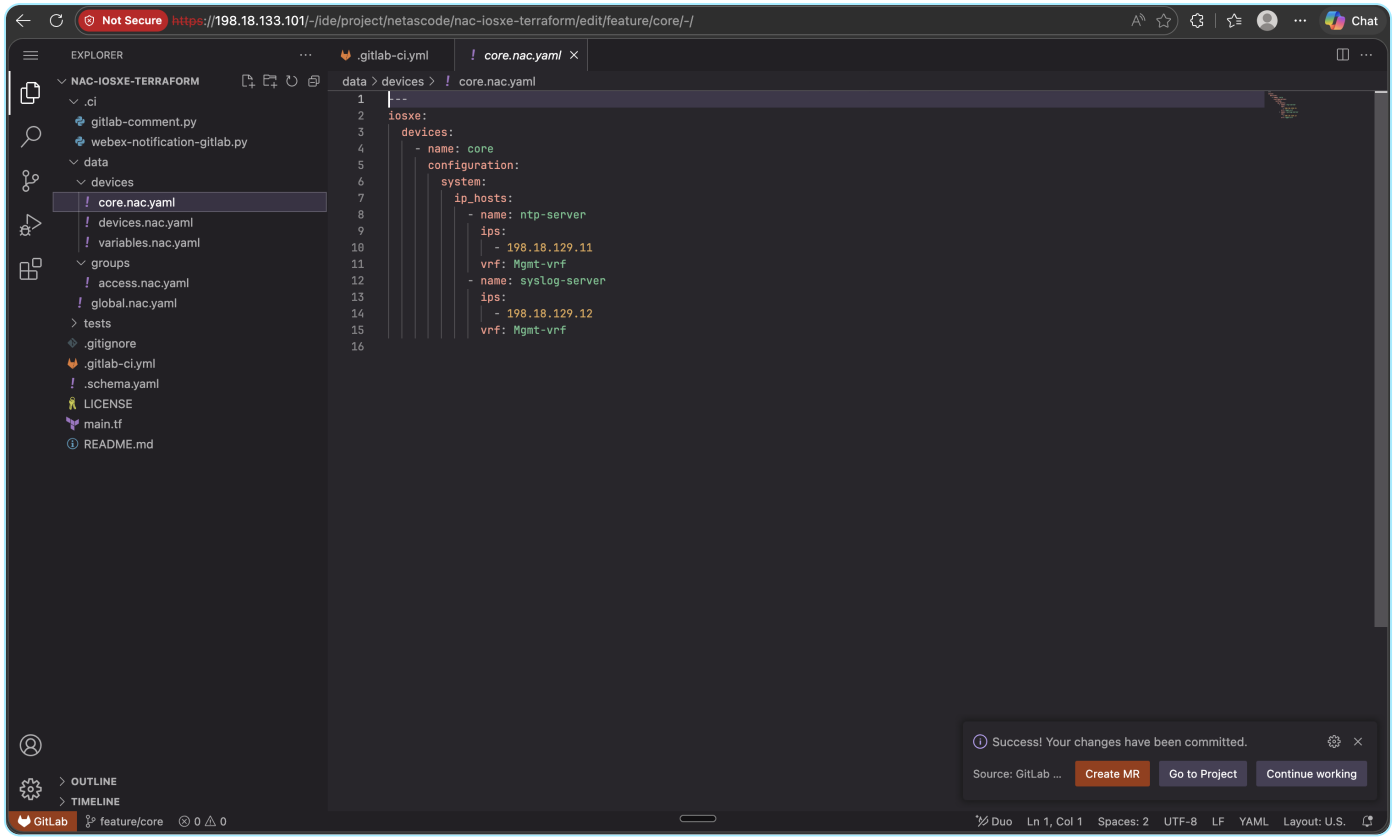
Example Branch Name	Purpose
`feature/add-vlan-100`	Adding a new feature
`fix/acl-typo`	Fixing a bug or error
`update/banner-message`	Updating existing configuration

This makes it easy to see what each branch contains without looking at the code.

Step 3: Create merge request

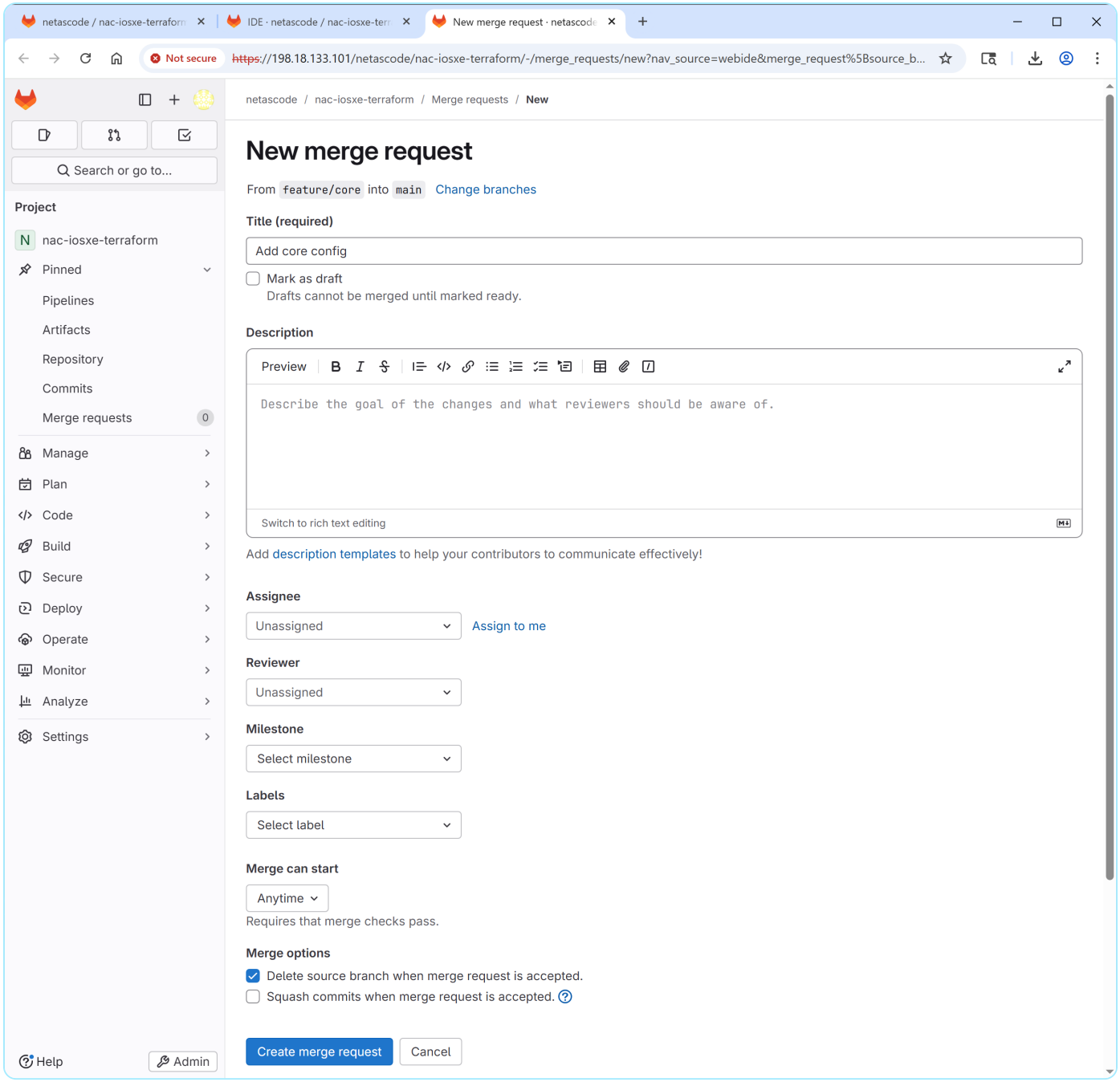
After committing to your feature branch, you need to create a merge request to propose merging your changes into main. Luckily, the GitLab Web IDE makes this very easy too: after pushing the commit, it shows a prompt to create a merge request.

Copyright © Cisco



Click the **Create merge request** button to proceed. It will take you to the merge request creation page.

Copyright © Cisco



You can leave the default values as they are for now. The source branch should be `feature/core`, and the target branch is `main`.

Click **Create merge request** at the bottom to finalize.

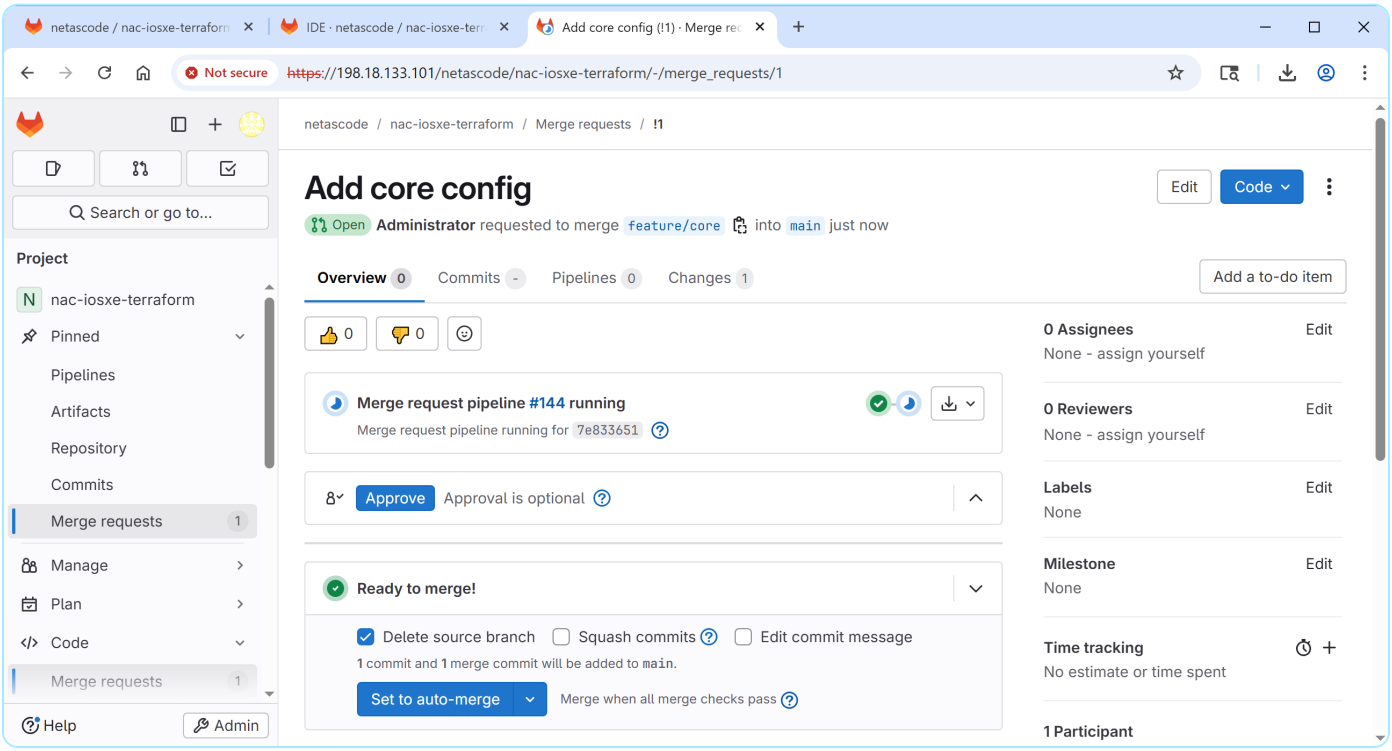
Copyright © Cisco

Alternatively, Create Merge Request From the GitLab UI

If you missed the prompt in the Web IDE, you can also create a merge request manually from the GitLab UI:

1. Go to your project in GitLab
2. Go to **Code** → **Merge requests** in the left sidebar
3. Click **New merge request**
4. Select the source branch (`feature/core`) and target branch (`main`)
5. Click **Compare branches and continue**

The merge request is now created! You should see the merge request page with details about the changes.



Step 4: Observe the first pipeline (validate + plan)

When you create the merge request, GitLab automatically triggers the same pipeline used in the previous tasks. However, since this is a merge request, the pipeline only runs the **validate** and **plan** stages - it does NOT deploy anything yet.

View the Pipeline Details

1. On the merge request page, click on the pipeline ID (#144 in the image above)
2. You'll see a pipeline running or completed

Copyright © Cisco

The pipeline uses the config from your feature branch (`feature/core`) and runs the following stages:

Stage	Job	Purpose	Typical duration
validate	`terraform fmt` , `nac-validate`	Check YAML syntax and schema compliance (See Task 10)	<10s
plan	`terraform plan`	Show what changes will be made to the network	~30-60s

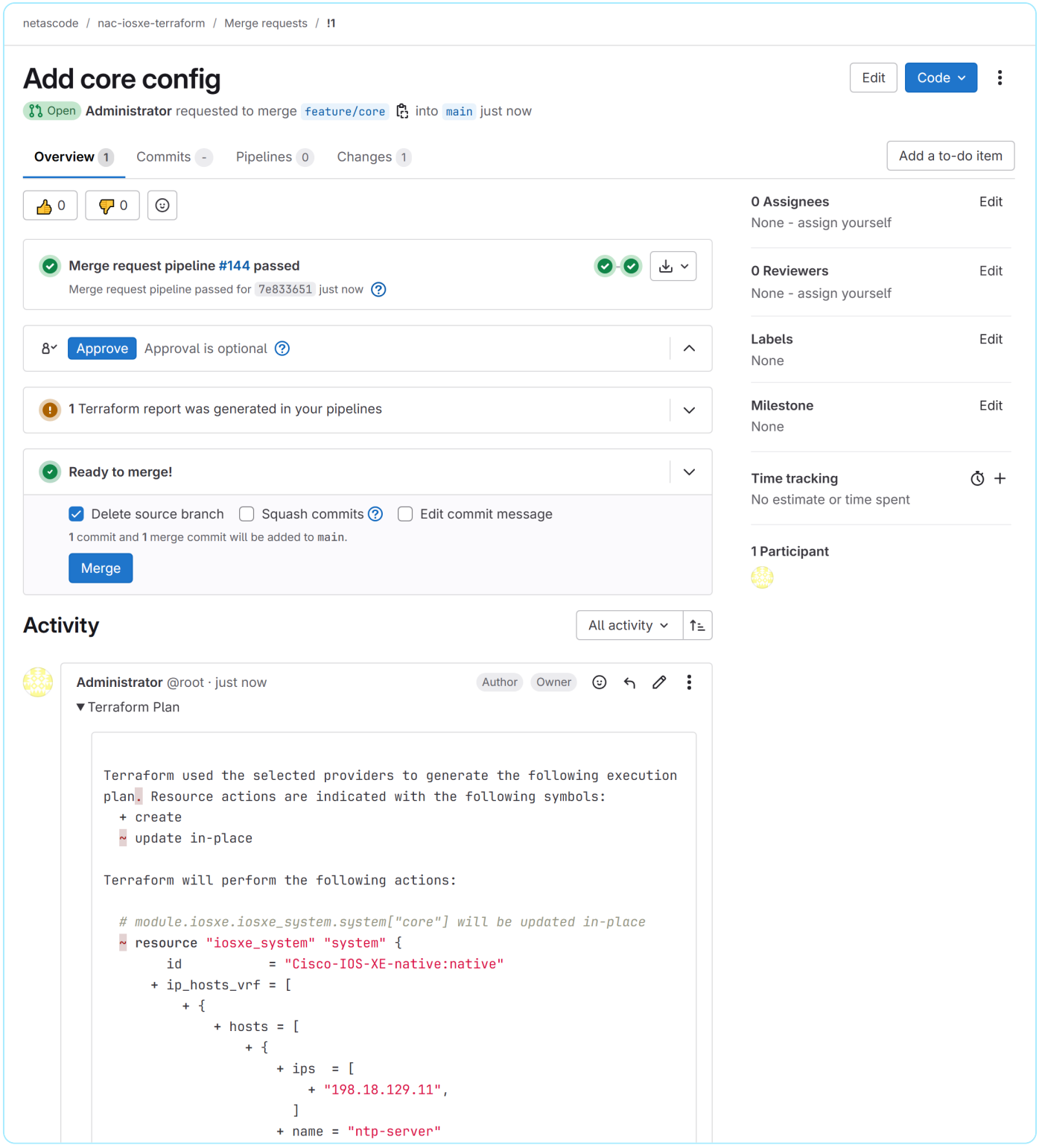
End-to-end the MR pipeline completes in about **90 seconds**. If it's still running at 3+ minutes, something's stuck - click into the running job to see where.

No Deploy Stage!

Notice that the **deploy** stage does NOT run on merge request pipelines. This is intentional - you want to see what will change without actually changing anything yet.

The **plan** stage also adds the terraform plan output as a comment to the merge request for easy review. Once the pipeline completes, you can expand the **Terraform plan** section in the comment under **Activity**.

Copyright © Cisco



Step 5: Review and approve the merge request

This is your opportunity to verify that the changes are correct before approving! In this case, you can see that the IP host entries will be added on the **core** switch.

In a real environment, a team lead or senior engineer would review the merge request. For this lab, you'll continue on your own.

Approval Step

Copyright © Cisco

In this lab, the approval step is not required to merge. In production environments, you would typically configure the project to require at least one approval from another team member.

> **Tip**
>
> You can also set up the project to require successful pipeline completion, and potentially perform other checks before allowing the merge.

Step 6: Merge to main

> **Verify Pipeline Passed**
>
> Before merging, ensure that the pipeline completed successfully, all stages passed, the config and the plan looks good.

You can now merge the changes into `main`.

1. On the merge request page, review the final diff one more time.
2. Review the merge options below the **Merge** button:
   - **Delete source branch** - removes the feature branch after merging. Leave **checked** to keep the repository tidy.
   - **Squash commits** - combines all commits from the feature branch into one before merging. Useful for noisy feature branches.
   - **Edit commit message** - lets you rewrite the merge commit message.
3. Click **Merge**.

> **Merged!**
>
> Your changes are now on `main`. Because `main` is the deployed branch, this automatically triggers the full deployment pipeline (validate + plan + deploy + test + notify).

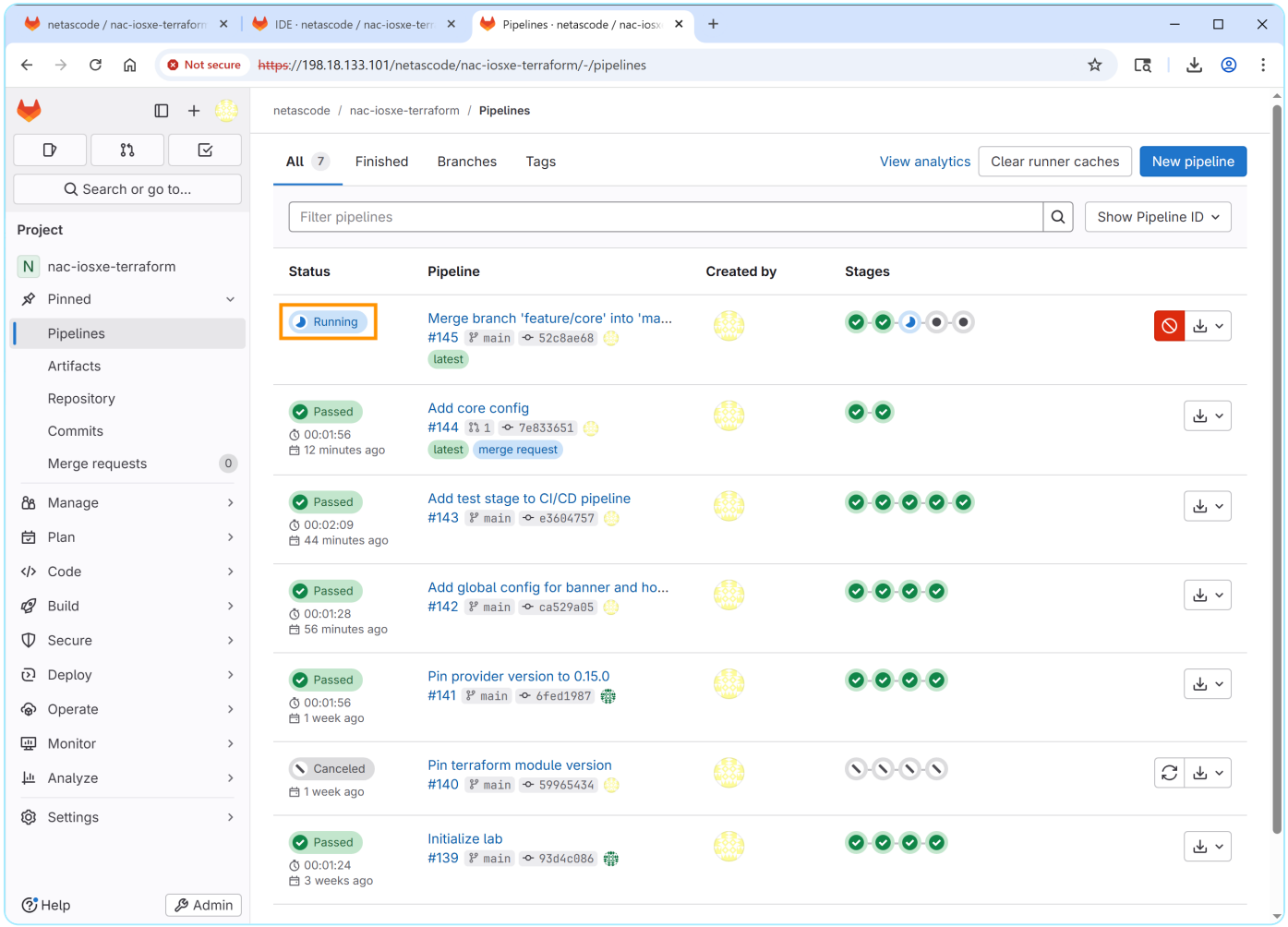
Step 7: Observe the deployment pipeline

Merging to main triggers a new pipeline - this time including the deployment and test stages. This pipeline is identical to the one run in Tasks 13 and 14.

View the Deployment Pipeline

1. Go to **Build** → **Pipelines** in the left sidebar
2. You'll see a new pipeline that was triggered by the merge
3. Click on it to view the stages

Copyright © Cisco



The main branch pipeline runs ALL stages:

- **validate** - <10s
- **plan** - ~30-60s
- **deploy** - ~60-90s
- **test** - ~45-60s
- **notify** - <5s

Total: about **3-4 minutes** end-to-end. If it stalls in `deploy` or `test` past 4 minutes, click into the job logs - it's usually a device that went unreachable or a test that timed out on operational state (see the troubleshooting section below).

After the pipeline completes successfully, verify the changes landed on the devices:

CONFIRM THE IP HOST ENTRIES ON `CORE`

Open **Solar-PuTTY** and connect to `core` (198.18.130.10). Run:

`DEVICE CLI · PuTTY` **core**

```
show run | include ip host
```


Copyright © Cisco

Expected output:

OUTPUT **Expected output**

```
core#show run | include ip host
ip host host01 192.168.100.100
ip host host02 192.168.100.200
```

Two `ip host` lines, one for each host. If you see fewer (or none), scroll to the Troubleshooting section below.

CONFIRM THE DEPLOY LOG, TOO

As in Task 13, click into the **deploy** job on the merge pipeline and look for:

OUTPUT **deploy job tail**

```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

The `Resources: N added/changed` line is the authoritative record of what Terraform actually pushed - useful for audit purposes once you take this pattern to production.

Troubleshooting - common issues

You cannot push to this branch

This means the branch protection is working! Create a feature branch instead of trying to push to main directly.

Validation failed

Check the job logs for `nac-validate` to see what schema or syntax errors occurred. Fix them in your feature branch and push again.

For more details, refer to Task 10 - Schema Validation.

Merge conflicts

If there are merge conflicts, GitLab will notify you on the MR page. To resolve:

1. Fetch the latest main branch into your feature branch
2. Resolve conflicts locally or in the Web IDE
3. Commit and push the resolved changes to your feature branch
4. The MR will update automatically

Copyright © Cisco

Terraform plan or apply failed

Check the job logs for `terraform plan` or `terraform apply` to see what went wrong. Common issues include:

- Connectivity issues to devices
- Authentication failures
- Configuration errors

If you see "Failed to install provider", re-run the job.

Pipeline failures

This can happen for various reasons. Common steps to resolve:

1. Check the job logs for specific errors
2. Fix the issues in your feature branch
3. Commit and push again to your feature branch - the MR pipeline will re-run automatically

If you need help, feel free to ask your instructors!

What you've accomplished

In this task, you have:

- Configured the main branch as **protected** to enforce change control
- Created a **feature branch** for your configuration changes
- Modified configuration safely in your branch
- Created a **merge request** and observed the preview pipeline (validate + plan)
- Experienced the **review and approval** process
- Merged changes and observed the deployment pipeline (validate + plan + deploy + test + notify)
- Learned the complete **end-to-end workflow** used in production environments

Task 14 - Extend CI/CD pipeline

Conclusion

Copyright © Cisco

Conclusion

You've reached the end of the **Network as Code for IOS XE** lab - well done.

What you've accomplished

Core skills

- **YAML configuration** - wrote declarative network configurations in human-readable YAML
- **Terraform workflow** - used `terraform init`, `plan`, `apply`, and `destroy` end-to-end
- **Configuration hierarchy** - applied settings at global, device-group, and device levels with proper precedence
- **Schema validation** - caught errors pre-deployment with `nac-validate`
- **CI/CD automation** - ran GitLab pipelines to validate, plan, and deploy changes

Configuration patterns practiced

- **Global** - login banners, system hostnames applied everywhere
- **Device group** - ACLs applied to access switches only
- **Device-specific** - per-device settings (loopbacks, BGP, host-specific ACLs)
- **Variables** - reusable values across the hierarchy

Optional skills (if you reached them)

- Template types: `model`, `file`, and `cli`
- Post-change validation with `nac-test` and Robot Framework
- Pipeline extensions: integration tests + idempotency checks
- GitOps guardrails: branch protection and merge-request workflows

Key takeaways

1. **Declarative over imperative** - define *what* you want; let the module figure out *how*.
2. **Version-control everything** - every change has an author, a reviewer, and a rollback path.
3. **Automate the workflow** - pipelines enforce consistency and make failure visible.
4. **Validate before deploy** - `nac-validate` catches syntax and schema errors before they reach a device.
5. **Test after deploy** - `nac-test` confirms intent actually landed on the wire.

Copyright © Cisco

How this scales from four devices to a hundred (or a thousand)

The patterns you just used were chosen because they scale. A fleet of 100 branches doesn't change the workflow; it just changes the input:

- **The YAML shape is identical.** One `devices/*.nac.yaml` per device regardless of fleet size. Adding branch 101 is creating one file, not restructuring anything. Shared configuration (a new NTP server, a security policy update) still lives in exactly one file (`global/` or a `groups/` file) and propagates via the same merge precedence.
- **Exceptions stay local.** If branch 47 needs a one-off setting that differs from every other branch, it goes in `devices/branch-47.nac.yaml` and overrides the global default for that one device. Nothing else in the project moves. The merge model is what makes "the exception that proves the rule" a one-file change instead of a refactor.
- **Blast radius is controllable.** `managed_devices = ["branch-47"]` (Task 12) or the equivalent `IOSXE_SELECTED_DEVICES` env var scopes a `terraform apply` to a named subset. Combined with CI pipelines gating on code review (Task 15), you can stage rollouts - apply to three canary branches first, observe, then expand. The 4-device lab version of this is running `terraform apply` against all 4; the 100-device production version is the same command with a subset filter.
- **State is the authoritative record.** One `terraform.tfstate` (or one pipeline-managed state backend, for a team) captures what each of the 100 devices currently has. Drift detection, audit trails, and "what did we actually change last Tuesday" answers all come from that state plus git history.

What you wrote for 4 devices is the same YAML you'd write for 400. The module, the merge semantics, the CI pipeline shape, and the verification loop stay identical; the `data/devices/` directory just gets bigger.

Try at home

A few small extensions make great follow-ups once you're back at your own environment:

- **Default gateway for host networks** - extend `devices/border.nac.yaml` to add a `GigabitEthernet3` interface with IP `192.168.100.1/24` so the host networks can route to the ISP. Validate by pinging `8.8.8.8` from `host01` / `host02` via the CML console. (This is also the "Advanced Challenge" in Task 08 if you finished early during the session.)
- **More variables, more precedence** - introduce a `${DNS_DOMAIN}` or `${MGMT_VLAN}` variable at the global level and override it on one device to see precedence in action.
- **Your own validation rules** - `nac-validate` is extensible. Write a rule that enforces a naming convention or flags a forbidden configuration (for example, any ACL entry with `action: permit` on `any`).
- **Verify via data sources, not SSH** - the `terraform-provider-iosxe` ships ~120 data sources that read operational state. Instead of SSHing in and running `show ip bgp summary`, define a `data "iosxe_bgp_neighbor"` resource and assert the state in a Terraform `check` or `output`. Same answer, fully programmatic - useful for letting another part of your IaC codebase condition on device state.
- **Brownfield import with `nac-tool`** - point `nac-tool` at one of your existing IOS XE devices at home and let it generate the equivalent NaC YAML. This is the fastest way to see how much of your current running-config maps cleanly to NaC's data model (and where you'd need to fall back to `cli` templates).

Continue your journey

Core documentation and code

- Network as Code documentation - the authoritative docs site; start with *Guides* → *Concepts* if you want to go deeper on merge semantics, defaults precedence, and variable scope resolution
- `terraform-iosxe-nac-iosxe` - the module used in this lab (Apache 2.0). File issues via its GitHub Issues page.
- `terraform-provider-iosxe` - the IOS XE Terraform provider. Full resource and data-source reference at registry.terraform.io/providers/CiscoDevNet/iosxe.
- NetAsCode on GitHub - the full org: modules, tools, and data models for every supported Cisco platform (IOS XE, SD-WAN, ACI, Catalyst Center, Nexus Dashboard, Meraki, etc.)

Sibling tools worth knowing about

Copyright © Cisco

This lab uses `nac-validate` (Task 10) and `nac-test` (Task 11). The NaC toolchain ships a few more:

- `nac-tool` – **brownfield import.** Point it at a running device and it writes out the equivalent NaC YAML. This is the bridge between "existing network" and "Network as Code adoption" – no need to start from scratch.
- `nac-api` – REST API over NaC configs. Useful for custom integrations.
- `nac-collector` – gather deployed configs from the field into a central store.

Related Cisco Live sessions

- **BRKXAR-2032** – NetDevOps testing fundamentals with Generative AI
- **LTRDCN-3439** – Nexus Dashboard VXLAN as Code
- **LTRATO-2223** – FastForward SD-WAN deployment with SDWAN as Code
- **LTRENS-3751** – SD-Access as Code with Catalyst Center and ISE
- **BRKENT-2115** – Automate Catalyst SD-WAN with Network as Code

Share your feedback

Your feedback drives next year's version of this lab. Two ways to contribute:

- **Cisco Live session survey** – rate LTRXAR-2008 in the Cisco Live app or portal. These ratings directly influence whether the session runs again, and at which events.
- **Direct feedback** – if you hit something confusing, broken, or wished-for-but-missing, email either presenter (below). Specific > general; "the Task 08 BGP verification was unclear because X" is 10x more useful than "Task 08 was confusing."

Thank you

Thanks for spending four hours with us. If you have questions or feedback after the session, please reach out:

- **Andrea Testino** – atestini@cisco.com – LinkedIn
- **Christopher Hart** – chart2@cisco.com – LinkedIn

Task 15 – Merge request workflow

Copyright © Cisco

3. Glossary

Acronyms and terms used throughout this lab guide. Bookmark or keep this tab open - new terms appear across almost every task.

Term	What it means
NaC	Network as Code - Cisco's Terraform-based declarative automation framework. This lab is one flavor of it (IOS XE); sibling labs cover SD-WAN, ACI, Catalyst Center, Nexus Dashboard, etc.
IaC	Infrastructure as Code - the broader category of "manage infrastructure by editing files, not clicking buttons". Network as Code is IaC for networking.
YAML	A human-readable data format (indentation-based). All your Network as Code "intent" files are YAML.
YANG	A standardized modelling language for network device configuration and state. IOS XE's configuration surface is described in YANG modules; Network as Code ultimately produces YANG data to push to devices.
NETCONF	A standards-based network management protocol (RFC 6241). Runs over SSH on TCP/830. Uses YANG models and a candidate/running datastore pair - this lab uses it for all configuration push.
RESTCONF	REST-style alternative to NETCONF (HTTPS-based). Not used in this lab but supported by the same Network as Code module and IOS XE provider.
Terraform	HashiCorp's declarative infrastructure engine. Reads `.tf` files + state, produces a plan, applies it. This lab uses Terraform as the orchestrator under Network as Code.
HCL	HashiCorp Configuration Language - the syntax `.tf` files are written in (braces, key = value). You'll write very little of this directly; the Network as Code module abstracts it.
CML	Cisco Modeling Labs - the virtualized network simulation platform that hosts the lab's IOS XE devices.
BGP	Border Gateway Protocol - used in Task 08's optional BGP-over-ISP example.
ACL	Access Control List - used in Task 04's device-group example.
CI/CD	Continuous Integration / Continuous Deployment - the automation that runs your code/config through build/test/deploy stages on every change. This lab uses GitLab CI.
GitOps	The practice of using Git as the source of truth for infrastructure state, with CI/CD as the delivery mechanism. Tasks 13-15 demonstrate it.
MR / PR	Merge Request (GitLab) / Pull Request (GitHub) - the code-review workflow for proposing a change to a branch. Task 15 uses MRs.
WSL	Windows Subsystem for Linux - the Linux environment running inside the lab's Win10 VM, where you'll run Terraform.
Declarative vs imperative	Declarative: describe *what* you want (e.g. "VLAN 10 exists with name DATA"). Imperative: describe *the steps* (`vlan 10` , `name DATA`). Network as Code is declarative - you describe state, the module figures out the steps.
`candidate` / `running`	NETCONF's two main datastores. You write to `candidate` (scratch), issue `<commit>` to atomically swap it into `running` (live config). See Task 01's NETCONF diagram.
dCloud	Cisco's demo / lab cloud environment - where this lab is hosted during Cisco Live.

Copyright © Cisco

For deeper conceptual explanations of any of these, see:

- Intro 01 - Learning Objectives - the "what and why" of Network as Code
- Task 01 - SSH to devices - NETCONF, datastores, and the transactional model
- Task 03 - Global configuration - Terraform's `init` / `plan` / `apply` workflow
- Task 13 - Run a CI/CD pipeline - CI/CD and GitOps in practice

Copyright © Cisco

4. Lab topologies

Lab devices

The following devices are available in the Cisco Modeling Labs (CML) topology.

Device	IP Address	Username	Password
core	198.18.130.10	cisco	cisco
access01	198.18.130.11	cisco	cisco
access02	198.18.130.12	cisco	cisco
border	198.18.130.20	cisco	cisco
isp	198.18.130.200	cisco	cisco
host01	192.168.100.100	cisco	cisco
host02	192.168.100.200	cisco	cisco
ntp-server	198.18.129.11	cisco	cisco
syslog-server	198.18.129.12	cisco	cisco

SSH from WSL Terminal

To connect via SSH from your WSL terminal, use:

```
ssh cisco@<IP_ADDRESS>
```

For example: `ssh cisco@198.18.130.10`

Host Device Connectivity

The host devices (**host01** and **host02**) are not reachable from your lab Win10 VM or Ubuntu VM. They are only reachable from CML - see below.

Lab VMs

Device	IP Address	Username	Password	Interface
Windows 10 VM	198.18.133.20	admin	cisco	rdp
CML	198.18.130.34	guest	CiscoLive	web
Ubuntu VM	198.18.133.101	guest	CiscoLive	ssh
GitLab	198.18.133.101	root	C1sco12345	web

Copyright © Cisco

Ubuntu VM and GitLab share a host

The Ubuntu VM and GitLab entries above both show `198.18.133.101` - that's not a typo. They're the same underlying VM, accessed as two different services:

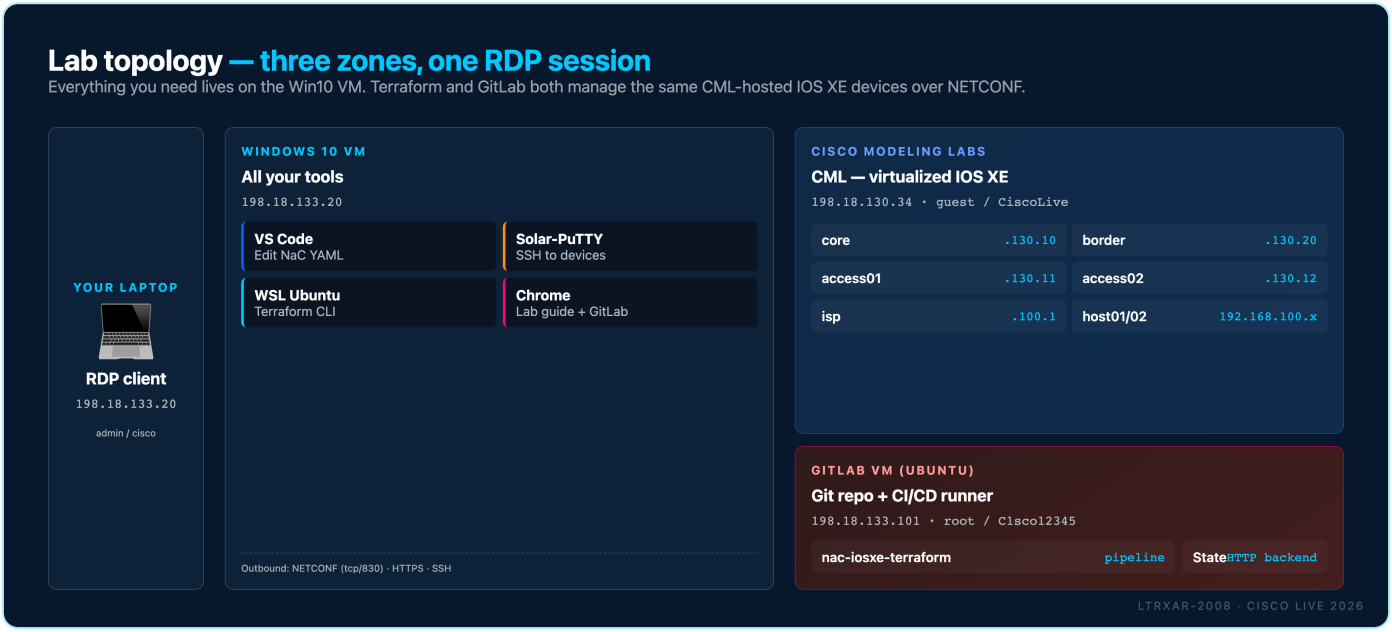
- **SSH as** `guest` → you get an Ubuntu shell
- **HTTPS as** `root` → you get the GitLab web UI

GitLab runs as a containerized application on top of the Ubuntu host. You won't SSH into this VM during the lab (the pipeline work happens in the GitLab UI), but it's useful context if you ever see the shared address and wonder which credentials to use.

Self-signed Certificates

The CML and GitLab web interfaces use self-signed SSL certificates. Your browser may show a warning when accessing these sites. You can safely ignore the warning and proceed to the site.

Below is the dCloud lab topology diagram:

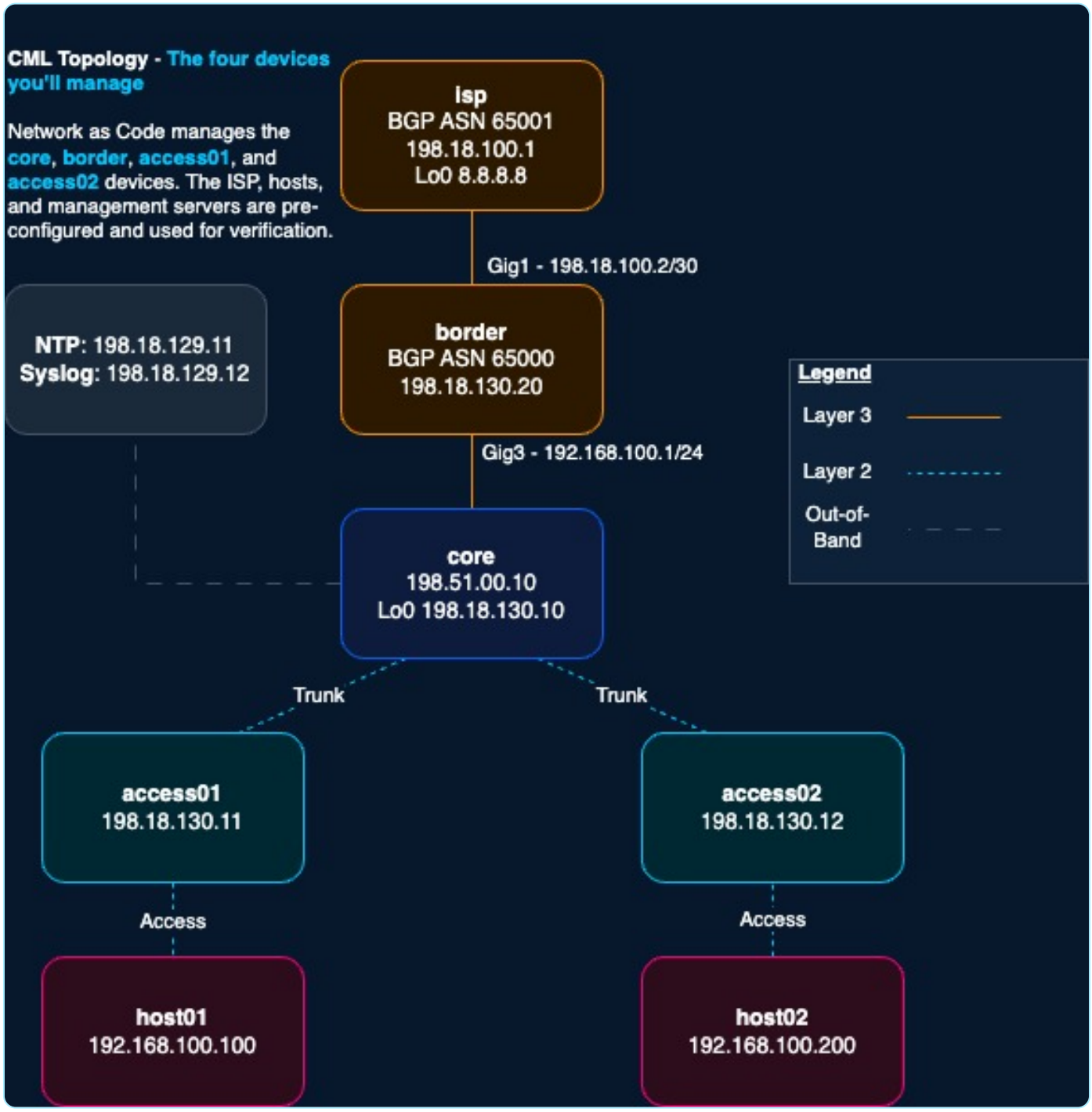


This lab consists of:

- Multiple **IOS XE** virtual switches running in CML (Cisco Modeling Labs - network simulation platform)
- **GitLab** as Git repository and to run CI/CD pipelines, running in an Ubuntu VM
- Windows 10 VM with:
  - **VS Code** for editing Infrastructure as Code YAML files
  - **Solar-PuTTY** SSH client to access the IOS XE devices
  - **Windows Subsystem for Linux (WSL)** to run Terraform, and other command-line tools
  - **Google Chrome** Web Browser to read this lab guide and access GitLab

Copyright © Cisco

The topology below shows the IOS XE devices running in **Cisco Modeling Labs (CML)**. CML is a network simulation platform that allows you to create virtual network environments using real Cisco operating systems like IOS XE. In this lab, CML hosts three Catalyst 9000v virtual switches (**access01**, **access02**, and **core**) and a Catalyst 8000v virtual router (**border**) that you'll configure using Network as Code:



The **isp** device simulates an external Internet Service Provider connection for BGP routing. It has a loopback interface configured with IP address `8.8.8.8`.

The host devices (**host01** and **host02**) are ubuntu-based simulated end-user devices connected to the network. They are configured with static IP addresses and default gateway `192.168.100.1`.

Copyright © Cisco

Additional Servers

The lab topology also includes **ntp-server** and **syslog-server** devices. Similarly to the host devices, these are empty Ubuntu VMs, only intended to perform reachability tests from the IOS XE devices.

Copyright © Cisco

5. Appendices

APPENDIX I

GitLab CI/CD pipeline definition file

Here is the final `.gitlab-ci.yml` file including the test stage from Task 14 - Edit CI/CD:

FILE **.gitlab-ci.yml**

```
---
image: danischm/nac:0.1.6
stages:
  - validate
  - plan
  - deploy
  - test
  - notify

variables:
  # Disable SSL verification for git operations
  GIT_SSL_NO_VERIFY: "true"

  IOSXE_USERNAME:
    description: "Cisco IOS XE Username"
  IOSXE_PASSWORD:
    description: "Cisco IOS XE Password"
  IOSXE_PROTOCOL:
    description: "Cisco IOS XE connection protocol"
    value: "netconf"
  GITLAB_TOKEN:
    description: "User Access Token. Used to create comments on Merge Requests"
  TF_HTTP_USERNAME:
    description: "GitLab Username"
    value: "gitlab-ci-token"
  TF_HTTP_PASSWORD:
    description: "GitLab Access Token"
    value: "${CI_JOB_TOKEN}"
  WEBEX_ROOM_ID:
    description: "Cisco Webex Room ID"
  WEBEX_TOKEN:
    description: "Cisco Webex Bot Token"
  GITLAB_API_URL:
    description: "GitLab API v4 root URL"
    value: "${CI_API_V4_URL}"
  TF_HTTP_ADDRESS:
    description: "GitLab HTTP Address to store the TF state file"
    value: "${GITLAB_API_URL}/projects/${CI_PROJECT_ID}/terraform/state/tfstate"
  TF_HTTP_LOCK_ADDRESS:
    description: "GitLab HTTP Address to lock the TF state file"
    value: ${TF_HTTP_ADDRESS}/lock
  TF_HTTP_LOCK_METHOD:
    description: "Method to lock TF state file"
    value: POST
  TF_HTTP_UNLOCK_ADDRESS:
    description: "GitLab HTTP Address to unlock the TF state file"
    value: ${TF_HTTP_ADDRESS}/lock
  TF_HTTP_UNLOCK_METHOD:
    description: "Method to unlock TF state file"
    value: DELETE

cache:
  key: terraform_modules_and_lock
  paths:
    - .terraform
    - .terraform.lock.hcl
    - defaults.yaml
    - model.yaml

validate:
  stage: validate
  script:
    - set -o pipefail && terraform fmt -check |& tee fmt_output.txt
    - set -o pipefail && nac-validate ./data/ |& tee validate_output.txt
  artifacts:
    paths:
      - fmt_output.txt
      - validate_output.txt
  cache: []
  rules:
    - if: $CI_COMMIT_TAG == null
```


Copyright © Cisco

```
plan:
  stage: plan
  resource_group: iosxe
  script:
    - terraform init -upgrade -input=false
    - terraform plan -out=plan.tfplan -input=false
    - terraform show -no-color plan.tfplan > plan.txt
    - terraform show -json plan.tfplan | jq > plan.json
    - terraform show -json plan.tfplan | jq '([.resource_changes[]?.change.actions?]|flatten)|{"create":(map(select(.=="create"))|length),"update":
(map(select(.=="update"))|length),"delete":(map(select(.=="delete"))|length)}' > plan_gitlab.json
    - python3 .ci/gitlab-comment.py
  artifacts:
    paths:
      - plan.json
      - plan.txt
      - plan.tfplan
      - plan_gitlab.json
    reports:
      terraform: plan_gitlab.json
  dependencies: []
  needs:
    - validate
  only:
    - merge_requests
    - main

deploy:
  stage: deploy
  resource_group: iosxe
  script:
    - terraform init -input=false
    - terraform apply -input=false -auto-approve plan.tfplan
  artifacts:
    paths:
      - defaults.yaml
  dependencies:
    - plan
  needs:
    - plan
  only:
    - main

test-integration:
  stage: test
  script:
    - echo "=== PWD and tree ==="; pwd; ls -la
    - echo "=== Inputs nac-test needs ==="; ls -la model.yaml defaults.yaml || true
    - echo "=== tests/ scaffolding ==="; ls -la tests/ tests/templates tests/filters || true
    - echo "=== nac-test version ==="; nac-test --version || true
    - mkdir -p tests/results
    -
set -o pipefail && nac-test --data ./model.yaml --data ./defaults.yaml --templates ./tests/templates --filters ./tests/filters --output ./tests/results |&
tee test_output.txt
  after_script:
    - echo "=== tests/results contents (post-run) ==="; ls -la tests/results || true
    - echo "=== last 50 lines of test_output.txt ==="; tail -n 50 test_output.txt || true
  artifacts:
    when: always
    paths:
      - tests/results/
      - test_output.txt
    reports:
      junit: tests/results/xunit.xml
  dependencies:
    - deploy
  needs:
    - deploy
  only:
    - main

test-idempotency:
  stage: test
  resource_group: iosxe
  script:
    - terraform init -input=false
    - terraform plan -input=false -detailed-exitcode
  dependencies:
    - deploy
  needs:
    - deploy
  only:
    - main

failure:
  stage: notify
  script:
    - python3 .ci/webex-notification-gitlab.py -f
  when: on_failure
  artifacts:
    when: always
    paths:
      - tests/results/*.html
      - tests/results/xunit.xml
```


Copyright © Cisco

```
      - plan.txt
      - fmt_output.txt
      - validate_output.txt
      - test_output.txt
  cache: []

success:
  stage: notify
  script:
    - python3 .ci/webex-notification-gitlab.py -s
  when: on_success
  artifacts:
    when: always
    paths:
      - tests/results/*.html
      - tests/results/xunit.xml
      - plan.txt
      - fmt_output.txt
      - validate_output.txt
      - test_output.txt
  cache: []
```

Conclusion

Appendix II - Schema reference

Copyright © Cisco

APPENDIX II

Schema reference (lab features only)

This is a ~15% slice of the full schema

The full IOS XE as Code data model schema covers ~249 top-level configuration keys. This appendix includes only the ~38 keys the lab actually uses (global, devices, device_groups, templates, banner, access_lists, ip_hosts, loopbacks, vlan, bgp, system, aliases). For the complete data model - including MACsec, DMVPN, QoS, HSRP/VRRP, MPLS, IS-IS, EIGRP, multicast, AAA/TACACS/RADIUS, STP, 802.1X, SNMP, EEM, segment routing, EVPN, and more - browse netascode.cisco.com/docs/data_models/iosxe/overview/.

How to read this schema (Yamale syntax reference)

The schema is written in Yamale, a YAML-based schema validation format. Quick reference for the syntax you'll see below:

Syntax	Meaning
`str()`	Any string.
`int()`	Any integer.
`int(min=1, max=4094)`	Integer with bounds (e.g. VLAN IDs).
`bool()`	`true` or `false`.
`enum('permit', 'deny')`	One of the listed values - nothing else.
`regex('^[A-Z]+$')`	String matching the given regex.
`ip()`	A valid IP address.
`list(include('X'))`	A list of objects conforming to the `X` sub-schema.
`include('X')`	A single object conforming to the `X` sub-schema.
`map(key=str())`	Key/value map with string keys.
`required=False`	Field is optional (defaults to required otherwise).
`any(...)`	Value can match any of several type constructors.

So a line like `login: str(required=False)` reads as "the `login` field is an optional string", and `sequence: int(min=1, max=2147483647)` reads as "`sequence` is a required integer between 1 and 2^31-1".

Copy this single schema block for reference. It includes only the features used in this lab:

FILE **schema.yaml**

```
---
iosxe: include('iosxe', required=False)
---

# Root structure
iosxe:
  global: include('global', required=False)
  devices: list(include('devices'), required=False)
  device_groups: list(include('device_groups'), required=False)
  templates: list(include('iosxe_templates'), required=False)
```


Copyright © Cisco

```
# Global configuration (Task03: Banner, Task06: Variables)
global:
  variables: map(key=str(), required=False)
  configuration: include('device_configuration', required=False)
  templates: list(str(), required=False)

# Device definitions (Task03, Task05, Task06, Task08)
devices:
  name: str()
  host: str(required=False)
  configuration: include('device_configuration', required=False)
  templates: list(str(), required=False)
  variables: map(key=str(), required=False)

# Device groups (Task04: ACL)
device_groups:
  name: str()
  devices: list(str(), required=False)
  configuration: include('device_configuration', required=False)
  templates: list(str(), required=False)

# Template definitions (Task07, Task08, Task09)
iosxe_templates:
  name: str()
  type: enum('model', 'file', 'cli')
  file: str(required=False)
  content: str(required=False)
  configuration: include('device_configuration', required=False)

# Device configuration block (used by global, devices, device_groups, templates)
device_configuration:
  banner: include('banner', required=False)
  access_lists: include('access_lists', required=False)
  system: include('system', required=False)
  vlan: include('vlan', required=False)
  routing: include('routing', required=False)
  interfaces: include('interfaces', required=False)

# Banner (Task03)
banner:
  exec: str(required=False)
  login: str(required=False)
  motd: str(required=False)

# Access Lists (Task04)
access_lists:
  standard: list(include('access_lists_standard'), required=False)

access_lists_standard:
  name: str()
  entries: list(include('access_lists_standard_entries'), required=False)

access_lists_standard_entries:
  sequence: int()
  action: enum('deny', 'permit')
  prefix: ip(required=False)
  prefix_mask: str(required=False)
  any: bool(required=False)
  host: ip(required=False)

# System - IP Hosts (Task05), Hostname (Task06)
system:
  hostname: regex('^[^\s]*$', required=False)
  ip_hosts: list(include('system_ip_hosts'), required=False)

system_ip_hosts:
  name: str()
  ips: list(ip())
  vrf: str(required=False)

# VLAN (Task07)
vlan:
  vlans: list(include('vlan_vlans'), required=False)

vlan_vlans:
  id: int(min=1, max=4094)
  name: str(required=False)

# Routing - BGP (Task08)
routing:
  bgp: include('routing_bgp', required=False)

routing_bgp:
  as_number: any(int(), str())
  router_id_interface_type: enum('Loopback', required=False)
  router_id_interface_id: any(int(), str(), required=False)
  neighbors: list(include('bgp_neighbors'), required=False)

bgp_neighbors:
  ip: ip()
  remote_as: any(int(), str())
  description: str(required=False)

# Interfaces (Task05: Loopbacks, Task08: Ethernets)
```


Copyright © Cisco

```
interfaces:
  ethernets: list(include('ethernets'), required=False)
  loopbacks: list(include('loopbacks'), required=False)

ethernets:
  type: any(enum('GigabitEthernet', 'FastEthernet', 'Ethernet', 'Port-channel', 'FiveGigabitEthernet', 'TenGigabitEthernet', 'TwentyFiveGigE',
'FortyGigabitEthernet', 'HundredGigE', 'TwoHundredGigE', 'FourHundredGigE'), regex("^.*[\$\%]\{.*$"))
  id: str()
  description: str(required=False)
  shutdown: any(bool(), regex("^.*[\$\%]\{.*$"), required=False)
  ipv4: include('interface_ipv4', required=False)

interface_ipv4:
  address: any(ip(), regex("^.*[\$\%]\{.*$"), required=False)
  address_mask: any(ip(), regex("^.*[\$\%]\{.*$"), required=False)

# Loopback Interfaces (Task05)
loopbacks:
  id: any(int(), regex("^.*[\$\%]\{.*$"))
  description: str(required=False)
  shutdown: any(bool(), regex("^.*[\$\%]\{.*$"), required=False)
  ipv4: include('interface_ipv4', required=False)
```

Appendix I - CI/CD pipeline definition

Appendix III - nac-test reference

Copyright © Cisco

Appendix III - `nac-test` reference

Task 11 uses `nac-test` to render and execute Robot Framework tests from your intent YAML. The lab WSL image ships the full test scaffolding at `~/tests/`; Task 11's Step 1 copies it into your project with `cp -r ~/tests ~/nac-iosxe/`.

This appendix shows the **rendered** Robot test you'll end up with - the file that `nac-test` produces after expanding the Jinja2 template with your merged model data. It's the file you open in `TestResults/config/access_lists.robot` after running `nac-test` in Task 11.

For the full template sources (`access_lists.robot`, `iosxe_common.resource`, `UtilsLib.py`, `url_encode.py`), read them directly from `~/tests/` in the WSL image, or browse them online in the Cisco-maintained nac-test-pyats-common repository.

Rendered `access_lists.robot`

Below is what you get after `nac-test` renders the Jinja2 template against your intent YAML. Notice how the two entries from your `AccessLayerACL` (sequence 10 and 20) become explicit XPath assertions, one per device in the `ACCESS_SWITCHES` group:

OUTPUT **tests/results/config/access_lists.robot (rendered)**

```
*** Settings ***
Documentation   Verify Access Lists Configuration
Suite Setup     Run Only Once   Get Configs
Resource        ../iosxe_common.resource
Default Tags    config   iosxe   access_lists

*** Test Cases ***

Get Config access01
    ${config}=   Get Parallel Value For Key   access01_config
    Set Suite Variable   ${config}

Verify Standard Access List AccessLayerACL Device access01
    ${acl}=   Set Variable   data/native/ip/access-list/standard[name='AccessLayerACL']
    Should Be Equal Value Xml String   ${config}   ${acl}/name   AccessLayerACL
    ${entry}=   Set Variable   ${acl}/access-list-seq-rule[sequence='10']
    Should Be Equal Value Xml String   ${config}   ${entry}/remark
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/ipv4-address-prefix   10.0.0.0
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/mask   0.0.0.255
    Should Be Equal Value Xml Bool   ${config}   ${entry}/permit/std-ace/any
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/host-address
    Should Be Equal Value Xml Bool   ${config}   ${entry}/permit/std-ace/log
    ${entry}=   Set Variable   ${acl}/access-list-seq-rule[sequence='20']
    Should Be Equal Value Xml String   ${config}   ${entry}/remark
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/ipv4-address-prefix   20.0.0.0
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/mask   0.0.0.255
    Should Be Equal Value Xml Bool   ${config}   ${entry}/permit/std-ace/any
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/host-address
    Should Be Equal Value Xml Bool   ${config}   ${entry}/permit/std-ace/log

Get Config access02
    ${config}=   Get Parallel Value For Key   access02_config
    Set Suite Variable   ${config}

Verify Standard Access List AccessLayerACL Device access02
    ${acl}=   Set Variable   data/native/ip/access-list/standard[name='AccessLayerACL']
    Should Be Equal Value Xml String   ${config}   ${acl}/name   AccessLayerACL
    ${entry}=   Set Variable   ${acl}/access-list-seq-rule[sequence='10']
    Should Be Equal Value Xml String   ${config}   ${entry}/remark
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/ipv4-address-prefix   10.0.0.0
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/mask   0.0.0.255
    Should Be Equal Value Xml Bool   ${config}   ${entry}/permit/std-ace/any
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/host-address
    Should Be Equal Value Xml Bool   ${config}   ${entry}/permit/std-ace/log
    ${entry}=   Set Variable   ${acl}/access-list-seq-rule[sequence='20']
    Should Be Equal Value Xml String   ${config}   ${entry}/remark
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/ipv4-address-prefix   20.0.0.0
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/mask   0.0.0.255
    Should Be Equal Value Xml Bool   ${config}   ${entry}/permit/std-ace/any
    Should Be Equal Value Xml String   ${config}   ${entry}/permit/std-ace/host-address
    Should Be Equal Value Xml Bool   ${config}   ${entry}/permit/std-ace/log
```

Assertions where the expected value is blank (e.g. `remark`, `host-address`) are no-ops - the keyword returns immediately when the value is empty. They appear because the Jinja2 template renders every possible field; only the fields you defined in YAML actually perform a check.

Copyright © Cisco

How `nac-test` wires the pieces together

1. **Jinja2 template** - `tests/templates/config/access_lists.robot` is a Jinja2 template that iterates over every device's `access_lists.standard[]` in the merged model.
2. **Filters** - `tests/filters/url_encode.py` provides the `url_encode` filter (e.g. to encode ACL names with special characters in XPath queries).
3. **Resource file** - `tests/templates/iosxe_common.resource` defines reusable Robot keywords shared across all test suites.
4. **Python library** - `tests/templates/lib/UtilsLib.py` uses `scrapli_netconf` to fetch the running config via NETCONF (`Get Configs` keyword) and exposes XML assertion keywords like `Should Be Equal Value Xml String` and `Should Be Equal Value Xml Bool`.
5. **Pabot** - `nac-test` invokes the rendered Robot suites in parallel via Pabot, so all four devices run concurrently.

Extending the pattern

Most of the Cisco-maintained `nac-test` template set (interfaces, routing protocols, VLANs, etc.) lives in the same style as the snippet above. To add your own test:

1. Create a new `.robot` Jinja2 template under `tests/templates/config/` .
2. Walk `iosxe.devices[]` → `device.configuration.<your feature>` to pull fields from the merged model.
3. Emit one `*** Test Cases ***` block per device/resource.
4. Re-run `nac-test` - the new suite is rendered and executed automatically.

The Jinja2 context `nac-test` exposes is the full merged data model (the same `model.yaml` you inspected in earlier tasks) plus a `defaults` variable containing the NaC module's default values. Anything you can put in YAML, you can assert against from a template.

Appendix II - Schema reference

Appendix IV - ISP Router Configuration

Copyright © Cisco

Appendix IV - ISP router configuration

The **isp** router in the CML topology is pre-configured by the lab environment - you don't edit it, but it's useful to know what's on it. This appendix is the reference.

Why is the ISP pre-configured?

Task 08 (Templates type `file`) has you configure `border` as an eBGP speaker peering with `isp`. For that peering to come up, the other end has to be ready - `isp` needs its own BGP configuration (AS 65001, neighbor statement pointing back at `border`, network advertisement for `8.8.8.0/24`).

Two reasons it's pre-provisioned rather than having you configure both sides:

- **Time.** Configuring both ends of a BGP session inside a 4-hour lab burns ~15 minutes of wall clock on a pattern that isn't the learning point - the learning point is "Network as Code can push BGP config to my side", not "I can configure BGP on two devices by hand".
- **Simulation scope.** The `isp` router stands in for an external service provider. Real ISPs don't give you a login to their edge routers; you configure your side, they configure theirs, and the session comes up if both sides are consistent. Splitting the work across both sides of the peering preserves that mental model.

With the ISP side already in place, Task 08's job is just "get `border`'s side consistent with it". You'll see the BGP session move from `Idle` → `Active` → `OpenSent` → `Established` over 30-60 seconds once Terraform applies your config.

Configuration reference

Below is the BGP and interface configuration on the **isp** router. Use this as a reference if you want to see exactly what your `border` configuration needs to match (AS 65001, neighbor `198.18.100.2`, advertisement of `8.8.8.0/24`).

OUTPUT **isp router show run**

```
isp#show run | sec bgp
router bgp 65001
 bgp log-neighbor-changes
 neighbor 198.18.100.2 remote-as 65000
 neighbor 198.18.100.2 description eBGP to BORDER
 neighbor 198.18.100.2 timers 60 180 30
 !
 address-family ipv4
  network 8.8.8.0 mask 255.255.255.0
  neighbor 198.18.100.2 activate
 exit-address-family
isp#
isp#show run int gig 0/1
Building configuration...

Current configuration : 142 bytes
!
interface GigabitEthernet0/1
 description border G1
 ip address 198.18.100.1 255.255.255.252
 duplex auto
 speed auto
 media-type rj45
end

isp#
```

The `8.8.8.0/24` advertisement is a stand-in for "the internet" - when your BGP session comes up, `border` learns this route and can forward to it (see Task 08's verification step).

Appendix III - nac-test reference

Appendix V - Quick reference

Copyright © Cisco

Appendix V - Quick reference card

One-page cheat sheet covering everything you need to rerun the lab - at home, on your own devices, or from memory. Bookmark this page; print it if that's your style.

Lab devices

Device	Role	Mgmt IP	User	Pass
core	Core switch (Catalyst 9000v)	`198.18.130.10`	cisco	cisco
border	Border router (Catalyst 8000v)	`198.18.130.20`	cisco	cisco
access01	Access switch (Catalyst 9000v)	`198.18.130.11`	cisco	cisco
access02	Access switch (Catalyst 9000v)	`198.18.130.12`	cisco	cisco
isp	Upstream BGP peer (pre-configured)	`198.18.130.200`	cisco	cisco
host01 / **host02**	End-hosts (CML console only)	`192.168.100.100/200`	cisco	cisco
ntp-server	NTP service (Ubuntu VM)	`198.18.129.11`	-	-
syslog-server	Syslog service (Ubuntu VM)	`198.18.129.12`	-	-

Network as Code manages the top four. The rest are pre-configured and used for verification only.

Lab VMs

VM	Address	User	Pass	Access
Windows 10	`198.18.133.20`	admin	cisco	RDP
CML	`198.18.130.34`	guest	CiscoLive	HTTPS
Ubuntu	`198.18.133.101`	guest	CiscoLive	SSH
GitLab	`198.18.133.101`	root	C1sco12345	HTTPS

GitLab and Ubuntu share the host (same `198.18.133.101`), different services.

Automation seed config (on each Network as Code-managed device)

DEVICE CLI · PuTTY **all NaC-managed devices**

```
username nac_admin privilege 15 secret cisco
netconf-yang
```

That's the minimum. `restconf` + `ip http secure-server` are optional if you want RESTCONF-based verification as well.

Environment variables

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

Copyright © Cisco

```
export IOSXE_USERNAME=nac_admin
export IOSXE_PASSWORD=cisco
export IOSXE_PROTOCOL=netconf   # or restconf
```

Load per-session with `source .env`, or append to `~/.bashrc` for persistence.

NETCONF reachability preflight

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
ssh -s -p 830 -o StrictHostKeyChecking=no $IOSXE_USERNAME@<device_ip> netconf
```

Expect an XML `<hello>` with a capability list. **Ctrl+C** to exit.

Project layout

OUTPUT **~/nac-iosxe/ project layout**

```
~/nac-iosxe/
├── .env                                # credentials
├── .schema.yaml                        # NaC schema (for nac-validate)
├── main.tf                             # Terraform module configuration
├── data/                               # your YAML intent
│   ├── devices/core.nac.yaml       # one file per device
│   ├── devices/border.nac.yaml
│   ├── devices/access01.nac.yaml
│   ├── devices/access02.nac.yaml
│   ├── global.nac.yaml             # applies to all devices
│   ├── groups/access.nac.yaml       # applies to ACCESS_SWITCHES group
│   └── template-*.nac.yaml               # optional reusable templates
├── tftpl/                              # optional .tftpl template files
├── tests/                              # optional nac-test suite
├── model.yaml        ← generated by `terraform apply`
├── defaults.yaml     ← generated
└── terraform.tfstate ← generated (Terraform state)
```

`main.tf` skeleton

FILE **main.tf**

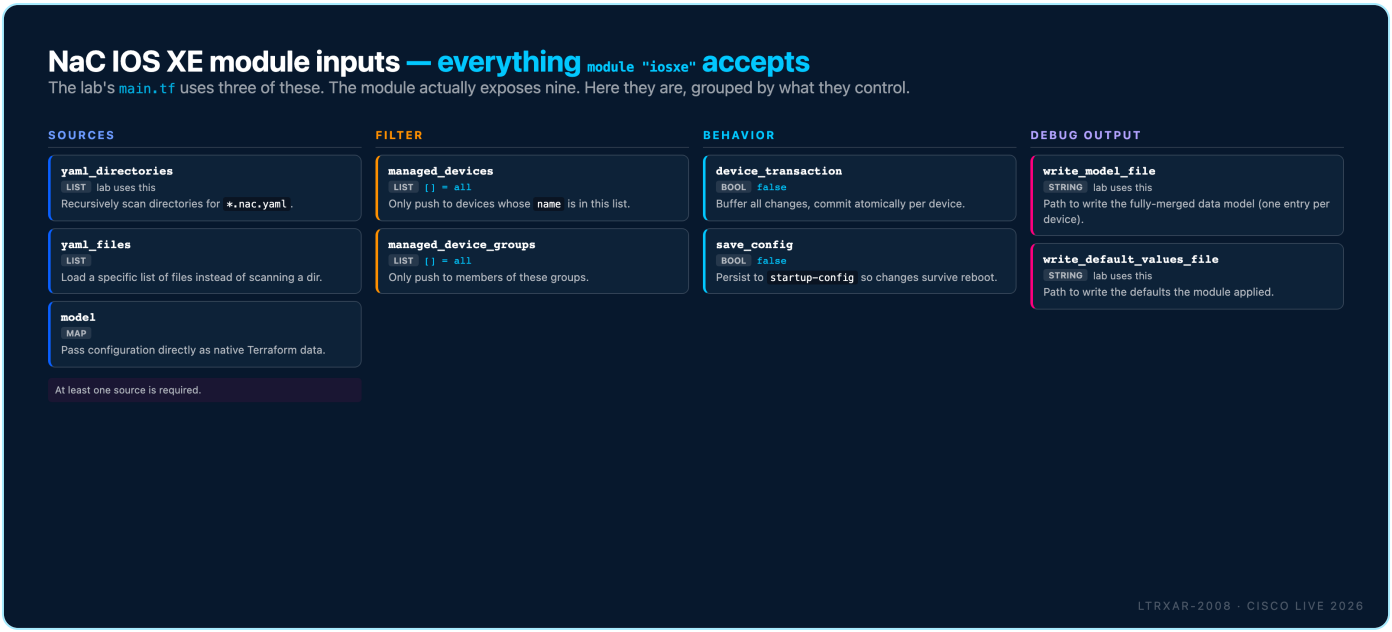
```
module "iosxe" {
  source                   = "git::https://github.com/netascode/terraform-iosxe-nac-iosxe.git"
  yaml_directories         = ["data/"]
  write_model_file         = "model.yaml"
  write_default_values_file = "defaults.yaml"
}
```

In production, pin to a specific tag: `?ref=v0.12.3`.

Module inputs reference

The IOS XE as Code module accepts nine input variables. The lab uses three. Here are all nine, grouped by what they control:

Copyright © Cisco



Quick decision table:

You want to...	Set
Scan a directory of YAML files	`yaml_directories = ["data/"]`
Load a specific list of files	`yaml_files = ["data/core.nac.yaml", ...]`
Provide config as a Terraform map	`model = { … }`
Push to only a subset of devices	`managed_devices = ["core", "border"]`
Push to only members of certain groups	`managed_device_groups = ["ACCESS_SWITCHES"]`
Commit atomically per device (multi-RPC safety)	`device_transaction = true`
Persist to `startup-config` so config survives reboot	`save_config = true`
Write the merged data model for inspection	`write_model_file = "model.yaml"`
Write the resolved defaults for inspection	`write_default_values_file = "defaults.yaml"`

Fleet-level vs per-device transactions

Two separate "atomicity" layers are worth distinguishing because they're easy to conflate:

Scope	Mechanism	Guarantee
Single device, single apply	NETCONF candidate datastore + `<commit>` (see Task 01 diagram)	All RPCs in one Terraform-to-device session commit together, or all roll back. Requires `netconf-yang feature candidate-datastore` on the device.
Multiple RPCs, single apply, same device	Module's `device_transaction = true` input	Terraform buffers all changes destined for a device and commits them in one NETCONF transaction. Same guarantee, applied per-device across every changeset Terraform computes.
Fleet-level (across devices)	Not available - Terraform applies per-device	If an apply touches 10 devices and device #7 fails, the first six have their changes committed; devices 7-10 don't. There is **no** "all 10 devices or none" primitive.

Copyright © Cisco

For the lab's 4-device deployment, the per-device transaction layer is what matters. For real production rollouts, plan for the fleet-level case: stage rollouts by pipeline phase, roll back via a subsequent `apply` that reverses the intent, or use `managed_devices` to scope the blast radius.

Terraform command reference

Command	What it does	Changes devices?
`terraform init`	Download module + provider, once per project	No
`terraform init -upgrade`	Pull latest module versions	No
`terraform validate`	Check `.tf` syntax	No
`terraform fmt`	Auto-format `.tf` files	No
`terraform plan`	Diff: desired vs actual. Print the plan	No
`terraform plan -out=plan.tfplan`	Same, save for later apply	No
`terraform apply`	Execute the plan (prompts to confirm)	**Yes**
`terraform apply -auto-approve`	Apply without confirmation (CI only)	**Yes**
`terraform apply plan.tfplan`	Apply a previously-saved plan	**Yes**
`terraform destroy`	Remove all Terraform-managed config	**Yes**
`terraform show`	Display current state	No
`terraform state list`	List tracked resources	No

`nac-validate` - pre-deployment schema check

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
pip install nac-validate
export PATH=$PATH:~/.local/bin

# Validate every YAML under data/ against .schema.yaml
nac-validate -s .schema.yaml data/
```

Exit code 0 = valid. Non-zero + `ERROR - Syntax error ...` = something's wrong. The error message names the exact merged-model path at fault.

`nac-test` - post-deployment verification

TERMINAL · WSL/Ubuntu **cisco@wkst1:~/nac-iosxe$**

```
pip install nac-test

# Assumes `tests/` directory is present (see Task 11)
nac-test \
  --data ./model.yaml \
  --data ./defaults.yaml \
  --templates ./tests/templates \
  --filters ./tests/filters \
  --output ./tests/results
```

Artifacts in `./tests/results/` :

- `report.html` - human-readable summary, open in browser
- `log.html` - step-by-step execution log
- `xunit.xml` - JUnit XML for CI integration

Copyright © Cisco

GitLab CI/CD stages

Stage	Job	Runs on MR?	Runs on main?
`validate`	`terraform fmt -check` , `nac-validate`		
`plan`	`terraform plan` + post plan to MR comment		
`deploy`	`terraform apply -auto-approve plan.tfplan`		
`test`	`nac-test` integration + `terraform plan -detailed-exitcode` idempotency		
`notify`	Webex / email on success or failure		

MR pipelines are preview-only. `main` pipelines deploy.

Further reading

- netascode.cisco.com - Network as Code data models + tool docs for IOS XE, SD-WAN, ACI, Catalyst Center, Nexus Dashboard
- `terraform-iosxe-nac-iosxe` - the module used in this lab (Apache 2.0)
- `terraform-provider-iosxe` - the IOS XE Terraform provider underneath
- netascode on GitHub - all the modules, tools, and data models

Appendix IV - ISP Router Configuration

Copyright © Cisco