Semester Project

Network Configuration Automation with Infrahub and Nornir

Semester HS 2024

Version 1.0
December 20, 2024

Students: Simon Linder
Polina Lisetska
Ramon Stutz

Advisors: Urs Baumann
Jan Untersander

INS | Institute for Network and Security

OST Eastern Switzerland University of Applied Sciences

Department of Computer Science
OST - Eastern Switzerland University of Applied Sciences

1. Abstract

This document outlines the design and implementation of an automated system for **VLAN management** as part of the project **"Network Configuration Automation with Infrahub and Nornir"**. The solution leverages Infrahub as a single source of truth, providing a unified platform for managing network infrastructure data. **Nornir**, a Python-based automation framework, handles configuration deployment with **dry-run validation** to detect potential misconfigurations.

- **Infrahub** serves as the central repository for network infrastructure data, providing version control and collaboration through **GitLab**.
- **Python Transformers:** convert Infrahub data into a Pydantic model representing YANG models.
- **NETCONF XML Exporter** generates valid NETCONF payloads.
- **Nornir** automates the deployment of NETCONF configurations, offering flexibility and scalability.
- **Conditional Runner** enforces concurrency limits and controlled task execution, enhancing automation stability.

Key Features

- **Dry-Run Validation**: Identifies misconfigurations preemptively.
- **Reconciliation Mechanism**: Detects configuration drifts for remediation.
- **Centralized Management**: Streamlines oversight of network configurations.

Benefits

Automating VLAN configuration reduces human intervention, minimizes errors, and enhances network reliability. Validation and centralized management improve efficiency, reliability, and scalability.

Conclusion

The **"Network Configuration Automation with Infrahub and Nornir"** project automates VLAN management by enforcing validation and centralizing oversight, modernizing network management for greater efficiency and scalability.

2. Vision

Each company manages a network inventory in some form. Imagine leveraging this inventory to automate the deployment of network configurations. Infrahub centralizes network configurations, enabling network operators to easily update the user-friendly inventory system. By automating deployment, our solution ensures the consistent application of the desired state across the network. Pydantic is utilized for model validation, enhancing data integrity and ensuring configuration validity. This streamlines network management and improves efficiency. The platform is designed to be easy to use, scalable, and secure, allowing network operators to focus on strategic initiatives and innovation.

Goals

- The centralization of network configurations
- The automation of network configuration deployment
- Reconciliation of the desired state across the network
- The enhancement of data integrity
- The streamlining of network management
- The improvement of efficiency

Technologies

We leverage cutting-edge technologies that are not yet widely adopted in the industry. Infrahub serves as the inventory system and configuration user interface. Data from Infrahub is transformed into a Pydantic model, and Nornir is utilized to deploy NETCONF configurations to network devices.

Technology stack

- Infrahub
- Pydantic
- Nornir
- NETCONF
- YANG

3. Management Summary

Initial Situation

Network configuration management is a critical aspect of maintaining modern infrastructures, requiring consistent updates and validations across a wide range of devices. Traditionally, this process is time-consuming, error-prone, and often relies on manual intervention, which can lead to inconsistencies and network disruptions. Organizations often struggle with reconciling current network states with their desired configurations while ensuring operational stability and scalability. Furthermore, auditing is tedious with configurations distributed across devices.

Objective

The project, Network Configuration Automation with Infrahub and Nornir, aims to enhance network configuration management by automating deployment tasks and consolidating network configurations through a centralized system, Infrahub. Developed by OpsMill, Infrahub merges Git's version control with the flexibility of graph databases, offering a unified platform for managing infrastructure data. Our solution focuses on continuous reconciliation, ensuring that the network's state is consistently aligned with the defined configurations.

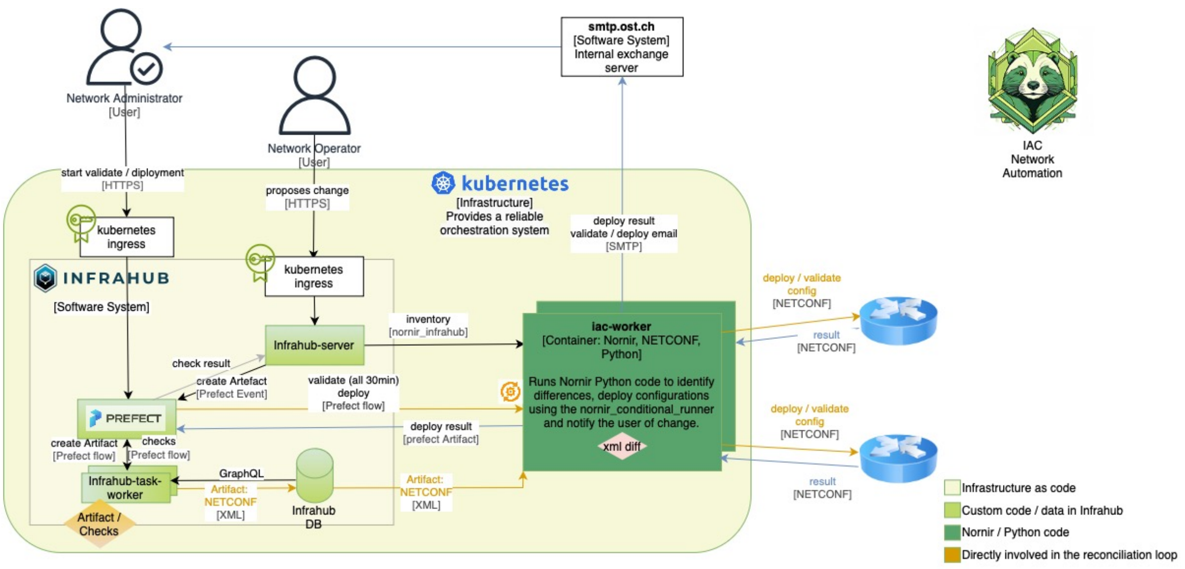


Figure 3.1.: The diagram shows an automated deployment via Infrahub, Prefect, and IaC Worker, integrating validation and deployment.

By centralizing oversight through Infrahub and automating deployment tasks with the Nornir framework, the project significantly improves efficiency and control. To enhance scalability and modularity, we integrated Prefect for workflow orchestration, NETCONF for communication, and dry-run validation to preview configurations before applying them safely. Designed for diverse network environments, it supports various device types and YANG models for compatibility. GitLab integration adds version control and traceability, while the Conditional Runner plugin for Nornir enhances stability by managing concurrency limits during maintenance. These features collectively streamline network operations and improve overall efficiency.

Results

The project successfully delivered a comprehensive and automated network configuration system tailored for VLAN management. By leveraging **Infrahub** as the single source of truth, the solution ensures all network data remains centralized, consistent, and easily accessible. **Nornir** was employed as the core automation framework, enabling efficient deployment of configurations while supporting dry-run validation to identify and prevent potential issues before applying changes. To ensure system stability and reliability, **Conditional Runner** was integrated, allowing for controlled execution and reducing the likelihood of unexpected failures. Additionally, **GitLab** was utilized for version control, ensuring that all changes are tracked, auditable, and aligned with modern CI/CD practices.

In summary, this integrated solution significantly enhances **scalability** and improves **reliability** by minimizing human errors through automation and introducing robust validation mechanisms. Furthermore, it strengthens overall **network stability** by enforcing consistent configurations and maintaining visibility into all changes. By reducing manual effort, the system not only saves time but also allows network teams to focus on higher-value tasks, ultimately driving operational efficiency and improving long-term maintainability.

4. Acknowledgments

We sincerely express our deepest gratitude to our advisors, `Urs Baumann` and `Jan Untersander`, for their professional support and expert guidance throughout the course of this project. Their insightful advice and constructive feedback have been valuable and helpfull.

We are also grateful to the OpsMill Infrahub and Prefect Discord communities for their quick and helpful responses during challenging moments.

Finally, we thank our friends and family for proofreading and providing valuable feedback.

Contents

Part I.

Product Documentation

1. Requirements

The requirements for the project are divided into two categories: functional and non-functional requirements.

1.1. Functional Requirements

The functional requirements are derived from the actors and their goals, as outlined in user stories identified during a dedicated requirements meeting with our customer, INS. The actors are grouped into two key personas: (1) Admin (Network Architect/Administrator), representing the individual responsible for managing INS's network, and (2) User (Network Operator), representing the person using our solution. Overall, these functional requirements provide us with comprehensive coverage of our assignment and serve as a clear roadmap for our development.

1.1.1. Persona

- **Admin (Network Architect/Administrator)**: A technical expert responsible for designing, automating, and overseeing the network architecture. The Admin ensures that the overall network design is scalable, secure, and compliant with organizational standards, and supervises the deployment and integration of network changes.
- **User (Network Operator)**: A less technical persona responsible for maintaining and managing the day-to-day operations of the network infrastructure. The User performs network configuration updates, ensures network stability, and handles troubleshooting tasks.

1.1.2. User Stories

In this section, we outline the user stories for the project, each written from the perspective of the defined personas. The user stories are categorized into two scopes: (1) In Scope – user stories that will be implemented as part of this project; (2) Implementation Out of Scope – user stories that will not be required for this project but could be implemented if there is enough time and may be documented for potential future work.

In Scope

- **U-1:** As a User, I want a centralized inventory system (Infrahub) that manages the current and desired state of the network, so that I can easily view and update the network configuration without needing manual tracking.
- **U-2:** As a User, I want to reliably apply the VLAN network configurations only using the easy GUI of the inventory system, so that I do not need to connect to each network device individually and manually configure them.
- **U-3:** As a User, I want to simulate VLAN configuration changes with a dry run, so that I can verify the network updates before they are applied and avoid potential errors.
- **U-4:** As a User, I want to easily adjust network VLAN configuration inputs via Infrahub, so that I can quickly respond to network changes or issues without reconfiguring everything from scratch.
- **U-5:** As a User, I want to receive notifications when a configuration change is applied successfully or fails, so that I can stay informed about the state of the network in real time.

- **U-6:** As an Admin, I want the solution to utilize structured data, such as Pydantic models for data structures and NETCONF XML payloads with universal YANG models wherever possible, to ensure that configurations stay consistent and compatible across a wide range of network devices.
- **U-7:** As an Admin, I want to parallelize the configuration using Nornir but ensure with a custom runner certain conditions, such as ensuring that the two core routers do not update simultaneously. This will allow for faster reconfiguration while maintaining network stability.
- **U-8:** As an Admin, I want the solution to utilize the machine configuration standard NETCONF so that configurations can be reliably applied without needing to undo current configurations that were applied manually or resetting a device before deployment.
- **U-9:** As an Admin, I want a solution that will compare the current network state with the desired state before applying changes, so that I only implement necessary updates and maintain network stability.
- **U-10:** As an Admin, I want the system to apply changes only when there is a difference between the current and desired state, so that I avoid unnecessary changes and keep the network running efficiently.
- **U-11:** As an Admin, I want the solution to perform a rollback of the configuration changes if the configuration fails, so that I can ensure the network remains operational and stable.
- **U-12:** As an Admin, I want access to logs of configuration changes and dry run results so that I can review and troubleshoot any issues that arise after the solution has applied a configuration or rejected one.

Out of Scope

- **U-13:** As a User, I want access to simplified logs and results of configuration changes and dry runs back in Infrahub, so that I can review and troubleshoot any issues after configurations are applied.
- **U-14:** As an Admin, I want detailed feedback in Infrahub from the dry run and configuration deployment, so that I can ensure the configurations are correct and troubleshoot any issues before deployment.
- **U-15:** As an Admin, I want the solution to also configure and monitor the base configuration of the network devices, so that I can ensure that the network devices are correctly configured and operational.
- **U-16:** As an Admin, I want the system to support version control for configurations, so that in case of a failure, I can roll back to the previous state without experiencing downtime.

1.2. Non-Functional Requirements

1.2.1. Approach

The non-functional requirements were established during the requirements analysis meeting with INS and are derived from the functional requirements. Based on ISO/IEC 25010, they are categorized as follows:

In Scope

- **Reliability**: The system should ensure that network configuration changes are applied correctly **99.9% of the time**.
- **Compatibility**: The system must comply with industry-standard protocols like **NETCONF** and **YANG**, ensuring compatibility with existing networking hardware and configurations.
- **Security**: Access to the system should be restricted based on user roles and permissions.
- **Maintainability**: Maintainability refers to the system's ability to efficiently adapt to changes through modularity, reusability, analysability, modifiability, and testability.

Out of Scope

- **Performance Efficiency**: The system should optimize resource utilization and maintain high performance under load, ensuring timely deployment of configurations.
- **Interaction Capability**: The system must be user-friendly, providing appropriate user assistance, error protection, and self-descriptive messages for troubleshooting and operations.
- **Flexibility**: The system should support diverse network environments and scalability to adapt to future infrastructure needs.

2. Preliminary Work

Several open-source projects, including "Infrahub," "Nornir_Netconf," "Pydantic," and "Pydantify," are currently available. However, these tools have not yet been integrated to support our specific use case. Consequently, there is no existing preliminary work to build upon within the context of our project. Additionally, bridging these tools will facilitate streamlined network configuration management.

3. Architecture

This document outlines the network automation system's architecture using the C4 model, emphasizing core components, interactions, and architectural considerations. It serves as a reference for understanding the system, supported by Use Case, Context, Container, and Component diagrams to illustrate various levels of detail.

3.1. Introduction and Goals

The project's goal is to develop an automated network management system using **Infrahub** as a central hub and single source of truth for network configurations. The system automates pulling current network states, validating changes via dry runs, and applying new configurations to network devices using the **NETCONF** protocol. Python is the primary language for automation, integrated with **Pydantic** for model generation based on **YANG** data models, chosen for its compatibility with our Python-based ecosystem.

Use Cases Diagram As the architecture is complex, we created use case diagrams to illustrate the system's functionality, highlighting two workflows: `change of the desired config` and `change of the running config`. The diagram includes the `Pydantic Model` component, representing interactions typically encapsulated within an Infrahub Artifact. This approach clarifies the data flow and conversion steps added by the project.

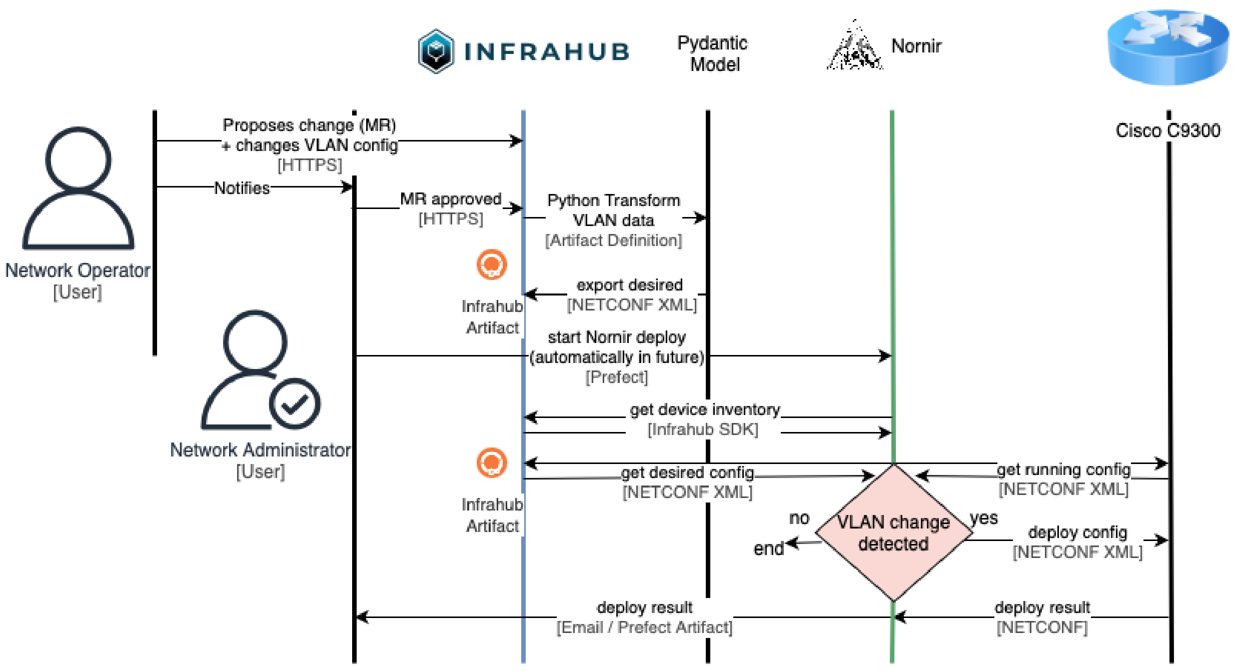

INFRAHUB
Pydantic Model
Nornir
Cisco C9300
Network Operator [User]
Network Administrator [User]
Proposes change (MR) + changes VLAN config [HTTPS]
Notifies
MR approved [HTTPS]
Python Transform VLAN data [Artifact Definition]
Infrahub Artifact
export desired [NETCONF XML]
start Nornir deploy (automatically in future) [Prefect]
get device inventory [Infrahub SDK]
get desired config [NETCONF XML]
get running config [NETCONF XML]
Infrahub Artifact
no
end
VLAN change detected
yes
deploy config [NETCONF XML]
deploy result [Email / Prefect Artifact]
deploy result [NETCONF]

Figure 3.1.: Use Case Workflow 1: Changing the network VLAN configuration

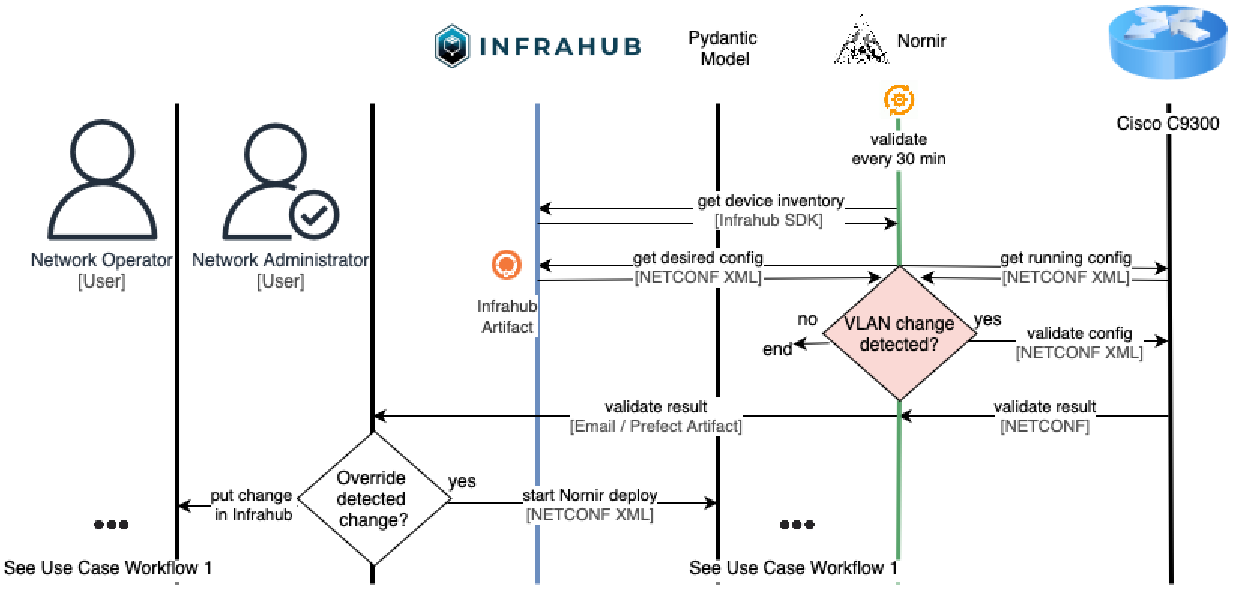

INFRAHUB
Pydantic Model
Nornir
Cisco C9300
validate every 30 min
Network Operator [User]
Network Administrator [User]
Infrahub Artifact
get device inventory [Infrahub SDK]
get desired config [NETCONF XML]
get running config [NETCONF XML]
VLAN change detected?
no
end
yes
validate config [NETCONF XML]
validate result [Email / Prefect Artifact]
validate result [NETCONF]
Override detected change?
yes
put change in Infrahub
start Nornir deploy [NETCONF XML]
See Use Case Workflow 1
See Use Case Workflow 1

Figure 3.2.: Use Case Workflow 2: Showing the reconciliation of desired and running config

3.2. Context Diagram (Level 1)

The context diagram provides a high-level overview of the system and its interactions with external entities, such as users and other systems.

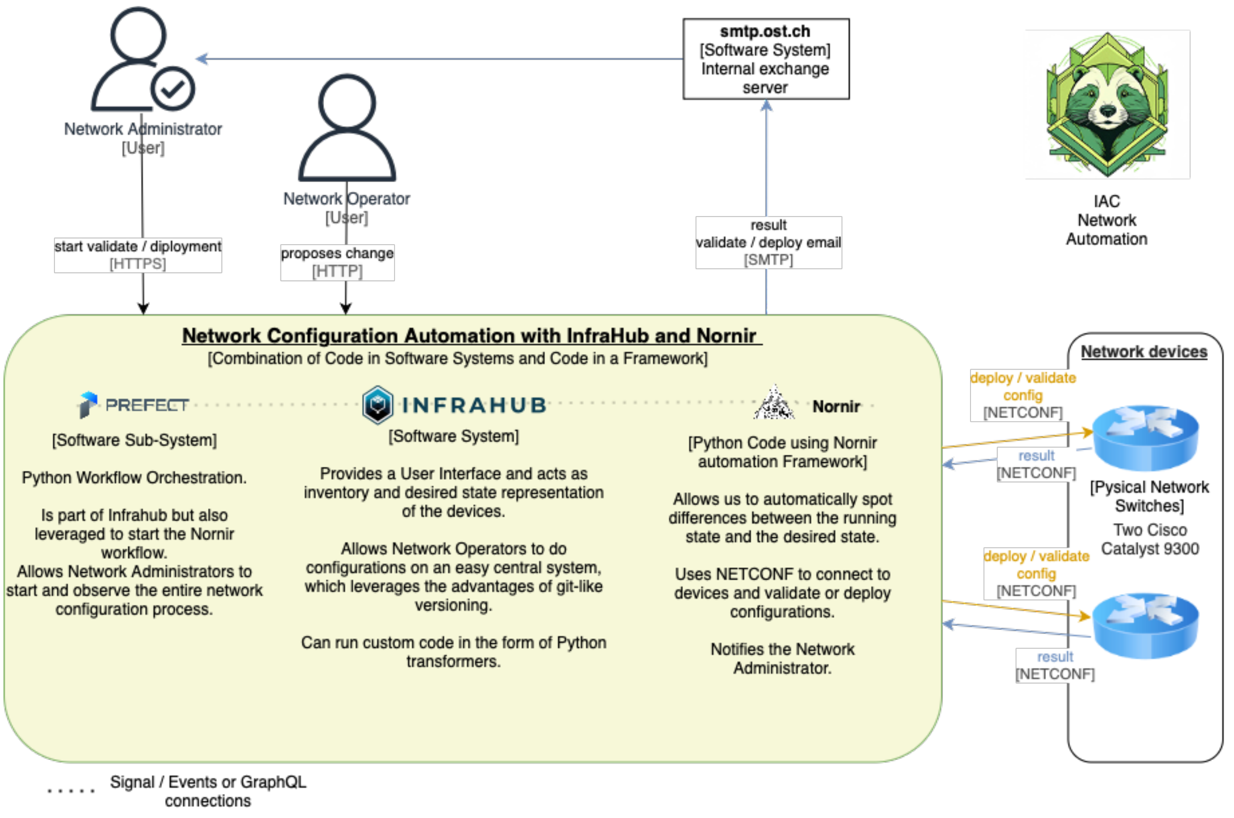

smtp.ost.ch
[Software System]
Internal exchange server
Network Administrator [User]
Network Operator [User]
IAC Network Automation
start validate / diployment [HTTPS]
proposes change [HTTP]
result validate / deploy email [SMTP]
Network Configuration Automation with InfraHub and Nornir
[Combination of Code in Software Systems and Code in a Framework]
PREFECT
[Software Sub-System]
Python Workflow Orchestration.
Is part of Infrahub but also leveraged to start the Nornir workflow.
Allows Network Administrators to start and observe the entire network configuration process.
INFRAHUB
[Software System]
Provides a User Interface and acts as inventory and desired state representation of the devices.
Allows Network Operators to do configurations on an easy central system, which leverages the advantages of git-like versioning.
Can run custom code in the form of Python transformers.
Nornir
[Python Code using Nornir automation Framework]
Allows us to automatically spot differences between the running state and the desired state.
Uses NETCONF to connect to devices and validate or deploy configurations.
Notifies the Network Administrator.
Network devices
deploy / validate config [NETCONF]
result [NETCONF]
[Pysical Network Switches]
Two Cisco Catalyst 9300
deploy / validate config [NETCONF]
result [NETCONF]
Signal / Events or GraphQL connections

Figure 3.3.: C4 Context Diagram

1. **User:** The user (network administrator or operator) interacts with Infrahub or Prefect to configure and manage the network.

2. **Network Configuration Automation with Infrahub and Nornir:** As our code resides within Infrahub and Nornir, and both use Prefect for workflow orchestration, the system we develop is a combination of all these components. The components interact with each other to manage the network configuration using concepts like signal, events, and the wish template pattern (GraphQL).
   - **Infrahub:** Serves as the system's user interface, managing both the network device inventory and desired states of the network.
   - **Nornir:** This Python automation framework handles network configuration tasks, including fetching the current network state, detecting differences between the current network and desired state, deploying changes, validating configurations, and notifying the network administrator about network changes.
   - **Prefect:** Orchestrates the system's workflows, managing task initiation and providing feedback to the user. It provides a user interface for the network administrator to interact with the system.

3. **Network Devices:** External systems (routers, switches, etc.) managed via NETCONF. In our case, they are two Cisco Catalyst 9300 switches.

3.3. Container Diagram (Level 2)

The container diagram breaks down the system into its main containers, illustrating how each part collaborates to deliver overall functionality.

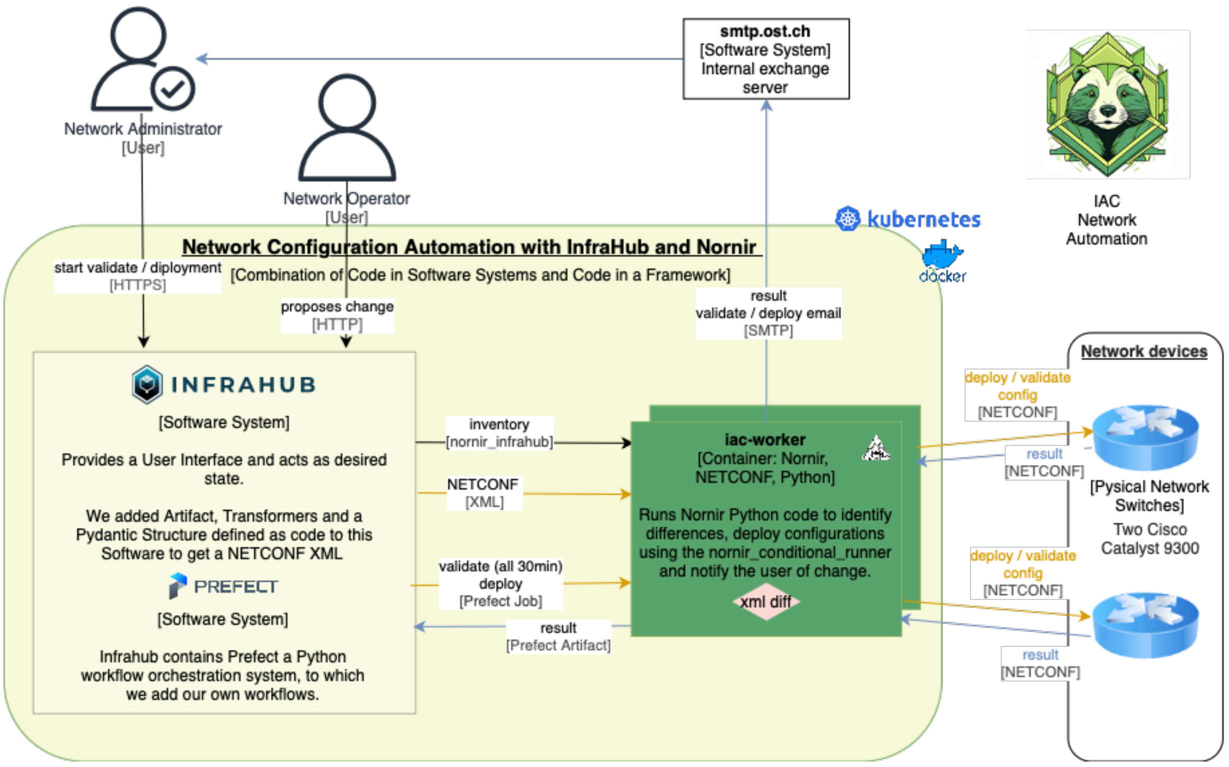

Network Administrator
[User]
Network Operator
[User]
smtp.ost.ch
[Software System]
Internal exchange server
IAC
Network
Automation
kubernetes
docker
Network Configuration Automation with InfraHub and Nornir
[Combination of Code in Software Systems and Code in a Framework]
start validate / diployment
[HTTPS]
proposes change
[HTTP]
result
validate / deploy email
[SMTP]
INFRAHUB
[Software System]
Provides a User Interface and acts as desired state.
We added Artifact, Transformers and a Pydantic Structure defined as code to this Software to get a NETCONF XML
PREFECT
[Software System]
Infrahub contains Prefect a Python workflow orchestration system, to which we add our own workflows.
inventory
[nornir_infrahub]
NETCONF
[XML]
validate (all 30min)
deploy
[Prefect Job]
result
[Prefect Artifact]
iac-worker
[Container: Nornir, NETCONF, Python]
Runs Nornir Python code to identify differences, deploy configurations using the nornir_conditional_runner and notify the user of change.
xml diff
Network devices
deploy / validate config
[NETCONF]
result
[NETCONF]
[Pysical Network Switches]
Two Cisco Catalyst 9300
deploy / validate config
[NETCONF]
result
[NETCONF]

Figure 3.4.: C4 Container Diagram

Key Containers

1. **User:** Interacts with the system through a web interface (UI) to provide the desired network state and review logs and feedback from dry runs and configuration deployments.
2. **Infrahub Server:** Handles multiple tasks:
   - **Web API/UI:** Interface for network operators to interact with Infrahub.
   - **Inventory Management:** Stores managed network devices and their configurations.
   - **State Management:** Manages the network's desired state.
   - **Pydantic Model + XML Generation:** Generates Pydantic models from YANG models for validating configurations and transforming them into XML for NETCONF.
   - **Workflow Orchestration:** Prefect orchestrates deployments, sending events to the sa-worker container and providing feedback via Prefect Artifacts.
3. **IaC-Worker:** Handles network configuration tasks:
   - **NETCONF Interaction:** Applies configurations and retrieves current states.
   - **Diff:** Compares current and desired states to identify changes.
   - **Dry Run Execution:** Simulates changes before application.
   - **Reconciliation Loop:** Ensures devices are in the desired state, notifying administrators of discrepancies.
   - **Configuration Deployment:** Applies desired state changes to devices.
   - **Logging and Feedback:** Provides logs and error messages for verification and troubleshooting.
4. **Network Devices:** External systems communicating with our IaC-workers via NETCONF to receive configurations and report states.
5. **SMTP Server:** Sends feedback to network administrators.

3.4. Component Diagram (Level 3)

This diagram details Infrahub's internal components, illustrating their interactions. We split the diagram into infrastructure and code components, aligning with our Infrastructure as Code (IaC) approach to enhance clarity and synchronization between infrastructure and software components.

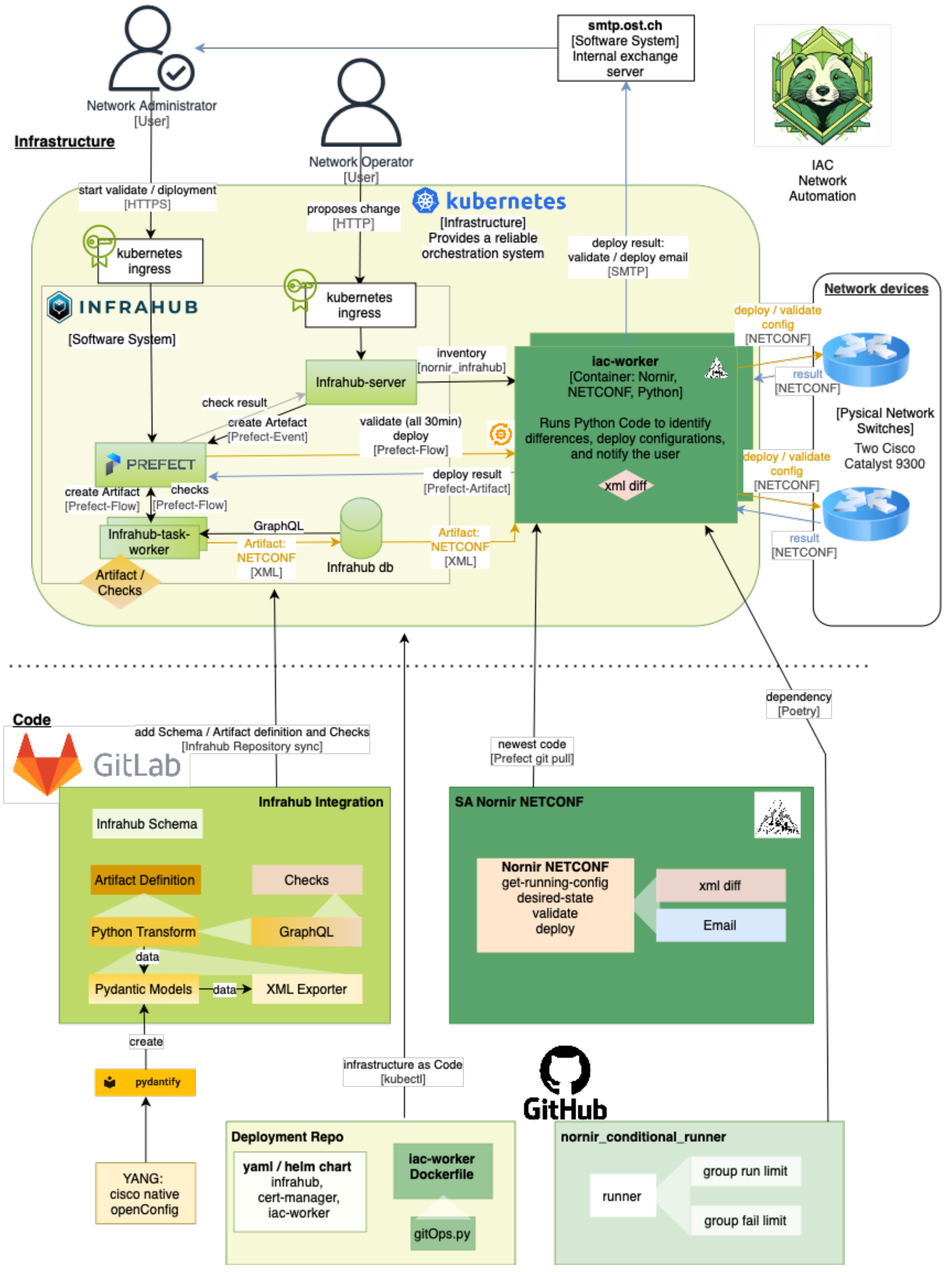

Network Administrator [User]
Network Operator [User]
smtp.ost.ch [Software System] Internal exchange server
IAC Network Automation
Infrastructure
start validate / diployment [HTTPS]
proposes change [HTTP]
kubernetes [Infrastructure] Provides a reliable orchestration system
deploy result: validate / deploy email [SMTP]
kubernetes ingress
kubernetes ingress
INFRAHUB [Software System]
Infrahub-server
inventory [nornir_infrahub]
iac-worker [Container: Nornir, NETCONF, Python]
Runs Python Code to identify differences, deploy configurations, and notify the user
xml diff
check result
create Artefact [Prefect-Event]
validate (all 30min) deploy [Prefect-Flow]
deploy result [Prefect-Artifact]
PREFECT
create Artifact [Prefect-Flow]
checks [Prefect-Flow]
GraphQL
Infrahub-task-worker
Artifact: NETCONF [XML]
Infrahub db
Artifact: NETCONF [XML]
Artifact / Checks
Network devices
deploy / validate config [NETCONF]
result [NETCONF]
[Pysical Network Switches] Two Cisco Catalyst 9300
deploy / validate config [NETCONF]
result [NETCONF]
dependency [Poetry]
Code
GitLab
add Schema / Artifact definition and Checks [Infrahub Repository sync]
newest code [Prefect git pull]
Infrahub Integration
Infrahub Schema
Artifact Definition
Checks
Python Transform
GraphQL
data
Pydantic Models
data
XML Exporter
create
pydantify
YANG: cisco native openConfig
SA Nornir NETCONF
Nornir NETCONF get-running-config desired-state validate deploy
xml diff
Email
infrastructure as Code [kubectl]
GitHub
Deployment Repo
yaml / helm chart infrahub, cert-manager, iac-worker
iac-worker Dockerfile
gitOps.py
nornir_conditional_runner
runner
group run limit
group fail limit

Figure 3.5.: C4 Component Diagram, showing the Infrastructure and Code components

Key Components

1. **Infrastructure:**
   - **Kubernetes Cluster:** Hosts Infrahub and IaC-Worker containers.
   - **Kubernetes Ingress:** Routes traffic to Infrahub and Prefect containers, utilizing HTTPS and a wildcard certificate from cert-manager and Let's Encrypt.
   - **Infrahub:** Central UI managing the network's desired state and generating custom NETCONF artifacts.
   - **Prefect:** Orchestrates system workflows, manages configurations, and handles internal Infrahub tasks.
   - **Infrahub Task Worker:** Handles tasks from Prefect, including artifact generation and running Python transformers.
   - **Infrahub DB:** Stores the network's desired state in a GraphQL database.
2. **Code:**
   - **YANG Models:** Utilizes OpenConfig VLAN and Cisco native interfaces YANG models to generate Pydantic models for network configuration.
   - **Pydantify:** Generates Pydantic models from YANG definitions.
   - **XML Exporter:** Converts Pydantic models into NETCONF XML payloads.
   - **Python Transform:** Transforms Infrahub data into Pydantic models using GraphQL to retrieve data from Infrahub DB.
   - **Artifact Definition:** Bundles Python transformers and XML Exporter into Infrahub Artifacts based on proposed changes.
   - **Checks:** Validates network configuration changes before deployment using data from the Infrahub DB.
   - **Infrahub Schema:** Customizes the UI and impacts the database schema.
   - **SA Nornir NETCONF:** Fetches inventory, retrieves configurations, performs diffs, and handles deployment and reconciliation loops.
   - **Deployment Repo:** Stores infrastructure as code, including Kubernetes resources, Helm charts, and Dockerfiles for Infrahub and IaC-Worker.
   - **Nornir_Conditional_Runner:** Ensures secondary devices remain online during primary device configuration changes.

3.5. Architectural Decisions

During the prototype setup to test components and their interactions, we made several key decisions:

3.5.1. Connection Plugin

Establishing a stable connection with network devices required a reliable plugin supporting the NETCONF protocol.

Requirements for the Connection Plugin:

1. Compatibility with Cisco IOS-XE and Arista EOS devices for integration testing.
2. Independent management of connection opening and closing.
3. Performance and reliability.
4. Ease of use and integration with the Nornir framework.
5. Comprehensive documentation.
6. Active maintenance.
7. Support for methods to get, lock, unlock, edit, commit, and validate configurations, enabling dry runs.
8. Structured output (XML or JSON) with RPC object attachment for advanced operations.

Tested Plugins We evaluated three NETCONF plugins/libraries: Nornir-Netconf, NCDiff, and Scrapli.

1. **NCDiff:** Extends the ncclient NETCONF client with diff capabilities. Although it meets our requirements, its performance was insufficient for our use case.
   - **Pros:** Provides diff capabilities and extensive functionality.
   - **Cons:** Slower performance and requires loading all YANG models, which is unnecessary for our needs.
2. **Nornir-Netconf:** Offers a high-level API for interacting with network devices using NETCONF via ncclient. It fully satisfies our requirements with reliable performance.
   - **Pros:** Well-integrated with Nornir, reliable, easy to use, with performance nearly matching Scrapli.
   - **Cons:** XML formatting in replies is less refined, though this can be mitigated with an XML formatter.
3. **Scrapli_Netconf:** Provides a user-friendly implementation for NETCONF interactions using the Scrapli driver, offering superior performance with additional setup requirements.
   - **Pros:** Well-documented, easy to use, well-maintained, and the fastest among the tested plugins.
   - **Cons:** Requires additional setup with ssh2, including installing libssh2 and downgrading Python to 3.10. Also necessitates switching the devcontainer to support specific ssh2 ciphers.

Decision We selected the Nornir-Netconf plugin for its comprehensive fulfillment of our requirements and straightforward setup. Its ease of integration with our existing infrastructure made it the optimal choice for managing network automation tasks.

The full analysis is available in Appendix 1.

3.5.2. Diff Implementation Strategy

We needed to decide where to perform the diff between the current and desired network device configurations.

The Two Options:

1. **XML Diff within Nornir Automation Framework:** Perform the diff by fetching and comparing current and desired states within the Nornir framework. This allows validation before applying changes and ensures Infrahub artifacts follow the device's YANG Model NETCONF.
   - **Pros:** Easy implementation and integration, clear separation between Nornir automation and Infrahub, and uniform Infrahub artifacts.
   - **Cons:** Generating human-readable diff reports is challenging, and the diff logic struggles with namespace handling and complex structures.
2. **Model Diff within Pydantic Model:** Perform the diff earlier by generating Pydantic models from running NETCONF XML. This approach requires implementing a NETCONF to Pydantic converter and designing a suitable data structure for Infrahub artifacts.
   - **Pros:** Generates smaller, more efficient NETCONF payloads by including only changes.
   - **Cons:** Requires additional development for conversion, complex data structures for artifacts, and challenges in producing human-readable diff reports.

While the Pydantic model diff offers technical efficiency, the XML diff provided sufficient performance (4-6 seconds) and allowed us to maintain lower complexity during the initial development phase.

Decision We opted to perform the diff within the Nornir automation framework using XML diffing. This choice prioritized safety and reduced development time, enabling us to focus on integrating complex YANG models and documenting the system. Although we began implementing the Pydantic model diff as a side task, challenges in creating human-readable reports and addressing YANG model issues led us to continue with the XML diff. Future improvements may revisit the Pydantic model diff if deemed necessary.

Conclusion The decision to use XML diff in the Nornir automation framework proved correct, as integrating all components revealed an error in the OpenConfig Interfaces YANG model implementation on Cisco switches. This prompted a shift to the Cisco native interfaces YANG model, which required additional time. Although we initiated the Pydantic model diff and successfully implemented the NETCONF to Pydantic converter, generating human-readable diff reports remained challenging. Given the complexity of the Cisco native interfaces YANG model and the priority of system documentation, we continued with the XML diff approach. The Pydantic model diff will remain a potential future enhancement to further improve the system.

4. Quality Measures

This chapter outlines the quality measures implemented throughout the project to ensure high standards in code development, testing, and deployment. These measures encompass organizational practices, coding guidelines, and tools used for continuous integration and deployment.

4.1. Organizational Means

Merge Requests Merge Requests (MRs) are crucial for maintaining code quality. They combine code review with the Four-Eyes Principle, promoting collaboration, early issue detection, and protecting the stability of the main branch. This fosters a culture of shared responsibility and continuous improvement.

Definition of Done Our Definition of Done ensures that every task, feature, or deliverable meets clear standards of quality, completeness, and readiness. A task is considered done only when it satisfies the following criteria:

1. **Development:** All code is written, committed, and pushed to the repository.
2. **Code Quality:** The code has been reviewed and approved through a merge request following the Four-Eyes principle. The code passes the automated GitLab CI/CD pipeline or GitHub Actions.
3. **Documentation:** Product or project documentation is always up-to-date.
4. **Functionality:** The feature or task meets all the acceptance criteria outlined in the user story or task description.

4.2. Guidelines

4.2.1. Python - PEP8

Our Python code adheres to the PEP8 standard to ensure consistency, readability, and maintainability across the project. Following PEP8 aligns our code with industry practices, covering aspects such as indentation, naming conventions, and spacing. This standardization improves team collaboration and makes the codebase easier to understand and extend.

4.2.2. Four-Eyes Principle

Our development process strictly follows the Four-Eyes Principle to ensure code quality and integrity. This principle requires that every change be reviewed and approved by at least one other team member before being merged. It helps identify potential errors, ensures adherence to coding standards, and fosters a collaborative approach to problem-solving.

4.3. Tools Used to Assess Product Quality in CI/CD

Our continuous integration and continuous deployment (CI/CD) processes are powered by GitLab CI/CD, ensuring efficient and reliable product builds, tests, and deployments. This automation streamlines our development workflow, maintaining high standards of quality and consistency across all code repositories.

We employ a unified CI/CD pipeline logic for all repositories, utilizing GitLab CI/CD or GitHub Actions. This standardization ensures that every repository adheres to the same rule set. Our CI/CD pipeline includes the following stages:
The shared before script installs Poetry and the necessary development dependencies, which include:

- `ruff`
- `mypy`
- `bandit`
- `pytest`
- `coverage`

4.3.1. Ruff

We use Ruff to ensure that the code consistently conforms to the PEP8 standard[1]. Ruff is configured in the `project.toml` to enforce coding style guidelines, detect potential issues, and maintain uniformity.

4.3.2. MyPy

We utilize MyPy as a type checker to ensure type correctness throughout our Python codebase. By enforcing static type checking, MyPy helps detect type-related errors early in the development process, improving code reliability and reducing runtime issues. We incorporate type hints to enhance code readability and comprehension.

4.3.3. Bandit

Bandit is used to check for security vulnerabilities in our code. We have configured Bandit to check for all issues and warn in the pipeline if any are found. For example, we mitigated security issues with the `etree` library by using the `defusedxml` library in the `nornir NETCONF deployment` code.

4.3.4. Pytest and Coverage

Pytest is used to run tests in the codebase. We have written tests for the `nornir conditional runner` as this code is open-sourced. For other code, we focused on manual testing due to time constraints. Pytest generates a coverage report, which is then uploaded to GitLab or GitHub.

4.4. Manual Testing

Manual testing was essential for our project, especially when working with physical devices that had implementation errors in the OpenConfig Interfaces model. We dynamically adapted to changes, which was more efficient with a good infrastructure setup and manual tests than unit tests. Additionally, it would be difficult to just test our code without the physical devices, Infrahub and Nornir, as we do not want to test their code. A lot of mocking would be needed to test the code without the physical devices, which would not help us as we do not actually see if it works or if we run into another well-hidden bug of the surrounding systems.

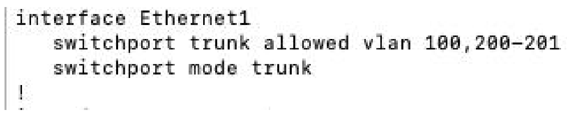
[1]PEP8 guidelines: December 11, 2024 https://peps.python.org/pep-0008/

To show this with an example: At one point, we did not know for sure if the OpenConfig model was badly implemented on our test devices, overwriting VLANs as described in chapter `Technical Issues and Obstacles` 8, or if our code was incorrect. We decided to manually test the code with different devices, as this is a feature of OpenConfig to be vendor-independent, to ensure that the code was correct and that the devices were the issue.

We tested it against a virtual Arista switch included in an example of Containerlab[2]. To set up this test, we ran Infrahub in the normal local devcontainer and the virtual Arista switch in a separate virtual Linux x86 machine on the same network, kindly provided by our advisor to minimize our setup time. We then forwarded the localhost ports of our local machine to its network address using socat[3]. This allowed us to set the correct environment variables on the virtual Linux machine running the Containerlab and start our Nornir NETCONF code loading the inventory based on the environment variables. We manually created the inventory and interface data for an Arista switch in Infrahub and ran the code. We then checked the Arista switch to see if the VLANs were not overwritten, which indicated that the OpenConfig model was not correctly implemented on the Cisco devices. The Arista switch even automatically created a VLAN range for the VLANs next to each other.

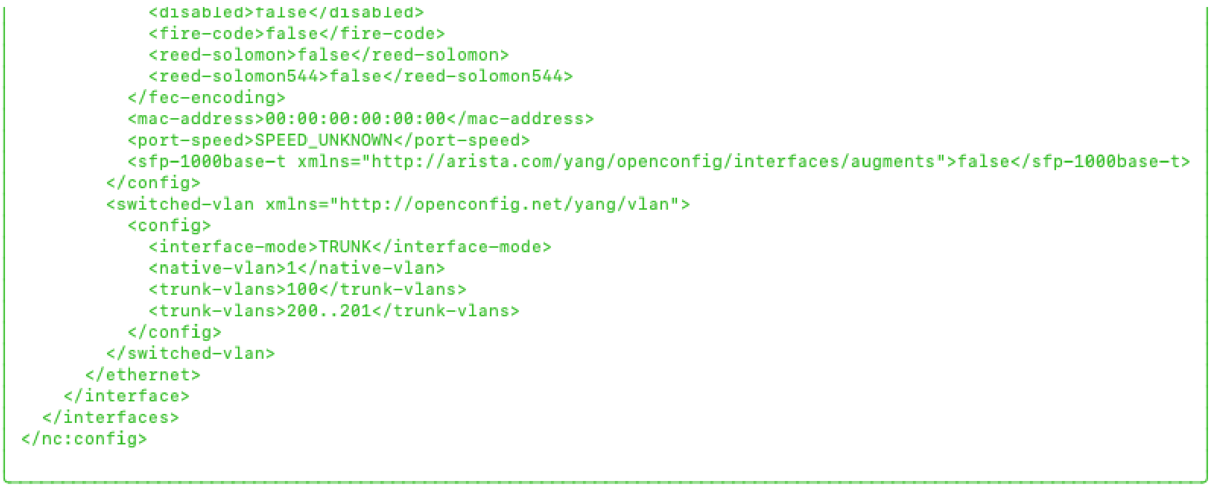
```
    <config>
        <interface-mode>TRUNK</interface-mode>
        <native-vlan>1</native-vlan>
        <trunk-vlans>100</trunk-vlans>
        <trunk-vlans>200</trunk-vlans>
        <trunk-vlans>201</trunk-vlans>
    </config>

```

Resulted in the following configuration, which you can see in Figure 4.1 and Figure 4.2.



```
interface Ethernet1
   switchport trunk allowed vlan 100,200-201
   switchport mode trunk
!
```

Figure 4.1.: Arista VLAN configuration after config deployment



```
            <disabled>false</disabled>
            <fire-code>false</fire-code>
            <reed-solomon>false</reed-solomon>
            <reed-solomon544>false</reed-solomon544>
          </fec-encoding>
          <mac-address>00:00:00:00:00:00</mac-address>
          <port-speed>SPEED_UNKNOWN</port-speed>
          <sfp-1000base-t xmlns="http://arista.com/yang/openconfig/interfaces/augments">false</sfp-1000base-t>
        </config>
        <switched-vlan xmlns="http://openconfig.net/yang/vlan">
          <config>
            <interface-mode>TRUNK</interface-mode>
            <native-vlan>1</native-vlan>
            <trunk-vlans>100</trunk-vlans>
            <trunk-vlans>200..201</trunk-vlans>
          </config>
        </switched-vlan>
      </ethernet>
    </interface>
  </interfaces>
</nc:config>
```

Figure 4.2.: Arista XML running config, showing the configuration of multiple trunk VLANs

[2]Containerlab quickstart demo: December 15, 2024 https://containerlab.dev/quickstart/
[3]socat: December 15, 2024 https://linux.die.net/man/1/socat

Way Forward: Finally, we tested if the same bug was present on a physical Cisco switch when we used the Cisco native model, as we needed a solution for the physical devices. For this, we simply changed the `defaults.yaml` to use the Cisco native model, ran the function `get_running_config`, changed the trunk VLAN list, and deployed the running config back to the switch again. This was successful, and the VLANs were not overwritten. So we had our way forward.

4.5. Code Review

We conducted code reviews at critical points in the project to ensure a comprehensive understanding of the entire project and to maintain high code quality. We utilized GitLab Merge Requests for this purpose. Once a feature was completed on a feature branch, a merge request to the main branch was created. This merge request was then reviewed by at least one other team member before being merged. At one point, when we combined our solutions, we held a meeting for a code walkthrough to ensure that everyone had a brief understanding of the entire codebase and could proceed with the integration of the components.

4.6. Conclusion

Our project adheres to a set of quality measures and guidelines to ensure that the code is of high quality, maintainable, and reliable. By following these practices, we aim to deliver a robust and scalable solution that meets the requirements of our stakeholders and our personal ones. Our continuous integration and deployment processes, along with manual testing and code reviews, help us maintain a high standard of quality throughout the development lifecycle. We are committed to delivering a product that is well-documented, thoroughly tested, and free of defects, ensuring a positive user experience and long-term success.

Our CI/CD pipeline ensures automatically that our code is always formatted, linted, checked, and deployed in a consistent manner, maintaining high code quality and security standards.

Part II.

Technical Documentation

1. Overview

The technical part of this documentation outlines the critical components, methodologies, and tools used in our project. Each section provides detailed insights into specific aspects of the system, from the deployment process to the handling of issues. Below is a summary of the topics covered:

Nornir NETCONF Deployment Details the deployment process using Nornir, focusing on its integration with NETCONF for network automation. It explains the setup, configuration, and execution of automated workflows to streamline network operations.

YANG and Pydantic Models Discusses the use of YANG models for defining network configurations and their translation into Python data structures using Pydantic. This section explains which YANG models are used and how we transformed them into the Pydantic model.

XML Exporter Covers the XML exporter function, which converts a given Pydantic model into a NETCONF-compliant XML.

Infrahub GitLab Integration Focuses on the integration with Infrahub and GitLab for version control and CI/CD pipelines. It explains how infrastructure code is managed, reviewed, and deployed, emphasizing collaboration and automation.

Nornir Conditional Runner Introduces the conditional runner feature in Nornir, allowing for flexible execution of tasks based on dynamic conditions.

Infrastructure Describes the underlying infrastructure supporting the system, including the environment setup, tooling, and resource management.

Issues and Obstacles Reflects on the challenges encountered during the development process and the solutions implemented to overcome them.

2. Nornir NETCONF Deployment

We utilize the Nornir framework, a Python-based automation tool, to deploy configurations to network devices. Nornir supports inventory management through plugins like `nornir_infrahub`, which integrates seamlessly with our single source of truth (Infrahub) for device information and enables concurrent task execution for each device. Nornir's design emphasizes simplicity and modularity, making it easy to create flexible automation using Python code. For device communication, we chose the `nornir_netconf` plugin, specifically designed for managing network devices via the Network Configuration Protocol (NETCONF) protocol. This chapter outlines the steps for deploying configurations to network devices using Nornir and NETCONF.

2.1. Nornir NETCONF Tasks

We structured the code into four callable functions (`get_running_config()`, `desired_state()`, `validate()`, and `deploy()`) in the main program, each leveraging underlying logical components. For example, the `validate()` and `deploy()` functions essentially share the same logic but are executed with different values for Nornir's `dry-run` variable. This distinction allows `validate()` to serve as a safety precaution, keeping the network administrator informed and involved in the process, rather than fully automating configuration changes without oversight. By using `validate()`, administrators can observe the tool's behavior and gain confidence in its accuracy before transitioning to the `deploy()` function, trusting it to configure the desired state as defined in Infrahub.

2.1.1. Retrieving Configuration

`get_config_by_filter(task: Task, from_running: bool = True) -> Result:`

Before deploying configurations, we first need to retrieve the current configuration from the network devices based on the selected YANG model. This step is essential to ensure that devices are in a consistent state and that configurations are up to date. To perform this task, we use the `nornir_netconf` plugin, which provides a straightforward interface for connecting to devices and retrieving configuration data. Since we utilize two different YANG models, we created a generic function that retrieves configuration data for all the specified models and filters, consolidating the data into a single XML NETCONF file per device.

This is done using the configuration in the `defaults.yaml` file, which stores the YANG model and the filters in a YAML file.

Example of the configuration in the `defaults.yaml` file: OpenConfig-interfaces and OpenConfig-vlan are used

```
yang:
  - interface:
    filter: |
      <interfaces xmlns="http://openconfig.net/yang/interfaces">
        <interface>
          <config>
          </config>
          <ethernet xmlns="http://openconfig.net/yang/interfaces/ethernet">
            <switched-vlan xmlns="http://openconfig.net/yang/vlan">
              <config>
              </config>
            </switched-vlan>
          </ethernet>
          <routed-vlan xmlns="http://openconfig.net/yang/vlan">
          </routed-vlan>
        </interface>
      </interfaces>
    ids:
      - tag: "{http://openconfig.net/yang/interfaces}interface"
        id_tag: "{http://openconfig.net/yang/interfaces}name"
    replace:
      - <routed-vlan xmlns="http://openconfig.net/yang/vlan">
      - <config>
  - vlan:
    filter: |
      <vlans xmlns="http://openconfig.net/yang/vlan">
      </vlans>
    ids:
      - tag: "{http://openconfig.net/yang/vlan}vlan"
        id_tag: "{http://openconfig.net/yang/vlan}vlan-id"
    replace:
      - <vlans xmlns="http://openconfig.net/yang/vlan">
```

Since the data we retrieve from the devices can be quite large and can easily fill the entire vertical space of any terminal, we implemented a `debug` parameter which writes the NETCONF XML to a file. Later, we modified it to create a Prefect artifact as well, which is even easier for the user to see the output of the NETCONF XML as it is presented as a markdown artifact inside the actual task.

2.1.2. Desired State

`desired_state(task: Task) -> Result:`

To obtain the desired state, we use the Infrahub SDK to retrieve the configuration from an Infrahub artifact. During development, we utilized a local file to store the desired state because the `python_transformations`, the Pydantic model, and the `xml_exporter` for Pydantic were still under development.

2.1.3. Validate or Deploy Configuration

`validate_or_deploy_config(task: Task) -> Result:`

After retrieving the configuration data, we validate it against the desired configuration to identify any discrepancies. This step ensures that configurations are consistent across all devices and adhere to the intended state. To detect deviations, we created a `diff()` function that compares the current configuration with the desired state, generating a report that highlights any differences. This report informs the user of any discrepancies.

If differences are detected, we apply the configuration to the device's candidate store for further validation. To achieve this, we added specific NETCONF replace operations to certain tags of the desired configuration. Replacing the entire configuration was not feasible because the OpenConfig-interfaces model does not support the `replace` operation for `interfaces`, and we only wanted to replace a small portion of the interface while leaving the `ip` configuration untouched. This adjustment allows for greater flexibility by enabling the addition of tag containers to be replaced alongside the filters within the tool configuration `defaults.yaml`, which is handled by the function `add_replace_operations()`.

Once the device validates the configuration, a report is generated and sent to the user. The user reviews the report and confirms the changes. If the user is satisfied with the validation results, the configuration is committed to the device by rerunning the same function with the `dry-run` option set to `false`.

2.1.4. Diff Functions

`diff(task: Task, first_xml: str, second_xml: str) -> Result:`

Comparing two XML files turned out to be more challenging than initially estimated. We initially intended to use the `xml-diff` library, but its output was not very human-readable, as shown in figure 2.1. Therefore, we decided to write our own `diff()` function.

The `diff()` function takes two XML files as input and returns a list of differences. It is based on the `xml.etree.ElementTree` library (later swapped with `defusedxml` for security reasons) and recursively traverses the XML tree to find differences. The function compares the tag, text, and attributes of each element and appends the differences to a list. It also handles cases where elements are in a different order, generating a human-readable report that highlights the differences between the two configurations.

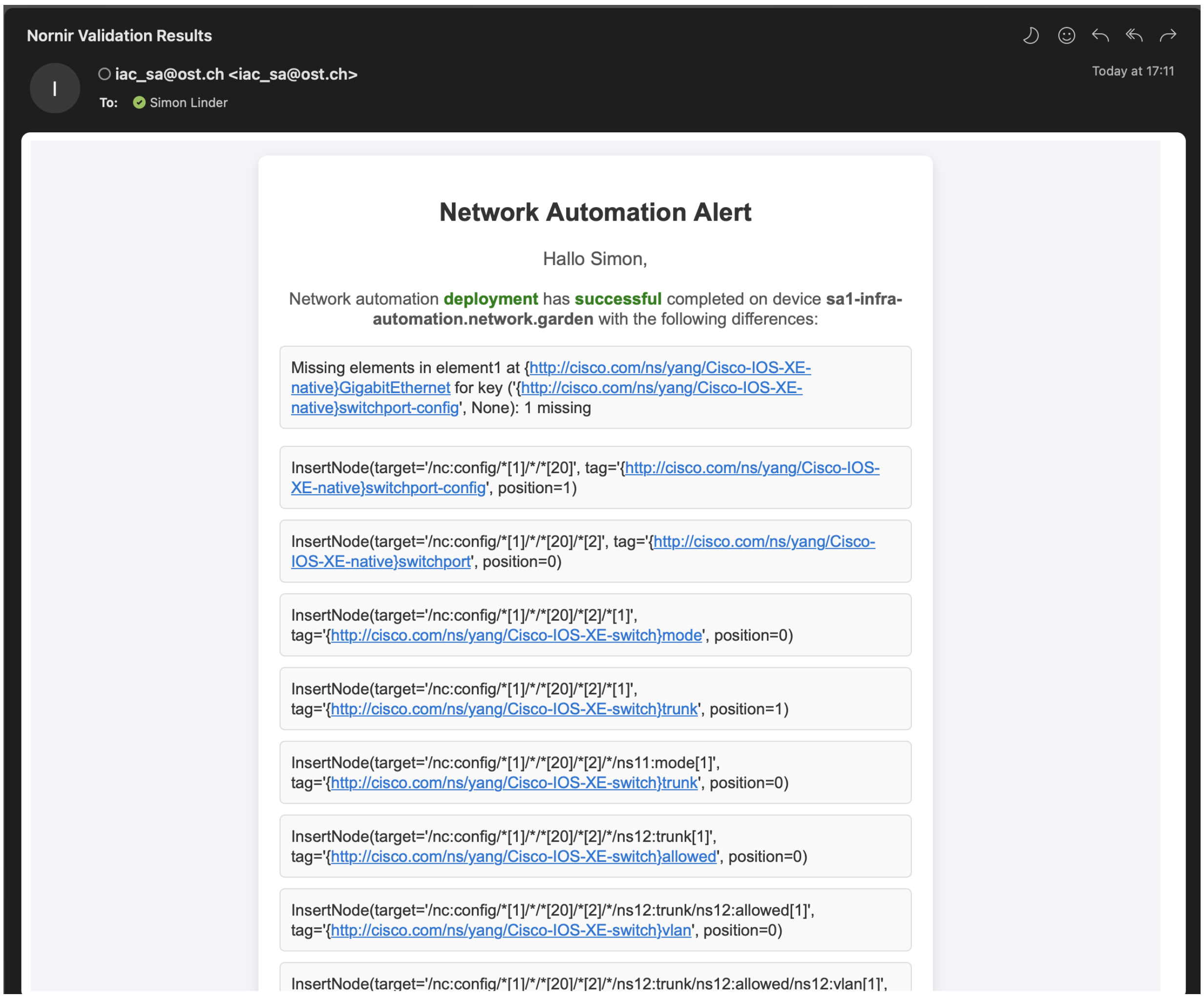
Both the `diff()` function and `xml-diff` use special identifiers to locate the nodes to compare. These identifiers are stored in the `defaults.yaml` file next to the corresponding filter. Both functions are implemented and can be enabled or disabled using start parameters. They can even be used in combination. By default, the self-developed `diff()` function is used, but this can be configured in the `defaults.yaml` file.

```yaml
# config
xmldiff: false
etreediff: true
ncdiff: false
debug: false
```

2.1.5. NcDiff

`validate_or_deploy_config(task: Task) -> Result:` in `nr_tasks_ncdiff.py`

During our evaluation of connection plugins, we initially implemented the diff function using NcDiff due to its straightforward approach. However, we found NcDiff to be too slow for our needs as it downloads all YANG models before performing the diff. Despite its slower performance, we retained NcDiff as a fallback option to mitigate the risk of bugs in critical components. Implementing the same logic in a different plugin was straightforward. NcDiff follows the same logic but uses a different connection plugin and diff mechanism. For the diff, it always loads the configuration into the candidate store and downloads it again to compare it with the running configuration. It can be enabled or disabled using start parameters.



Nornir Validation Results

iac_sa@ost.ch <iac_sa@ost.ch>
To: Simon Linder
Today at 17:11

Network Automation Alert

Hallo Simon,

Network automation **deployment** has **successful** completed on device **sa1-infra-automation.network.garden** with the following differences:

Missing elements in element1 at {http://cisco.com/ns/yang/Cisco-IOS-XE-native}GigabitEthernet for key ('{http://cisco.com/ns/yang/Cisco-IOS-XE-native}switchport-config', None): 1 missing

InsertNode(target='/nc:config/*[1]/*/*[20]', tag='{http://cisco.com/ns/yang/Cisco-IOS-XE-native}switchport-config', position=1)

InsertNode(target='/nc:config/*[1]/*/*[20]/*[2]', tag='{http://cisco.com/ns/yang/Cisco-IOS-XE-native}switchport', position=0)

InsertNode(target='/nc:config/*[1]/*/*[20]/*[2]/*[1]', tag='{http://cisco.com/ns/yang/Cisco-IOS-XE-switch}mode', position=0)

InsertNode(target='/nc:config/*[1]/*/*[20]/*[2]/*[1]', tag='{http://cisco.com/ns/yang/Cisco-IOS-XE-switch}trunk', position=1)

InsertNode(target='/nc:config/*[1]/*/*[20]/*[2]/*/ns11:mode[1]', tag='{http://cisco.com/ns/yang/Cisco-IOS-XE-switch}trunk', position=0)

InsertNode(target='/nc:config/*[1]/*/*[20]/*[2]/*/ns12:trunk[1]', tag='{http://cisco.com/ns/yang/Cisco-IOS-XE-switch}allowed', position=0)

InsertNode(target='/nc:config/*[1]/*/*[20]/*[2]/*/ns12:trunk/ns12:allowed[1]', tag='{http://cisco.com/ns/yang/Cisco-IOS-XE-switch}vlan', position=0)

InsertNode(target='/nc:config/*[1]/*/*[20]/*[2]/*/ns12:trunk/ns12:allowed/ns12:vlan[1]',

Figure 2.1.: Mail showcasing one etree diff change box followed by many xml-diff boxes representing the same change

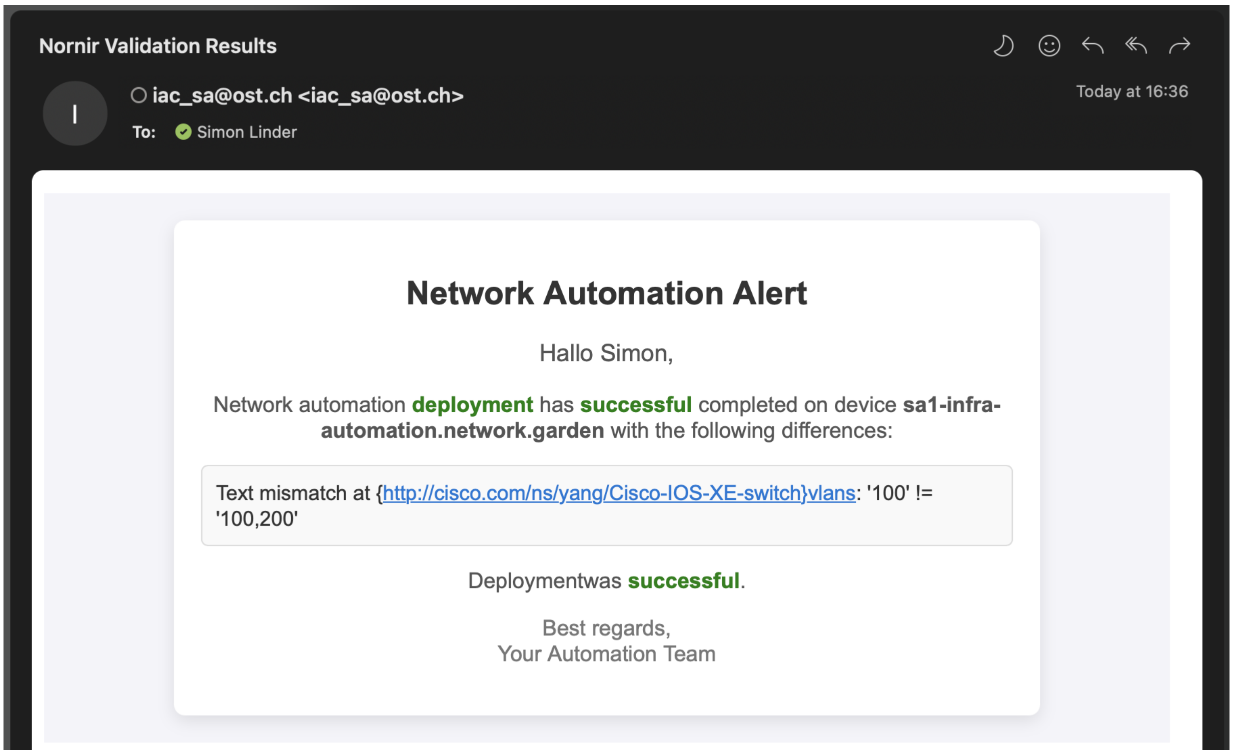
2.1.6. Mail

`send_email(task: Task, device_name: str, is_successful: bool, etreediff: Union[List[str], None] = None, xmldiff: Union[List[str], None] = None, ncdiff: Union[str, None] = None,) -> Result:`

To notify the user of the deployment results, we implemented several email functions. While text-based emails would have sufficed, we opted for HTML emails for a better user experience. Two examples are shown in figure 2.2 and figure 2.3. The email is sent to the user after the deployment or validation is completed and includes a summary of the deployment status and the diff report.

The configuration for the mail settings can be found in the `defaults.yaml` file.

```yaml
mail:
    smtp_server: "smtp.ost.ch"
    port: 25
    sender: "iac_sa@ost.ch"
    to: "Simon.linder@ost.ch"
```



Nornir Validation Results

I iac_sa@ost.ch <iac_sa@ost.ch>
To: Simon Linder
Today at 16:36

Network Automation Alert

Hallo Simon,

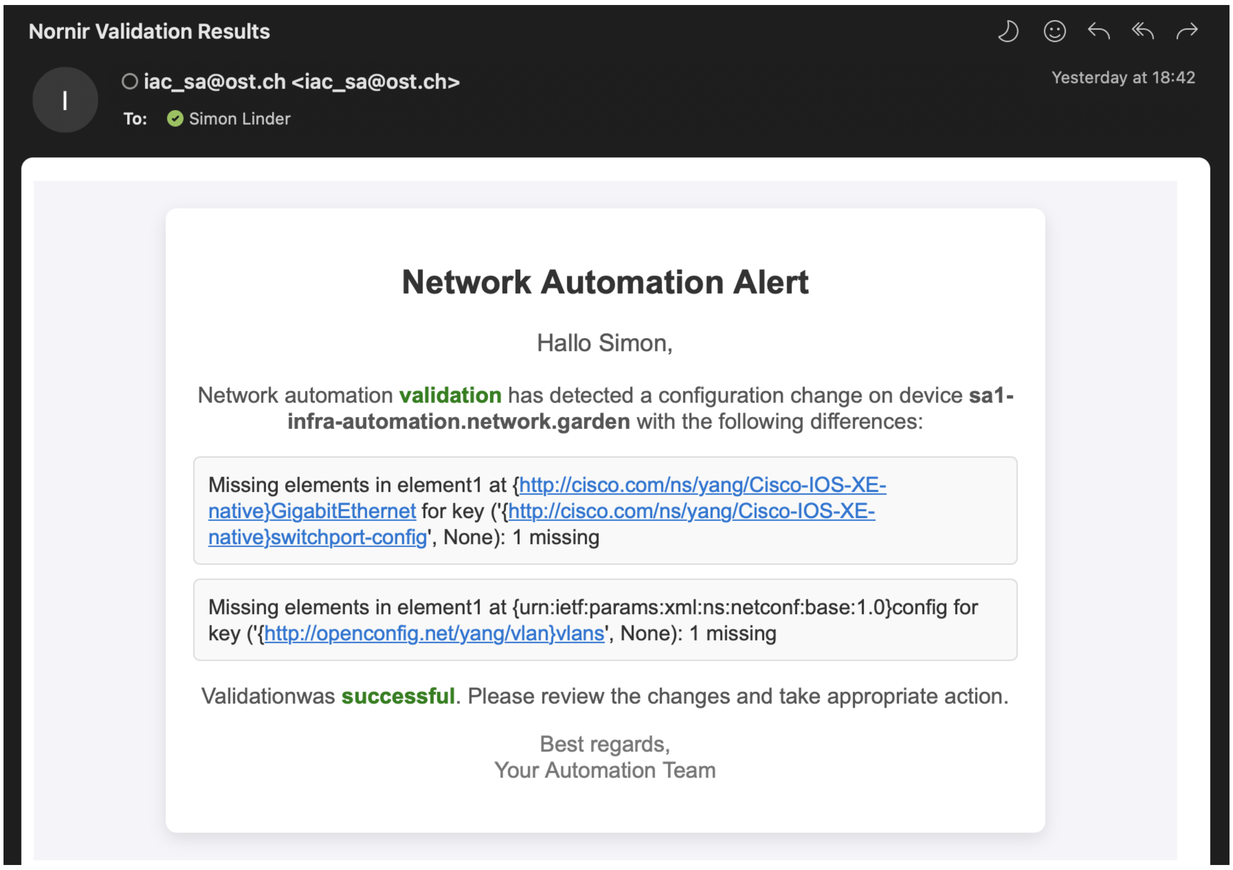
Network automation **deployment** has **successful** completed on device **sa1-infra-automation.network.garden** with the following differences:

Text mismatch at {http://cisco.com/ns/yang/Cisco-IOS-XE-switch}vlans: '100' != '100,200'

Deploymentwas **successful**.

Best regards,
Your Automation Team

Figure 2.2.: Deploy mail showcasing the removal of a trunk VLAN



Nornir Validation Results

I iac_sa@ost.ch <iac_sa@ost.ch>
To: Simon Linder
Yesterday at 18:42

Network Automation Alert

Hallo Simon,

Network automation **validation** has detected a configuration change on device **sa1-infra-automation.network.garden** with the following differences:

Missing elements in element1 at {http://cisco.com/ns/yang/Cisco-IOS-XE-native}GigabitEthernet for key ('{http://cisco.com/ns/yang/Cisco-IOS-XE-native}switchport-config', None): 1 missing

Missing elements in element1 at {urn:ietf:params:xml:ns:netconf:base:1.0}config for key ('{http://openconfig.net/yang/vlan}vlans', None): 1 missing

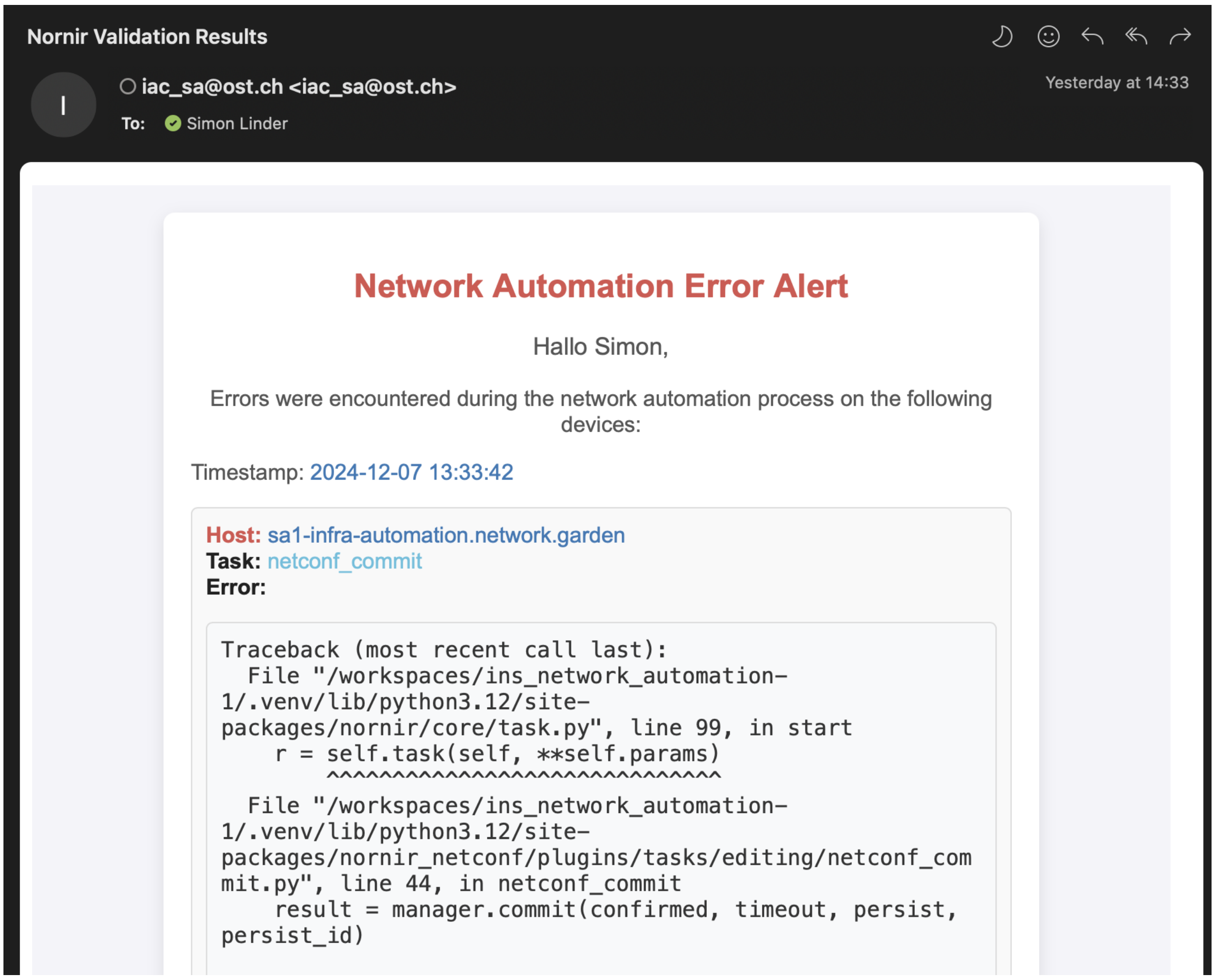
Validationwas **successful**. Please review the changes and take appropriate action.

Best regards,
Your Automation Team

Figure 2.3.: Validate Mail showcasing one missing <vlan> and <switchport-config> tags

To prevent flooding the user with emails, the mail is only sent if there is a diff, which should only occur if someone changes the desired state in Infrahub or makes changes on the device.

Although we anticipate that it won't be frequently used once the code runs smoothly, we implemented a `send_error_email(mail: Dict[str, Union[str, int]], result: Result) -> None:` function to notify users or developers during the development phase if any tasks fail. This function sends an email containing the error message and its traceback. It is invoked within the main function to alert the user when an issue arises anywhere in the code. The email is sent to the recipient specified in the `defaults.yaml` file. Since the email is directed to the network administrator, the traceback is included for detailed debugging.



Network Automation Error Alert

Hallo Simon,

Errors were encountered during the network automation process on the following devices:

Timestamp: 2024-12-07 13:33:42

Host: sa1-infra-automation.network.garden
Task: netconf_commit
Error:

```
Traceback (most recent call last):
  File "/workspaces/ins_network_automation-
1/.venv/lib/python3.12/site-
packages/nornir/core/task.py", line 99, in start
    r = self.task(self, **self.params)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/workspaces/ins_network_automation-
1/.venv/lib/python3.12/site-
packages/nornir_netconf/plugins/tasks/editing/netconf_com
mit.py", line 44, in netconf_commit
    result = manager.commit(confirmed, timeout, persist,
persist_id)
```

Figure 2.4.: Error mail with traceback

2.2. Making the Code Generic for Different YANG Models

We aimed to make the code as generic as possible. Consequently, we utilized the `defaults.yaml` file of Nornir as the configuration file for the entire tool. This approach allows for future integration into the Infrahub artifact, enabling communication with various devices like Arista, Nokia, and Cisco simultaneously, each with their own YANG models. However, since the INS is primarily focused on Cisco Catalyst devices, we concentrated on finding models suitable for these targets and only configured them in the `defaults.yaml`.

The `defaults.yaml` file includes the YANG model used as a `filter` element, allowing us to extract only the relevant sections of the device configuration needed for our operations.

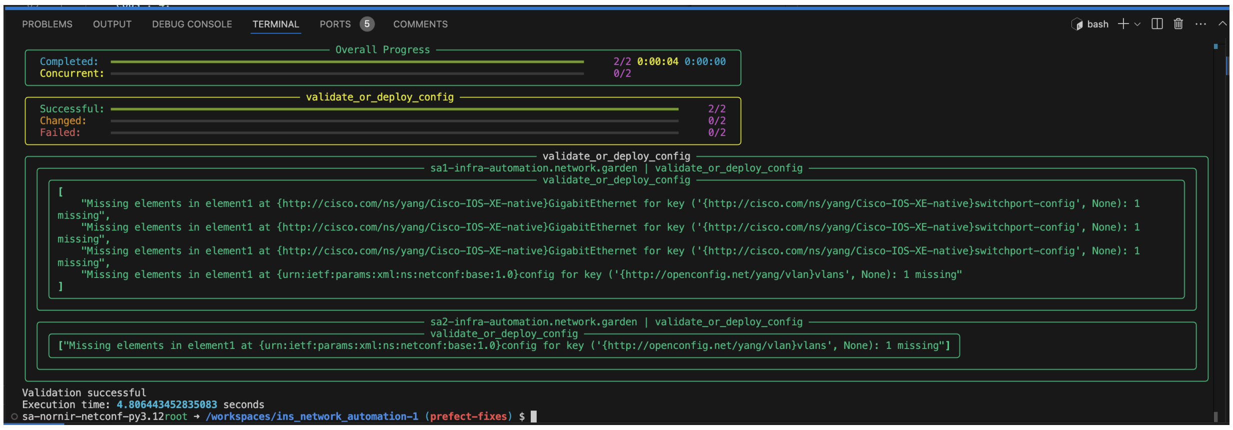
Additionally, the tool requires the `ids` of the YANG containers essential for the diff function. This ensures that the container *GigabitEthernet1/0/1* is always compared with its counterpart *GigabitEthernet1/0/1* in the running configuration, maintaining consistency in the comparisons.

To enhance customizability and performance, we introduced a `replace` list containing the XML tags of the containers where configuration replacements are required. This decision addresses limitations in certain YANG models, as not all of them support the replace function at the top-level element. For instance, the *openconfig-interface* model on Cisco switches, or the *cisco-native* model, becomes inefficient and generates excessively large configurations when attempting to replace all elements across containers.

The `defaults.yaml` structure we designed enables us to work with multiple YANG models simultaneously. This includes handling multiple `ids` within a single model (e.g., in the case of *cisco-native*) and specifying highly targeted configuration snippets containing only the desired elements for replacement.

2.3. Typer CLI GUI

Initially, we utilized the Typer CLI for interacting with the tool. This command-line interface allowed us to execute the tool with various parameters and provided an entry point, such as `iac_sa validate`, to run specific functions. The Typer app context was particularly useful for initializing the Nornir inventory with the Infrahub SDK and global variables like the `debug` option.



```
                         Overall Progress
  Completed:                                                   2/2 0:00:04 0:00:00
  Concurrent:                                                  0/2

                     validate_or_deploy_config
  Successful:                                                          2/2
  Changed:                                                             0/2
  Failed:                                                              0/2

                                     validate_or_deploy_config
                  sa1-infra-automation.network.garden | validate_or_deploy_config
                                     validate_or_deploy_config
  [
      "Missing elements in element1 at {http://cisco.com/ns/yang/Cisco-IOS-XE-native}GigabitEthernet for key ('{http://cisco.com/ns/yang/Cisco-IOS-XE-native}switchport-config', None): 1
  missing",
      "Missing elements in element1 at {http://cisco.com/ns/yang/Cisco-IOS-XE-native}GigabitEthernet for key ('{http://cisco.com/ns/yang/Cisco-IOS-XE-native}switchport-config', None): 1
  missing",
      "Missing elements in element1 at {http://cisco.com/ns/yang/Cisco-IOS-XE-native}GigabitEthernet for key ('{http://cisco.com/ns/yang/Cisco-IOS-XE-native}switchport-config', None): 1
  missing",
      "Missing elements in element1 at {urn:ietf:params:xml:ns:netconf:base:1.0}config for key ('{http://openconfig.net/yang/vlan}vlans', None): 1 missing"
  ]

                  sa2-infra-automation.network.garden | validate_or_deploy_config
                  validate_or_deploy_config
  ["Missing elements in element1 at {urn:ietf:params:xml:ns:netconf:base:1.0}config for key ('{http://openconfig.net/yang/vlan}vlans', None): 1 missing"]

Validation successful
Execution time: 4.806443452835083 seconds
○ sa-nornir-netconf-py3.12root → /workspaces/ins_network_automation-1 (prefect-fixes) $
```

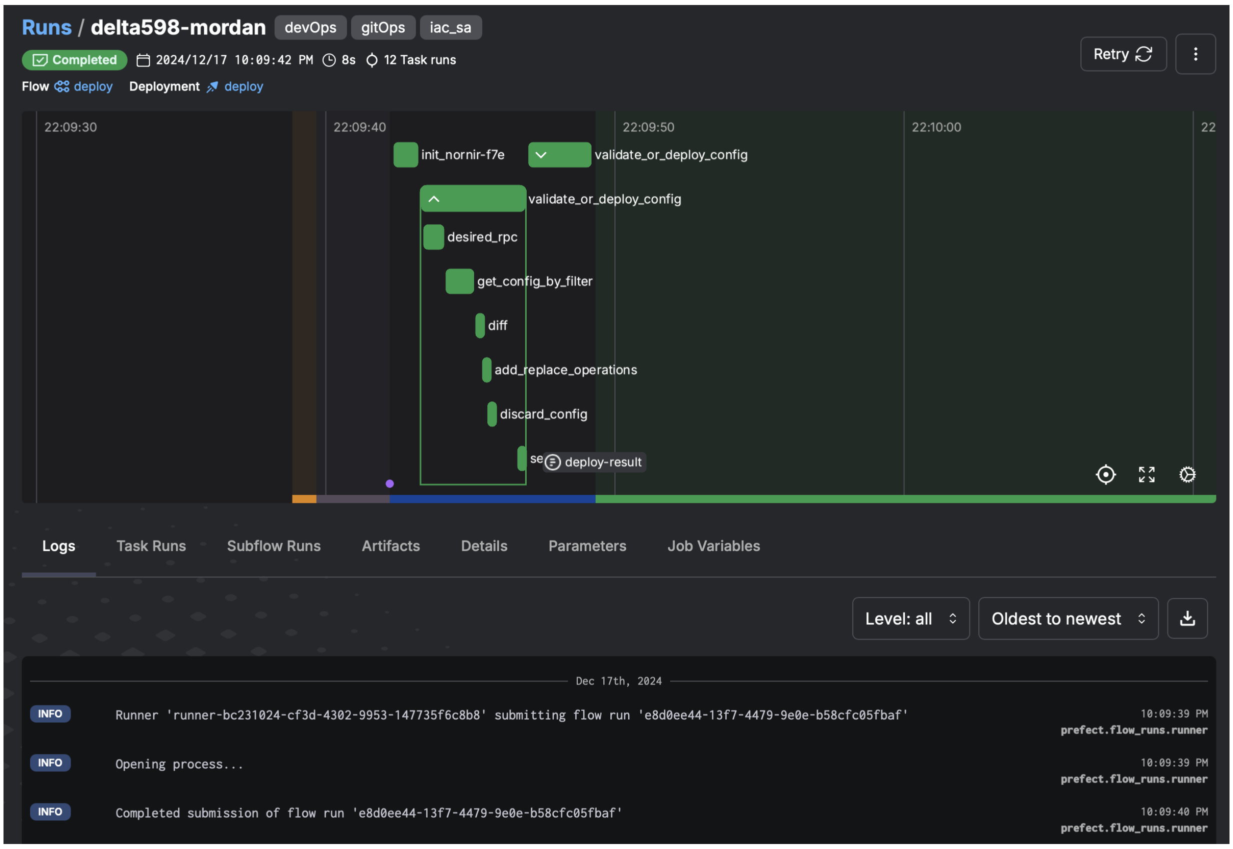
Figure 2.5.: Typer CLI GUI

Subsequently, we transitioned to using the Prefect GUI due to its user-friendly interface and scheduling capabilities for the `validate()` function. While the Typer CLI remains available for command-line interactions and serves as an alternative GUI, the Prefect GUI is now the recommended method for interacting with the tool.

2.4. Prefect

To execute our Nornir NETCONF tool, we utilize Prefect, a Python-based workflow management system. Prefect allows us to define, schedule, and monitor complex workflows with ease. Its intuitive interface simplifies the creation of workflows by enabling us to define tasks, dependencies, and schedules directly from our Python code using annotations on the functions.

Figure 2.6 illustrates a deployment run using Prefect, showcasing its user-friendly interface and robust capabilities. Each device gets its own task, which can be opened to see the sub-task of the main task, which is named the same as the device, by utilizing the parameter `task_run_name='{task.host.name}'` in the function decorator `@task()`. Additionally, the parameter `log_prints=True` is used to print the output of the task to the Prefect log.



```
Dec 17th, 2024
INFO  Runner 'runner-bc231024-cf3d-4302-9953-147735f6c8b8' submitting flow run 'e8d0ee44-13f7-4479-9e0e-b58cfc05fbaf'   10:09:39 PM
                                                                                                          prefect.flow_runs.runner
INFO  Opening process...                                                                                   10:09:39 PM
                                                                                                          prefect.flow_runs.runner
INFO  Completed submission of flow run 'e8d0ee44-13f7-4479-9e0e-b58cfc05fbaf'                              10:09:40 PM
                                                                                                          prefect.flow_runs.runner
```

Figure 2.6.: Prefect deploy run showing the subtasks of one device followed by the second device

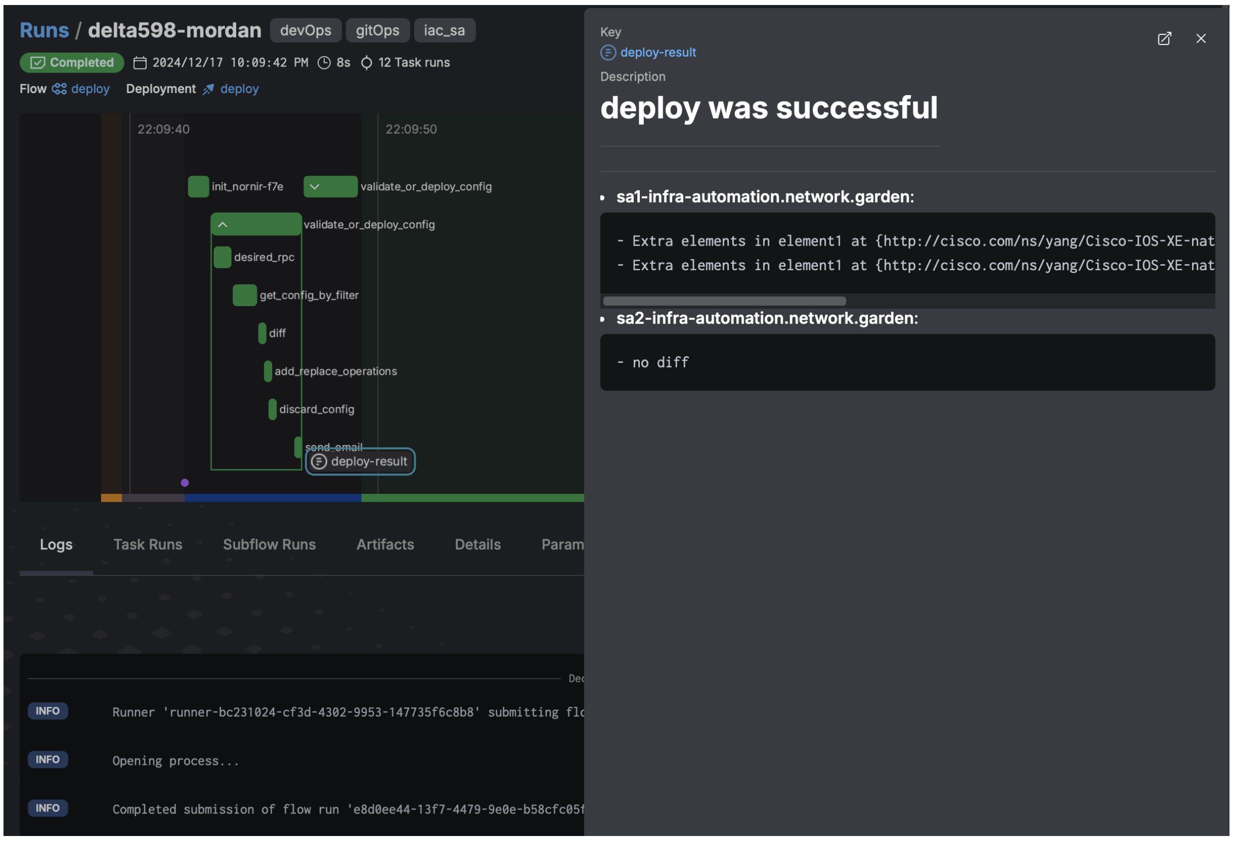
We also utilize Prefect to orchestrate the execution of our tool, offering comprehensive monitoring and logging capabilities to track the progress of configuration deployments and troubleshoot any issues that arise. Prefect's artifact feature allows us to provide detailed results after each task, facilitating easy exploration and analysis. Additionally, Prefect's user-friendly interface simplifies the management and monitoring of our workflows, enhancing overall efficiency.

Figure 2.7 illustrates the Prefect artifact generated during a deployment task, showcasing the detailed information available for analysis.

These Prefect artifacts are markdown files created by the function `create_artifact(result: Result, jobname: str, is_xml: bool) -> None:` which we wrote to convert any Nornir result we get into a markdown file, which was challenging as we wanted a nice and clean output.

By leveraging Prefect, we can automate the configuration deployment process, ensuring consistent configurations across all devices. Prefect's intuitive interface supports the scheduling of recurring jobs, aligning with DevOps principles of continuous integration and continuous deployment (CI/CD). This approach facilitates Infrastructure as Code (IaC) by pulling the latest code from the main branch into a Prefect worker. The worker operates within a custom Docker container that includes all necessary Python dependencies, ensuring seamless and efficient execution of the deployment process while maintaining a clear separation between code and infrastructure.

Figure 2.8 illustrates a Prefect artifact generated during a validation task, showcasing the detailed information available for analysis.



```
INFO    Runner 'runner-bc231024-cf3d-4302-9953-147735f6c8b8' submitting flo
INFO    Opening process...
INFO    Completed submission of flow run 'e8d0ee44-13f7-4479-9e0e-b58cfc05f
```

deploy was successful

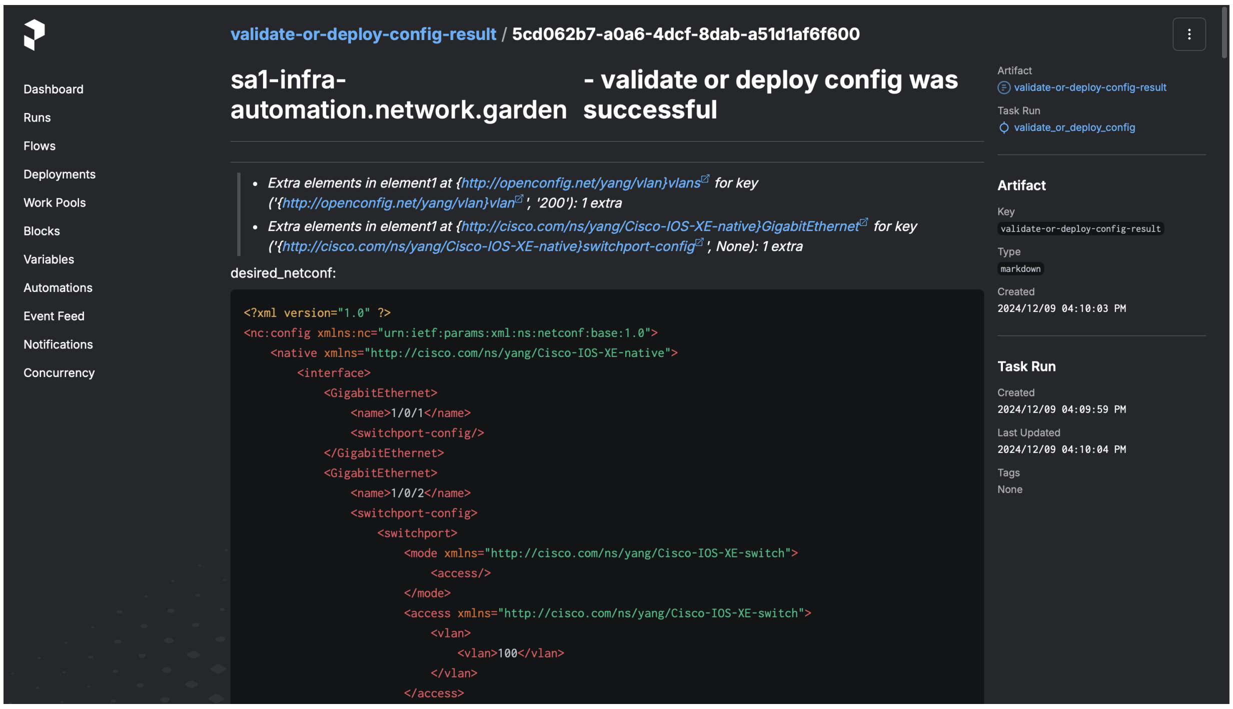
- **sa1-infra-automation.network.garden:**

```
- Extra elements in element1 at {http://cisco.com/ns/yang/Cisco-IOS-XE-nat
- Extra elements in element1 at {http://cisco.com/ns/yang/Cisco-IOS-XE-nat
```

- **sa2-infra-automation.network.garden:**

```
- no diff
```

Figure 2.7.: Prefect deploy artifact showing one device with two changes



validate-or-deploy-config-result / 5cd062b7-a0a6-4dcf-8dab-a51d1af6f600

sa1-infra-automation.network.garden - validate or deploy config was successful

- *Extra elements in element1 at {http://openconfig.net/yang/vlan}vlans for key ('{http://openconfig.net/yang/vlan}vlan', '200'): 1 extra*
- *Extra elements in element1 at {http://cisco.com/ns/yang/Cisco-IOS-XE-native}GigabitEthernet for key ('{http://cisco.com/ns/yang/Cisco-IOS-XE-native}switchport-config', None): 1 extra*

desired_netconf:

```
<?xml version="1.0" ?>
<nc:config xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
    <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
        <interface>
            <GigabitEthernet>
                <name>1/0/1</name>
                <switchport-config/>
            </GigabitEthernet>
            <GigabitEthernet>
                <name>1/0/2</name>
                <switchport-config>
                    <switchport>
                        <mode xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-switch">
                            <access/>
                        </mode>
                        <access xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-switch">
                            <vlan>
                                <vlan>100</vlan>
                            </vlan>
                        </access>
```

Artifact
validate-or-deploy-config-result
Task Run
validate_or_deploy_config

Artifact
Key
validate-or-deploy-config-result
Type
markdown
Created
2024/12/09 04:10:03 PM

Task Run
Created
2024/12/09 04:09:59 PM
Last Updated
2024/12/09 04:10:04 PM
Tags
None

Figure 2.8.: Prefect artifact of the validate task

2.5. Conclusion of the Nornir NETCONF Deployment Component

The Nornir NETCONF deployment tool provides a robust and flexible solution for managing network device configurations. By leveraging the Nornir framework and the NETCONF protocol, we can automate configuration deployments, ensuring consistency and accuracy across all devices. The tool's modular design and extensible architecture enable us to work with multiple YANG models simultaneously, providing a versatile solution for managing network configurations. By integrating with Prefect, we can orchestrate the deployment process, schedule recurring jobs, and monitor the progress of configuration deployments. This solution aligns with DevOps principles, enabling Infrastructure as Code (IaC) and supporting continuous integration and continuous deployment (CI/CD) practices. Overall, the Nornir NETCONF deployment tool offers a powerful and efficient solution for managing network device configurations and automating the deployment process, and I am quite proud of the modularity and flexibility of the codebase. Additionally, exploring Prefect was a great experience.

3. YANG and Pydantic Models

YANG models define the data structure for communication between the application and network devices. Pydantic models, which are Python classes, represent these YANG models in a Python-friendly format. Using the Pydantify tool, we generate Pydantic models from YANG models. These Pydantic models validate the data sent to network devices and are used to generate the corresponding XML for Netconf protocol communication.

3.1. YANG Models

We leverage YANG models to define the structure of both the Pydantic model and the NETCONF XML. YANG is a widely adopted language designed to structure data in an intuitive and organized way. Many network vendors provide their own YANG models tailored to their specific devices, while the OpenConfig YANG model serves as a vendor-neutral standard, creating a unified framework for all network devices.

For our project:

- **OpenConfig YANG Model:** Used for the VLANs section, ensuring broad compatibility across different devices.
- **Cisco Native YANG Model:** Selected for the interfaces section to meet a key customer requirement—managing Cisco switches effectively.

This combination was chosen to balance compatibility and address specific customer needs as well as challenges encountered during development. For more details, please refer to the chapter "Issues and Obstacles."

To facilitate the transformation from YANG models to Pydantic structures, we use the Python tool "Pydantify".

3.1.1. Cisco Native YANG Model - Interface

To meet the specific requirements of our project and ensure compatibility with Cisco switches, several modifications were made to the Cisco Native YANG Model for interfaces. These adjustments were necessary to align the model with the structure expected by the switches and to simplify the Pydantic model generation. Below are the changes implemented:

1. **Remove Unnecessary Parts of the YANG Model:** This reduction was made because Pydantic structures are generated for every grouping in the YANG model. By removing unused groupings, we streamlined the structure, focusing on the required data and making the Pydantic models more concise. We only need the `interface-switchport-grouping`.
2. **Simplify the Selected Grouping:** This ensured that only the necessary fields were included, optimizing the structure and avoiding irrelevant elements. We only need the following Containers

```
  container switchport-config {
    container switchport {
      container mode {...}
      container access {
        container vlan {...}
      }
      container trunk {
        container allowed {
          container vlan {...}
        }
      }
      container native {...}
    }
  }
```

3. **Add a vlans Leaf Inside the vlan Container:** This change was required because Cisco switches include an additional vlans container after the vlan container in their configuration.

```
    container trunk {
        container allowed {
            container vlan {
                leaf vlans {!Allow List}
            }
        }
    }
```

4. **Convert the mode Leaf to a Container and set the value to empty:** This adjustment was necessary because Cisco switches use closed tags (containers) for these values rather than enumerated values.

```
    container mode {
        description
            "Set trunking mode of the interface";
        leaf access {
            type empty;
          }

        leaf trunk {
            type empty;
        }
    }
```

5. **Rename the switchport-wrapper Container to switchport-config:** This change was required because Cisco switches only accept switchport-config as the tag name.

```
    grouping interface-switchport-grouping {
        container switchport-config {...}
    }
```

3.1.2. Openconfig YANG Model - vlans

We utilize the OpenConfig YANG Model for the VLAN definition. Unlike the Cisco Native YANG Model, the OpenConfig model was used without any modifications, ensuring a standard and vendor-neutral implementation.

3.2. Pydantic Model

The two Pydantic models generated from the YANG models using Pydantify required additional customization to meet the project's requirements and streamline their structure:

1. **Modified Aliases for Containers After Switchport:**
   - This adjustment ensured the correct namespace was applied. By default, all containers in the interface model were assigned the "cisco-native" namespace due to their origin in the same YANG file. By modifying the aliases, the desired namespaces were correctly created.
2. **Removed Overlapping Containers and Adjusted References:**
   - Pydantify generated duplicate containers for each interface type, leading to redundancy and requiring manual intervention to input data into the model.
   - To resolve this, overlapping containers were removed, and their references were updated. Since all interface types shared identical containers and leaves, this adjustment significantly streamlined the structure.
3. **Merged the Two Files into a Single Model Class:**
   - This integration simplified usage by providing a central structure for interacting with the data and avoided the need to manage multiple files or classes.

4. XML Exporter

This chapter details the configuration of the Python class XMLExporter. To bridge the gap between the data in Infrahub and the Nornir plugin, we aim to load the data into a Pydantic model and subsequently transform it into a NETCONF-compliant XML format. Our primary objective was to design the exporter to be as generic as possible, ensuring that no modifications would be required if changes were made to the Pydantic model in the future.

4.1. Python Libraries

lxml The lxml library is utilized because it provides advanced XML handling features, such as the `getparent()` method. This function is essential for validating the presence of a namespace in one of the parent elements of the XML tree. The necessity of this Python library is discussed in the "Issues and Obstacles" chapter.

pydantic The Pydantic library is employed to ensure reliable data validation and parsing. Pydantic enables the definition of structured models with strict type enforcement, ensuring that the data used to generate XML is accurate and conforms to the expected schema. This guarantees consistency in the input data, reduces errors, and simplifies the transformation process. Its robust validation capabilities make Pydantic a critical component of the workflow.

Enum The Enum module is used to define and manage enumerations, providing a set of symbolic names bound to unique, constant values. Enums ensure clarity and consistency in the code by restricting values to predefined options, thereby reducing the risk of invalid inputs. This enhances the robustness, readability, and maintainability of the code, especially when dealing with fixed sets of choices or configurations.

4.2. XMLModelConverter

4.2.1. Static Methods

Only static methods are employed in this code as it operates without the need for object instances. This approach simplifies the design, tying all methods to the class itself rather than to individual objects.

4.2.2. Pydantic Data Types

Various data types are used in the code, such as List, Enums, Strings, Integers, etc. This document does not delve further into these general data types. Additionally, the new data types from Pydantic, "BaseModel" and "RootModel," are utilized.

BaseModel The "BaseModel" type from Pydantic refers to any model from Pydantic. In the current use case and to clarify with the YANG Model Types, BaseModels include:

- **Container**
- Leafs

RootModel The "RootModel" type from Pydantic is a subcategory and a more specific data type within Pydantic. It is used for Leafs and the end of the model. To simplify understanding with the YANG Model Types, Root Models include:

- **Leafs**

4.2.3. Public Function - to XML

The "to xml" function converts a Python model (of type BaseModel) into a NETCONF-compatible XML document. An XML structure with a specific namespace is generated, and the model's structure is mapped into corresponding XML tags. The code operates as follows:

1. **Creating the XML Tree:** The function begins by creating an XML tree with a root element `<config>` and a specified namespace.

```
    root = etree.Element("{urn:ietf:params:xml:ns:netconf:base:1.0}config")
```

2. **Recursive Rendering:** The function recursively iterates over the Pydantic model to determine whether a new XML tag needs to be created, if it is a list, or if an existing tag's value should be set.

```
    if isinstance(model, BaseModel):
        for modelkey in model.model_fields:
            if isinstance(model, RootModel):
                # Call function renderrootmodel (Point 3 - Recursive Root
Rendering)
            if isinstance(getattr(model, modelkey), list):
                # Create new XML Tag for every list entry - Namespace Selection
is described in Chapter 4.2.4
                # Call same Function with every list entry
            if isinstance(getattr(model, modelkey), BaseModel):
                # Create new XML Tag - Namespace Selection is described in
Chapter 4.2.4
                # Call same Function with next Element in the Model
    )
    return roottree
```

3. **Recursive Root Rendering** Upon reaching the end of a model, the function iterates over the Root Models. As Pydantic generates multiple Root Leafs from the YANG Model, the value is nested in Root Models. A second recursive function is created to traverse the last model until a data type is encountered.

```
    if isinstance(model, RootModel):
        # Call same Function with next Leaf
    elif isinstance(model, bool):
        return str(model).lower()  # Lowercase is required as Cisco Switch
interprets bools in lowercase
    elif isinstance(model, (str, int)):
        return model
    elif isinstance(model, Enum):
        return model.value
    return None
```

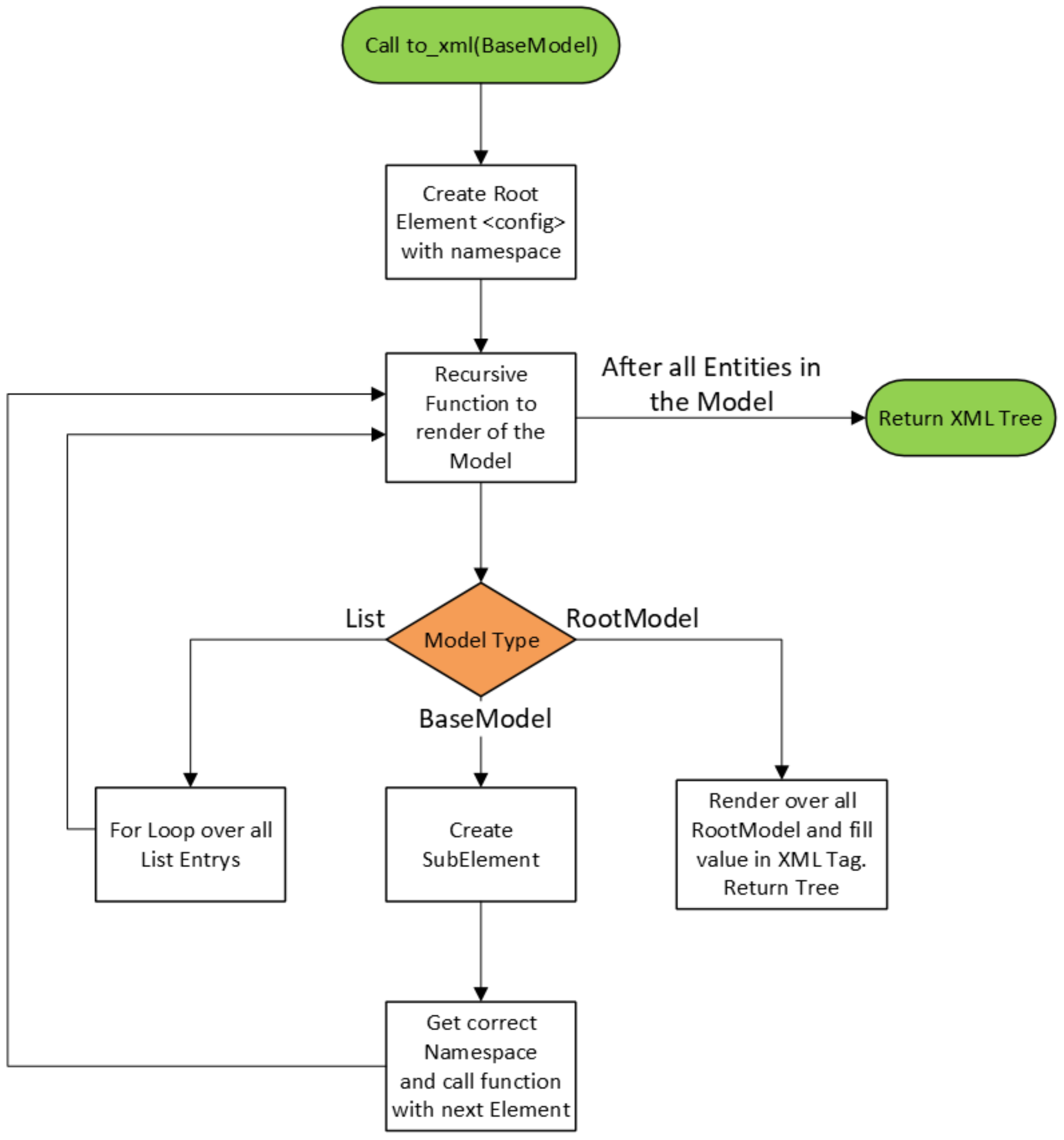

Call to_xml(BaseModel)
Create Root Element <config> with namespace
Recursive Function to render of the Model
After all Entities in the Model
Return XML Tree
List
Model Type
RootModel
BaseModel
For Loop over all List Entrys
Create SubElement
Render over all RootModel and fill value in XML Tag. Return Tree
Get correct Namespace and call function with next Element

Figure 4.1.: Flow Chart to Function 'to xml'

4.2.4. Namespace Handling

To set the correct namespace at the lowest XML tag in the tree, the alias from Pydantic (generated with the namespace from the YANG file) is used to obtain the correct namespace, and a rendering function sets it on the lowest element.

The namespace is calculated by splitting it into two parts: namespace and tag name.

```
modelkey_parts = modelkey.split(":", 2)
```

Handling Cisco Native for Interfaces

```
    if "Cisco-IOS-XE" in modelkey_parts[0]:
        # Create a list with Element 0 = "http://cisco.com/ns/yang/" and Element 1 =
    Left Part of the Alias
    # Combine the list without any separator
```

Handling OpenConfig for VLANs

```
    if "Cisco-IOS-XE" in modelkey_parts[0]:
    else:
        # Further split the left part of the alias by the dash "-" into a list

    # Iterate over the list and modify the values as follows:
    match namespace:
        case "openconfig":
            return "http://openconfig.net/yang/"
        case "if":
            return "interfaces/"
        case _:
            return namespace
    # Combine the list without any separator
```

1. The function `checkifnamespaceisintree` is called with the arguments (tree, namespace).
2. It is verified whether the tree contains the same namespace as the alias.
3. A recursive iteration over all parents is performed as long as:
   - The namespace matches the namespace from the TreeElement, resulting in a return of True.
   - The root element is reached, resulting in a return of False.
4. When "False" is obtained, the namespace tag is set.

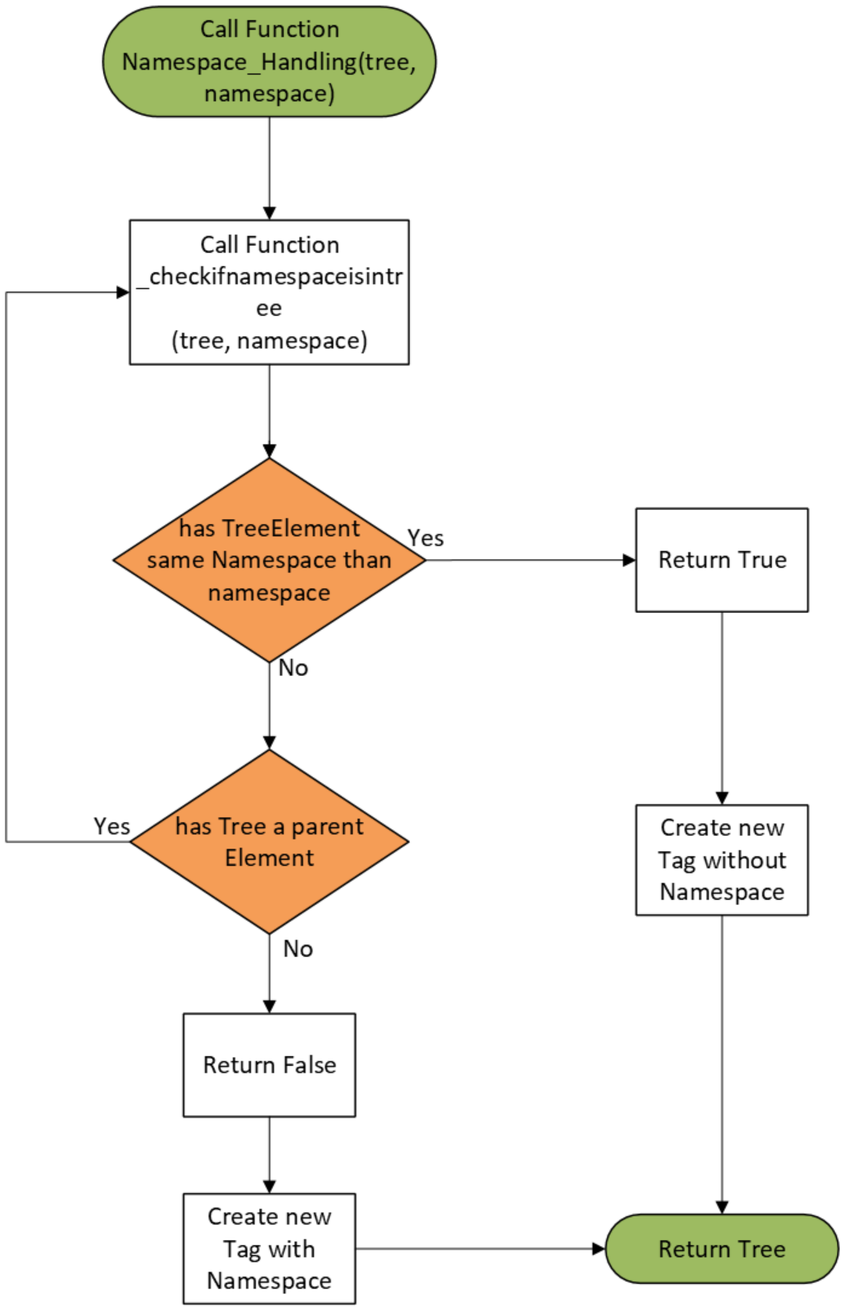

Call Function Namespace_Handling(tree, namespace)
Call Function _checkifnamespaceisintree (tree, namespace)
has TreeElement same Namespace than namespace
Yes
Return True
No
has Tree a parent Element
Yes
Create new Tag without Namespace
No
Return False
Create new Tag with Namespace
Return Tree

Figure 4.2.: Namespace Handling

4.2.5. Limitations

Due to the specific requirements of the NETCONF XML format, certain non-generic modifications have been implemented:

- Namespaces are hardcoded; for example, "Cisco-IOS-XE" must be changed to `http://cisco.com/ns/yang/`. If Cisco alters its namespace or new YANG models with different namespaces are utilized, the code must be adjusted.
- The NETCONF configuration namespace is hardcoded as `{urn:ietf:params:xml:ns:netconf:base:1.0}`.
- The root tag is explicitly set to "config".

4.2.6. Public Function - to Basemodel

The "to basemodel" function converts a given XML document into a Python model. The XML structure is parsed, and the elements are mapped back into the corresponding fields of the model. This method is used to transform data from XML format into a Python object for further processing or manipulation.

The operation is as follows:

1. The first root element is obtained, and the recursive function is called.

```
dict = XMLModelConverter._to_dict(tree.getroot())
```

2. A recursive iteration over the given XML tree is performed, creating a dictionary for each new tag.

```
result = {}
for child in tree:
```

3. The XML tag is split by "}" and the second part is retained, as the tag contains namespace and name. Only the name is required. If a value exists in the XML, it is retrieved.

```
name = child.tag.split("}", 1)[-1]
text = child.text.strip() if child.text else None
```

4. The name is manually adjusted to match the correct name for Pydantic. This manual task addresses the differences between NETCONF and Pydantic.

```
match name:
    case "vlan-id":  # Optional: Adjust names
        name = "vlan_id"
    case "GigabitEthernet":
        name = "gigabit_ethernet"
    case "TenGigabitEthernet":
        name = "ten_gigabit_ethernet"
    case "TwentyFiveGigE":
        name = "twenty_five_gig_e"
    case "FortyGigabitEthernet":
        name = "forty_gigabit_ethernet"
    case "switchport-config":
        name = "switchport_config"
```

5. It is verified whether the element has children:

```
if len(child):
```

When children are present:

1. The recursive function is called with the next element and saved.

```python
child_dict =
    XMLModelConverter._to_dict(child)
```

2. It is checked whether there is more than one element with the same tag, for example, multiple `<GigabitEthernet>`. If so, it is saved in a list; otherwise, it is saved as a new dictionary entry.

```python
if name in result:
    if not isinstance(result[name],
    list):
        result[name] = [result[name]]
    result[name].append(child_dict)
else:
    result[name] = child_dict
```

When no children are present:

- It is verified whether the name is already in the dictionary; otherwise, an empty dictionary is saved as the value. The empty dictionary is necessary due to the empty tag `<trunk/>` and `<access/>` in the Mode Container.

```python
if name in result:
    if not isinstance(result[name],
    list):
        result[name] = [result[name]]
    result[name].append(text)
else:
    result[name] = text if text else
    {}
```

- It is checked whether the XML tag has a list as its value.

```python
if not isinstance(result[name], list):
    result[name] = [result[name]]
result[name].append(child_dict)
```

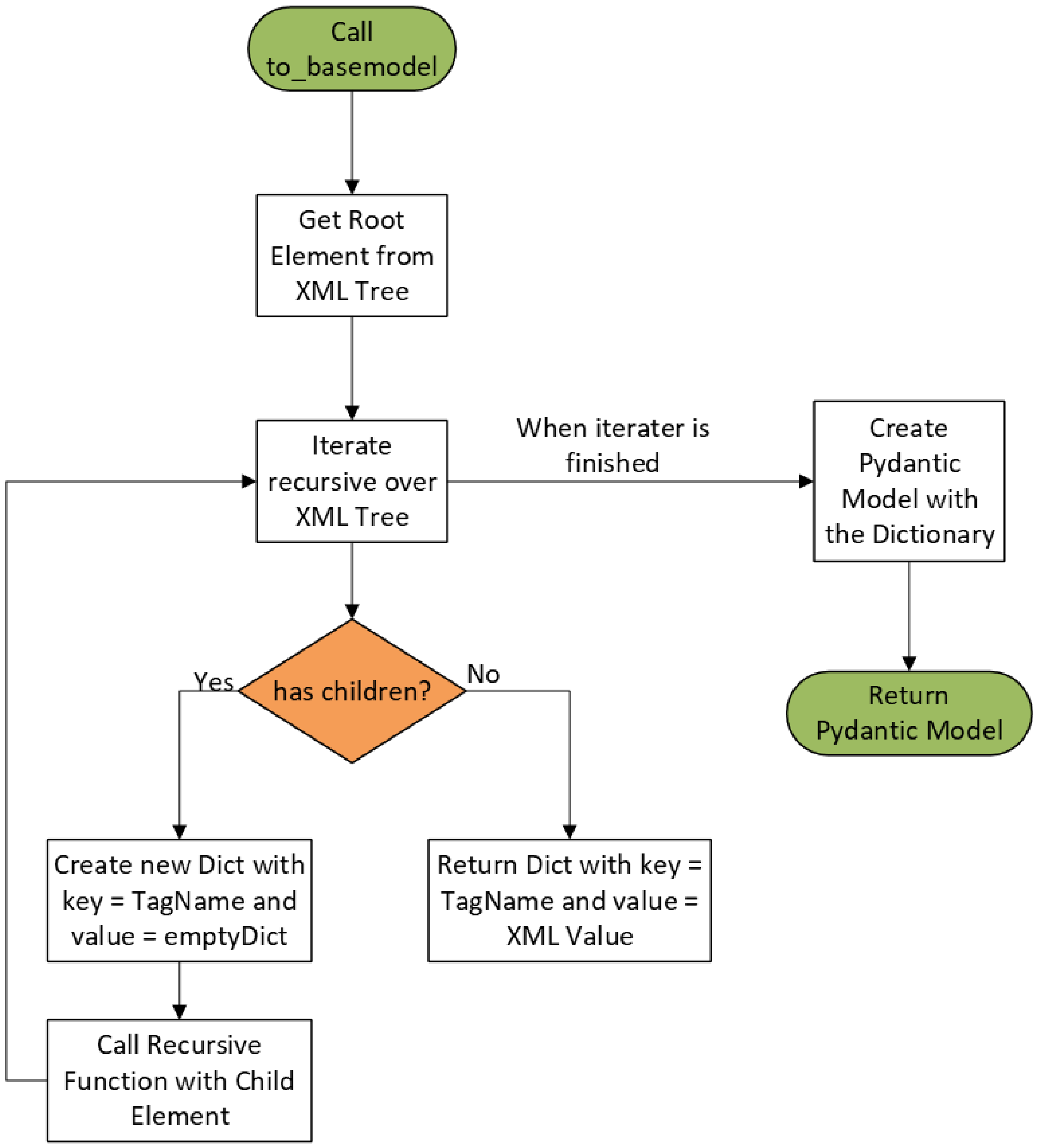

Call to_basemodel
Get Root Element from XML Tree
Iterate recursive over XML Tree
When iterater is finished
Create Pydantic Model with the Dictionary
Return Pydantic Model
has children?
Yes
No
Create new Dict with key = TagName and value = emptyDict
Return Dict with key = TagName and value = XML Value
Call Recursive Function with Child Element

Figure 4.3.: Flow Chart to Function 'to basemodel'

4.3. More Information

The code is published in the *GitHub Project*[1]. For additional information or comments, you are welcome to try out the XMLExporter in the GitHub repository.

[1]GitHub NetconfXMLExporter: December 16, 2024 https://github.com/raemsli/NetconfXMLExporter.git

4.3.1. Demo

The GitHub repository also includes a short demo with some interfaces and VLANs in Cisco Native and OpenConfig form for better explanation.

5. Infrahub GitLab Integration

For versioning and storage of our Python scripts, we utilize the integration between Infrahub and GitLab. This setup ensures efficient management of our codebase, enabling version control, collaboration, and seamless deployment workflows. GitLab provides robust versioning and repository management, while Infrahub facilitates infrastructure automation, ensuring a streamlined and scalable development pipeline.

5.1. Infrahub

We initially started using Infrahub version 0.16. Since Infrahub is a relatively new platform, it still undergoes frequent updates. Midway through our project, it had a major release, upgrading to version 1.0.0. Following that, several new releases were rolled out, focusing on enhancing functionality and fixing bugs.

5.1.1. Schema

The schema in Infrahub defines the structure, behavior, and relationships of entities within the system and directly influences the design of the GUI and data structure of Infrahub. This subsection provides an overview of the schema's components, including nodes, attributes, and relationships. Our straightforward design decision was driven by the client's needs. The primary goal was to make it as simple as possible, ensuring ease of use.

5.1.2. Schema Customization

Infrahub's schema is highly customizable, allowing users to define nodes, attributes, and relationships to match their specific use cases. This schema provides a robust framework to define and manage networking resources. By extending or modifying nodes, attributes, and relationships, users can tailor Infrahub to their specific requirements.

For further details, refer to the *Infrahub Schema Documentation*[1].

Detailed Components

The components seen in the schema diagram 5.1 directly correspond to the nodes in the GUI menu, as shown in Figure 5.2. Each node represents a specific entity in the system, such as a device, interface, or VLAN. The relationships between nodes define how they interact and connect with each other. The schema's structure is designed to reflect the real-world relationships between network devices and their configurations.

Nodes: Nodes represent the main entities within the system. Each node includes metadata such as name, namespace, attributes, and relationships.

Our Nodes

1. **Device**: Represents network devices with attributes like `name`, `description`, `platform`, `user`, `password`, `port`, and `status`. The relationships include `interfaces`.

2. **Interface**: Describes network interfaces, associated with a `device`. Attributes include `name`, `description`, `mode`, and `status`, with relationships to `device`, `ip_address`, and VLANs.

[1] Infrahub Schema Documentation: December 16, 2024 https://schema.infrahub.app

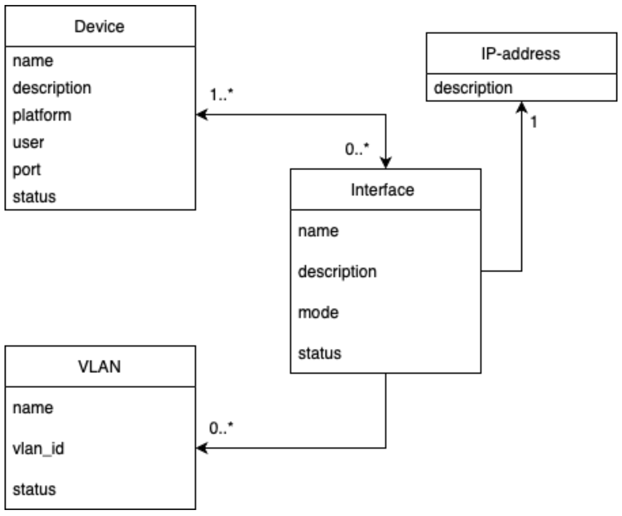


Figure 5.1.: Infrahub schema diagram

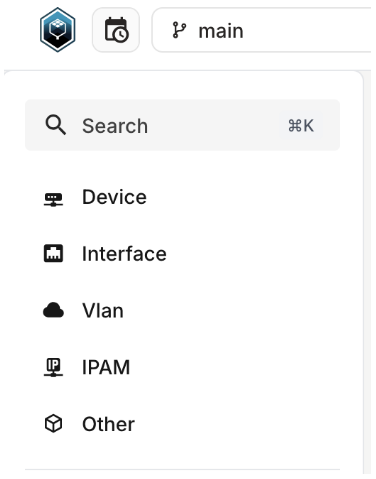


Figure 5.2.: Infrahub Menu showing all the nodes that are in the schema

3. **VLAN**: Represents VLAN configurations, with attributes for `name`, `vlan_id`, and `status`. This node is used to create a resource pool, allowing one to assign the pool to an interface, and Infrahub will create a new VLAN with the next free `vlan_id`.
4. **IP Prefix and IP Address**: Defines IP-related entities in the `IPAM` namespace, inheriting from built-in Infrahub components.

1. Attributes define the properties of a node. Each attribute may include:

- **name**: The attribute name must be unique within a model and must be all lowercase.
- **kind**: The data type, such as `Text`, `Number`, or `Dropdown`.
- **unique**: Indicates if the value of this attribute must be unique in the database for a given model.
- **optionality**: Indicates whether the attribute is mandatory or optional.

Example:

```
          attributes:
            - name: name
              kind: Text
              unique: true
            - name: status
              kind: Dropdown
              choices:
                - name: active
                  label: Active
                  description: "Device is active"
                  color: "#7fbf7f"

```

2. Relationships define connections between nodes. Each relationship includes:

- **name**: The relationship name must be unique.
- **identifier**: Unique identifier of the relationship within a model; identifiers must match to traverse a relationship in both directions.
- **peer**: The related node or entity.
- **kind**: The nature of the relationship (e.g., `Component`, `Parent`, or `Attribute`).
- **cardinality**: Specifies the number of connections (e.g., `one`, `many`).

Example:

```
          relationships:
            - name: interfaces
              identifier: "device__interface"
              cardinality: many
              peer: NetworkInterface
              kind: Component
```

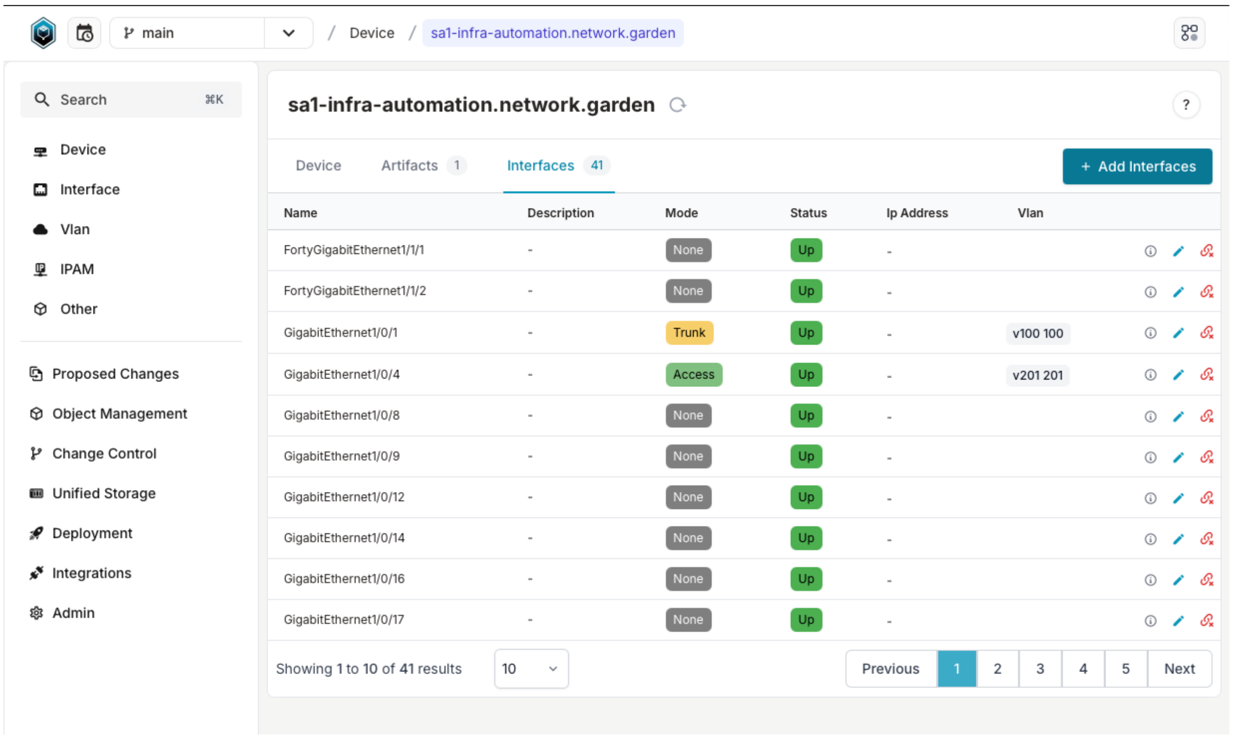
3. Inheritance and Uniqueness Nodes can inherit properties from other entities or enforce uniqueness constraints.
Example:

```
          inherit_from: ["CoreArtifactTarget"]
          uniqueness_constraints:
            - ["device", "name__value"]
```

4. Icons and Display Infrahub supports customizable icons and display labels for visual representation in the UI.
Example:

```
          icon: "mdi:router-network"
          display_labels:
            - "name__value"
```

After defining the schema, the nodes, attributes, and relationships are displayed in the Infrahub UI as seen in Figure 5.3.



Name	Description	Mode	Status	Ip Address	Vlan
FortyGigabitEthernet1/1/1	-	None	Up	-	
FortyGigabitEthernet1/1/2	-	None	Up	-	
GigabitEthernet1/0/1	-	Trunk	Up	-	v100 100
GigabitEthernet1/0/4	-	Access	Up	-	v201 201
GigabitEthernet1/0/8	-	None	Up	-	
GigabitEthernet1/0/9	-	None	Up	-	
GigabitEthernet1/0/12	-	None	Up	-	
GigabitEthernet1/0/14	-	None	Up	-	
GigabitEthernet1/0/16	-	None	Up	-	
GigabitEthernet1/0/17	-	None	Up	-	

Showing 1 to 10 of 41 results 10

Figure 5.3.: Infrahub showing all the interfaces that are currently available on a specific device

5.2. GitLab

To manage and store `Schemas`, `PythonTransformations`, `Checks`, `Artifact Definitions`, and `GraphQL Queries`, we integrate Infrahub with GitLab for version control and file storage. This integration provides a secure and efficient way to track changes to our scripts while maintaining a history of revisions.

The GitLab repository instance is hosted at our university's GitLab (gitlab.ost.ch), chosen specifically for its enhanced security and suitability for handling sensitive data. This ensures that all scripts and configurations are kept in a secure environment that meets our data protection requirements.

5.2.1. GitLab Integration

For authentication, we use a GitLab Access Token. This token allows Infrahub to securely interact with our GitLab repository, providing the necessary permissions to pull branches and commits while ensuring that access is properly controlled.

The integration within Infrahub is configured as a repository, which allows us to automatically pull all branches and commits. This setup ensures that any changes made to the repository are automatically reflected in the Infrahub system without the need for manual intervention.

We opted for a `Repository` integration rather than a `Read-only Repository` for practical reasons. With a `Read-only Repository`, new commits would need to be manually loaded by updating the "ref" in the repository definition. This process could introduce unnecessary delays and manual steps, so using a fully functional repository that supports automatic pulling of new commits and branches provides a more streamlined and efficient workflow.

5.2.2. GraphQL

We use a GraphQL query to retrieve and filter data from Infrahub. The flexibility of GraphQL allows us to specify the individual device name in the query, enabling us to tailor the request to the specific device's data. This is essential because we create a separate NETCONF XML for every device and only require the data linked to the specific device. In the GraphQL query, this is achieved as follows:

Listing 5.1: GraphQL query to retrieve device-specific data

```
query GetDevice ($device: String!) {
    NetworkDevice(name__value: $device) {
        edges { ... }
    }
}
```

The query processes the data and returns it as a dictionary, which is subsequently passed to the transformation function.

5.2.3. PythonTransform

To convert the retrieved data into a NETCONF XML, we employ a Python-based transformation. This transformation is implemented as a class that inherits from the `InfrahubTransform` base class. The core asynchronous function is `transform`, which takes `self` and `data` as arguments. Additionally, the GraphQL query required to fetch the data must be defined prior to implementing the transformation logic. To better understand how the function operates, refer to the flow diagram in Figure 5.4.

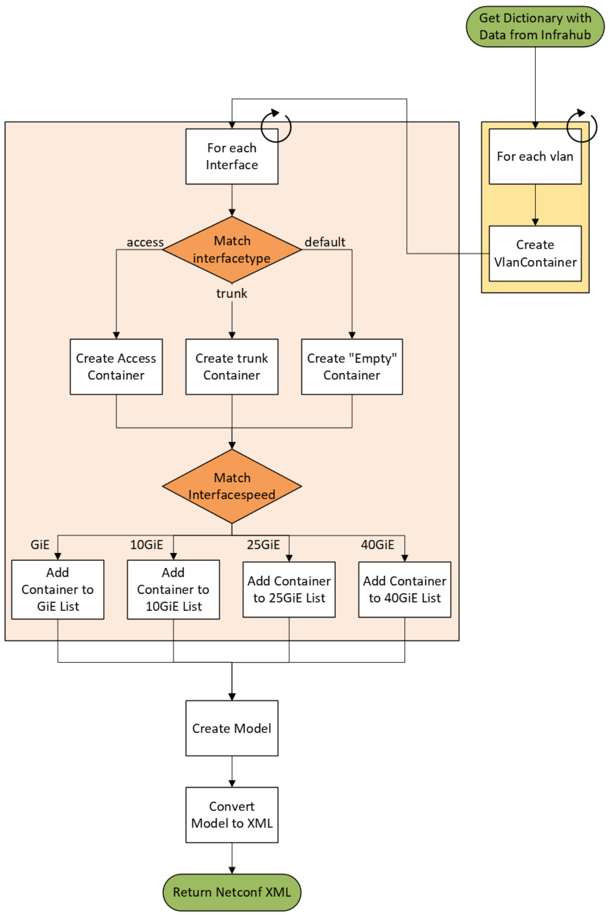


Figure 5.4.: Flow diagram for the Python Transformation Function

For clarity, the `transform` function is divided into four key parts:

1. Class Definitions
2. Creating the VLAN Model (yellow in Figure 5.4)
3. Creating the Interface Model (light orange in Figure 5.4)
4. Using the `XMLExporter` class to convert the Pydantic model into a NETCONF XML.

Class Definitions Using the `InfrahubTransform` Python library, we define the transformation class and inherit its methods. Within the inherited class, we specify the GraphQL query name and implement the asynchronous `transform` function, which accepts `self` and the data obtained from GraphQL.

```python
from infrahub_sdk.transforms import InfrahubTransform

class TransformIntoNetconf(InfrahubTransform):
    query = "GetInterfacefromDevice"

    async def transform(self, data: Dict[str, Any]) -> str:
        # Implementation goes here
```

Creating the VLAN Model

For the VLAN model, we adhere to the structure of the OpenConfig YANG model. The input data is nested, with the key `edges` containing a list of all VLANs defined in Infrahub. We iterate through this list, extract the required values from the nested dictionaries, and assign them to a Pydantic container. Finally, we append the container to a newly created list.

```python
vlanlist = []

for vlan in data["NetworkVlan"]["edges"]:
    vlanname = vlan["node"]["name"]["value"]
    vlanid = vlan["node"]["vlan_id"]["value"]
    vlanentry = VlanListEntry2(
        vlan_id=vlanid, config=ConfigContainer(vlan_id=vlanid, name=vlanname)
    )
    vlanlist.append(vlanentry)
```

Creating the Interface Model

For the interface model, we follow the Cisco native structure. The logic mirrors that used for VLANs: iterating through interfaces, creating the appropriate containers, and appending them to the corresponding lists.

```python
gigabit_ethernet_interfacelist = []
ten_gigabit_ethernet_interfacelist = []
twenty_five_gigabit_ethernet_interfacelist = []
forty_gigabit_ethernet_interfacelist = []

for intf in data["NetworkDevice"]["edges"][0]["node"]["interfaces"]["edges"]:
    # Create interface list entries

    match interfacespeed:
        case "GigabitEthernet":
            gigabit_ethernet_interfacelist.append(interfacelistentrys)
        case "TenGigabitEthernet":
            ten_gigabit_ethernet_interfacelist.append(interfacelistentrys)
        case "TwentyFiveGigE":
            twenty_five_gigabit_ethernet_interfacelist.append(interfacelistentrys)
        case "FortyGigabitEthernet":
            forty_gigabit_ethernet_interfacelist.append(interfacelistentrys)
```

However, Cisco's native structure is more complex, requiring additional logic. This section is further divided into sub-steps for clarity:

Interface Name The interface speed (e.g., "GigabitEthernet") and name (e.g., "1/0/1") are combined in Infrahub but need to be separated for Cisco. A regex function is used to split these values based on the following pattern:

1. Unlimited letters
2. One or two digits
3. Optional backslash or empty string (e.g., `Vlan1` is valid)
4. One or two digits, or empty
5. Optional backslash or empty string (e.g., `GigabitEthernet1/1` is valid)

6. One or two digits, or empty

If no match is found, the interface is skipped. Otherwise, the regex splits the name into two parts:

```python
pattern = r"(.*?)([0-9](?:/[0-9](?:/[0-9]([0-9])?)?)?)$"

match = re.match(pattern, intf["node"]["name"]["value"])
if not match:
    continue  # Skip this interface if the name doesn't match the pattern

interfacespeed = match.group(1)
interfacename = match.group(2)
```

Container Matching Each port speed requires a specific container and mode. Using a match-case clause, we define the container for each case. If no match is found, the interface is created with just a name (equivalent to "no config" in Cisco):

```python
trunkvlanidlist = []

match intf["node"]["mode"]["value"]:
    case "trunk":
        for trunkvlan in intf["node"]["vlan"]["edges"]:
            # Assign VLAN IDs to trunkvlanidlist

        interfacelistentrys = InterfaceListEntry(
            name=interfacename,
            switchport_config=SwitchportConfigContainer(
                switchport=SwitchportContainer(
                    mode=ModeContainer(trunk=TrunkLeaf()),
                    trunk=TrunkContainer(
                        allowed=AllowedContainer(
                            vlan=VlanContainer2(
                                vlans=",".join(map(str, trunkvlanidlist))
                            )
                        )
                    ),
                )
            ),
        )

    case "access":
        interfacelistentrys = InterfaceListEntry(
            name=interfacename,
            switchport_config=SwitchportConfigContainer(
                switchport=SwitchportContainer(
                    mode=ModeContainer(access=AccessLeaf()),
                    access=AccessContainer(
                        vlan=VlanContainer(

    vlan=intf["node"]["vlan"]["edges"][0]["node"]["vlan_id"]["value"]
                        )
                    ),
                )
            ),
        )

    case _:
        interfacelistentrys = InterfaceListEntry(
            name=interfacename,
            switchport_config=SwitchportConfigContainer(),
        )
```

Building the Final Model

Finally, we create the model, convert it into XML, and return it as a string. Currently, Infrahub only supports JSON and plain text as artifact types, so the XML must be returned as a string. Support for XML as a native data type is planned for future releases.

```python
model = Model(
    vlans=VlansContainer(vlan=vlanlist),
    native=NativeContainer(
        interface=InterfaceContainer(
            gigabit_ethernet=gigabit_ethernet_interfacelist,
            ten_gigabit_ethernet=ten_gigabit_ethernet_interfacelist,
            twenty_five_gig_e=twenty_five_gigabit_ethernet_interfacelist,
            forty_gigabit_ethernet=forty_gigabit_ethernet_interfacelist,
        )
    ),
)
xmlcontent = XMLModelConverter.to_xml(model)
xml_string = ETree.tostring(xmlcontent, encoding="unicode")
return parseString(xml_string).toprettyxml()
```

Artifact Definition

Artifacts are used to link the generated NETCONF XML with the corresponding devices in Infrahub. Whenever a value is changed in Infrahub, a new artifact is automatically created to reflect these changes, and the previous artifact is stored for versioning purposes. This ensures that every version of the configuration is preserved and can be referenced as needed.

Artifacts are defined using an `Artifact Definition`, which specifies the necessary parameters and input variables for the artifact. For our use case, the `Artifact Definition` is configured with several key elements:

- **PythonTransformation:** The transformation logic that converts the data from Infrahub into a Pydantic model and generates the NETCONF XML.
- **GraphQL:** The GraphQL query used to retrieve the device-specific data from Infrahub.
- **Device as Input Variable:** The device being processed, which serves as the input for the transformation.
- **StandardGroup:** A group of target devices that the artifact applies to. The devices in this group are considered the recipients of the generated NETCONF XML.

5.2.4. Checks

We decided to create our custom check to address a specific need in our Infrahub schema. The `Interface` node in our schema supports multiple modes: `none`, `trunk`, and `access`. According to configuration standards, interfaces in `access` mode should be associated with exactly one VLAN, while interfaces in `trunk` mode can have multiple VLANs.

However, due to the cardinality setting for the relationship, we had to choose `many` to allow multiple VLANs for `trunk` mode. This created a challenge: it became possible to assign more than one VLAN to interfaces in `access` mode, violating our intended configuration rules.

To solve this, we implemented a custom check. This check ensures that any proposed changes to the schema are validated against the rule that `access` mode interfaces must have exactly one VLAN. If an interface in `access` mode is found to have more than one VLAN, the check logs an error, notifying users that this configuration is not allowed.

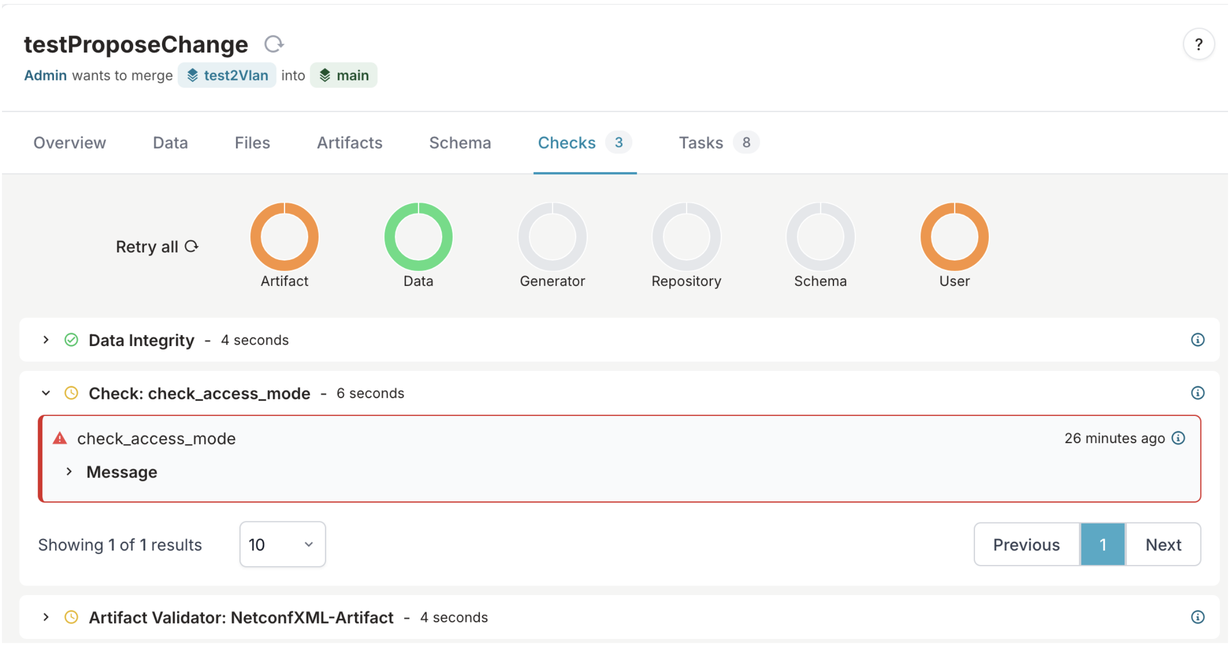
The `validate_interfaces.py` script implements a custom Infrahub check named `check_access_mode`. This check was created to ensure compliance with network configuration rules regarding VLAN assignments for interfaces in `access` mode. We created this check with the help of the Infrahub documentation.[2].

How It Works:

1. **GraphQL Query:** The `GetInterfaceDetails` query fetches details about all interfaces, including their mode and associated VLANs.
2. **Validation Logic:** The `check_access_mode` class processes the query results. For each interface in `access` mode, it counts the associated VLANs. If the count is not exactly one, an error is logged.
3. **Error Logging:** Errors include detailed messages identifying the misconfigured interface and the number of associated VLANs:

```
Interface 'Interface1' is in 'access' mode but has 2 VLAN(s).
```

After creating a proposed change with an incorrect configuration (in our example, configuring the interface in access mode with 2 VLANs), the check will fail, and a window will appear as you can see in Figure 5.5.


testProposeChange
Admin wants to merge test2Vlan into main

Overview · Data · Files · Artifacts · Schema · Checks 3 · Tasks 8

Retry all · Artifact · Data · Generator · Repository · Schema · User

Data Integrity - 4 seconds

Check: check_access_mode - 6 seconds

check_access_mode — 26 minutes ago
Message

Showing 1 of 1 results · 10 · Previous · 1 · Next

Artifact Validator: NetconfXML-Artifact - 4 seconds

Figure 5.5.: Check 'check_access_mode' failed because an interface in 'access' mode has 2 VLANs

[2]Infrahub Documentation: December 16, 2024 https://docs.infrahub.app/topics/check

Use Case: This check is particularly useful during:

- **Schema Updates:** Ensuring new configurations comply with access mode VLAN rules.
- **Data Imports**: Validating imported data to maintain consistent relationships and avoid misconfigurations.
- **Automated Pipelines:** Running this check as part of CI/CD workflows to catch errors before deployment.

Benefits:

- Prevents invalid configurations, reducing troubleshooting time.
- Ensures adherence to network standards for VLAN assignments.
- Automates validation, eliminating manual checks and improving efficiency.

6. Nornir Conditional Runner

To ensure the network remains operational during configuration updates, especially when the changes extend beyond VLANs or if there is a misconfiguration, we have been tasked with developing a custom runner plugin for Nornir. This plugin will help manage and control task execution to avoid disruptions and maintain network stability. This part of the project is open-source and will be published on GitHub[1] and made available through PyPi[2].

6.1. Introduction

The `ConditionalRunner` is a custom Nornir runner that enforces concurrency limits based on host groups. It allows you to control task execution by defining limits on the number of simultaneous tasks for specific groups of hosts, ensuring your Nornir tasks do not update vital network devices simultaneously. You can also specify skipping the rest of the group if a certain number of vital tasks fail. It is built on the threaded runner, with added conditional `group_limits` and `group_fail_limits` managed internally by a data structure consisting of semaphores, conditions, and counters, allowing tasks to remain idle in a waiting state until the start conditions are met. Installation and usage instructions are available in Appendix 2.

6.2. Fail Limits Feature

The `group_fail_limits` option allows you to specify the maximum number of failed tasks for a group before the runner skips the remaining tasks in the group. This feature is useful when you want to limit the impact of failing tasks on your network. For example, if one core device fails, you may want to skip the rest of the core devices to avoid further issues. The runner will only skip the tasks that are still waiting to run, not those that are already running.

The `skip_unspecified_group_on_failure` option sets the fail limit to 1 for all groups that do not have a `group_fail_limit` specified. This default behavior can be overridden by specifying `skip_unspecified_group_on_failure = False`, which will cause the runner not to skip the unspecified groups on failure. The specified `group_fail_limits` will always be used to skip the group on failure.

6.3. Logging

The ConditionalRunner leverages Python's built-in logging system to provide insights into its operation. It logs key events, such as:

- Warnings when a group is configured on a host but is missing in `group_limits`, defaulting to the global limit.
- Warnings when an invalid or missing `conditional_group_key` causes a fallback to host groups.
- Warnings if the `group_fail_limits` for a group are met or exceeded.

[1] GitHub repository: December 11, 2024 https://github.com/InfrastructureAsCode-ch/nornir_conditional_runner
[2] PyPi package: December 11, 2024 https://pypi.org/project/nornir-conditional-runner/

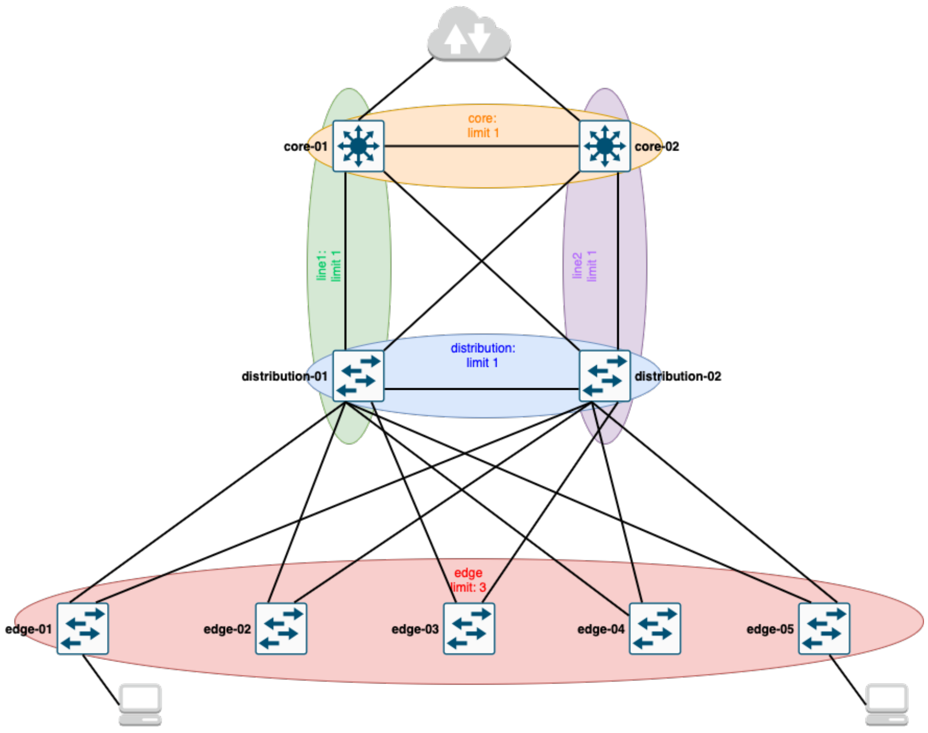


Figure 6.1.: Demo topology with conditional groups

6.4. Demo

A demo is available in the GitHub repository[3]. The topology of the INS network is utilized to demonstrate the ConditionalRunner, as it presents an interesting and complex use case involving circular conditions.

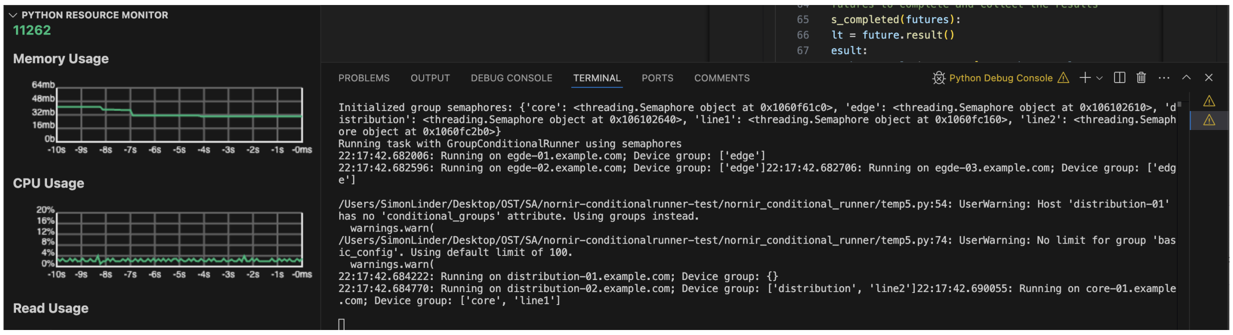
6.5. Error Handling / Fallback to Default Behavior of the Threaded Runner

- If the `conditional_group_key` is provided but no conditional groups are defined in the host data, the runner will warn you and default to using the host groups as the conditional groups.
- If no `group_limits` are specified for a group, the runner will default to using the global `num_workers` value as the limit.
- If neither `group_limits` nor a `conditional_group_key` are provided, the runner will fall back to using the host groups as conditional groups, with the default limits set to the global `num_workers`. This behavior mirrors that of the default threaded Nornir runner.
- Invalid group limits (i.e., non-positive integers) will result in a ValueError.

[3]GitHub repository - demo: December 11, 2024 https://github.com/InfrastructureAsCode-ch/nornir_conditional_runner/blob/main/demo/demo.py

6.6. Code Decisions

Initially, we tried to implement the `ConditionalRunner` with a combination of [4] but quickly realized that although this approach might be sufficient to meet the requirements, there is a better way to implement the runner. We decided to implement the runner using a combination of concepts we learned in the course on parallel programming. Semaphores are used to control the number of tasks that can run concurrently, conditions to signal when a task can start running, and counters and locks to keep track of failed tasks. The first prototype of the runner used only semaphores, but this meant that once a semaphore was locked, a task further down in the start list that could have started was also blocked. We first solved this issue by implementing a start queue and shuffling the tasks in the queue. This approach worked but used a small portion of CPU cycles to shuffle the queue and perform a busy wait. As shown in Figure 6.2, the CPU usage is only the busy wait, as the tasks we run just wait for 20 seconds.

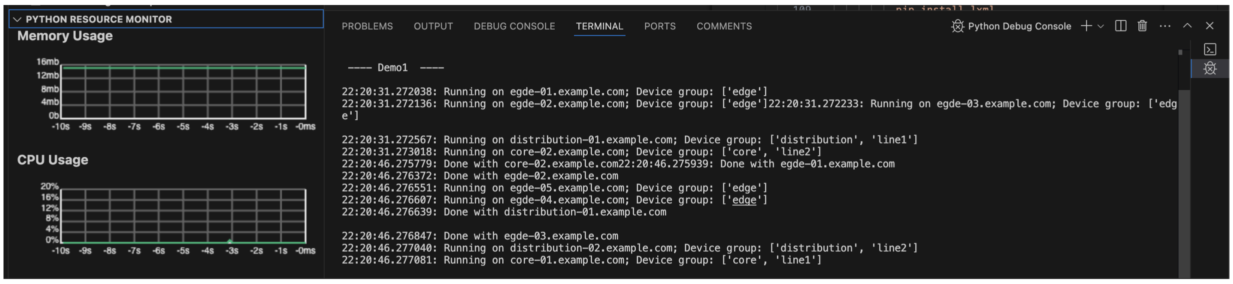


```
s_completed(futures):
lt = future.result()
esult:
```

```
Initialized group semaphores: {'core': <threading.Semaphore object at 0x1060f61c0>, 'edge': <threading.Semaphore object at 0x106102610>, 'd
istribution': <threading.Semaphore object at 0x106102640>, 'line1': <threading.Semaphore object at 0x1060fc160>, 'line2': <threading.Semaph
ore object at 0x1060fc2b0>}
Running task with GroupConditionalRunner using semaphores
22:17:42.682006: Running on egde-01.example.com; Device group: ['edge']
22:17:42.682596: Running on egde-02.example.com; Device group: ['edge']22:17:42.682706: Running on egde-03.example.com; Device group: ['edg
e']

/Users/SimonLinder/Desktop/OST/SA/nornir-conditionalrunner-test/nornir_conditional_runner/temp5.py:54: UserWarning: Host 'distribution-01'
has no 'conditional_groups' attribute. Using groups instead.
  warnings.warn(
/Users/SimonLinder/Desktop/OST/SA/nornir-conditionalrunner-test/nornir_conditional_runner/temp5.py:74: UserWarning: No limit for group 'bas
ic_config'. Using default limit of 100.
  warnings.warn(
22:17:42.684222: Running on distribution-01.example.com; Device group: {}
22:17:42.684770: Running on distribution-02.example.com; Device group: ['distribution', 'line2']22:17:42.690055: Running on core-01.example
.com; Device group: ['core', 'line1']
```

Figure 6.2.: Resource usage with busy wait

Therefore, we decided to eliminate the busy wait and add conditions, dispatching the task start into its own function so that later tasks can skip waiting ones. This approach allowed us to implement the runner more efficiently and reliably. Additionally, we reduced the CPU usage to 0% when the tasks are waiting to start, as shown in Figure 6.3.

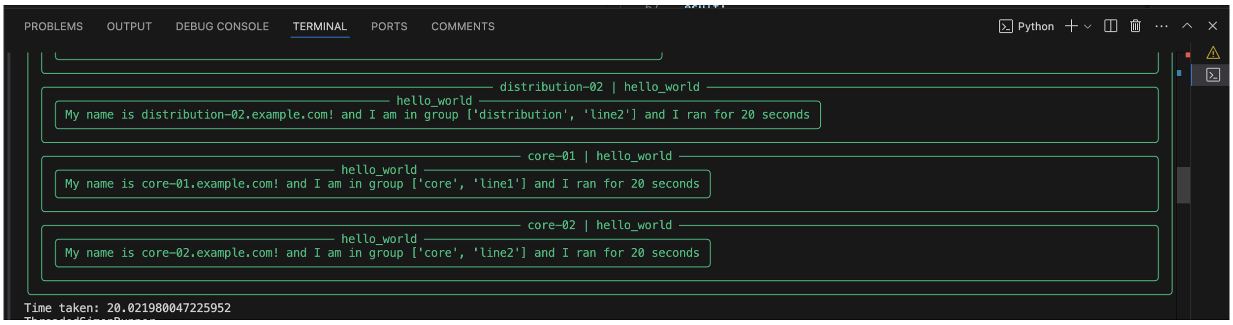


```
 ---- Demo1  ----

22:20:31.272038: Running on egde-01.example.com; Device group: ['edge']
22:20:31.272136: Running on egde-02.example.com; Device group: ['edge']22:20:31.272233: Running on egde-03.example.com; Device group: ['edg
e']

22:20:31.272567: Running on distribution-01.example.com; Device group: ['distribution', 'line1']
22:20:31.273018: Running on core-02.example.com; Device group: ['core', 'line2']
22:20:46.275779: Done with core-02.example.com22:20:46.275939: Done with egde-01.example.com
22:20:46.276372: Done with egde-02.example.com
22:20:46.276551: Running on egde-05.example.com; Device group: ['edge']
22:20:46.276607: Running on egde-04.example.com; Device group: ['edge']
22:20:46.276639: Done with distribution-01.example.com

22:20:46.276847: Done with egde-03.example.com
22:20:46.277040: Running on distribution-02.example.com; Device group: ['distribution', 'line2']
22:20:46.277081: Running on core-01.example.com; Device group: ['core', 'line1']
```

Figure 6.3.: Resource usage of the final conditional runner

We even achieved performance comparable to the threaded runner. As shown in Figure 6.4, the threaded runner has overhead to start a task of around 21 milliseconds.

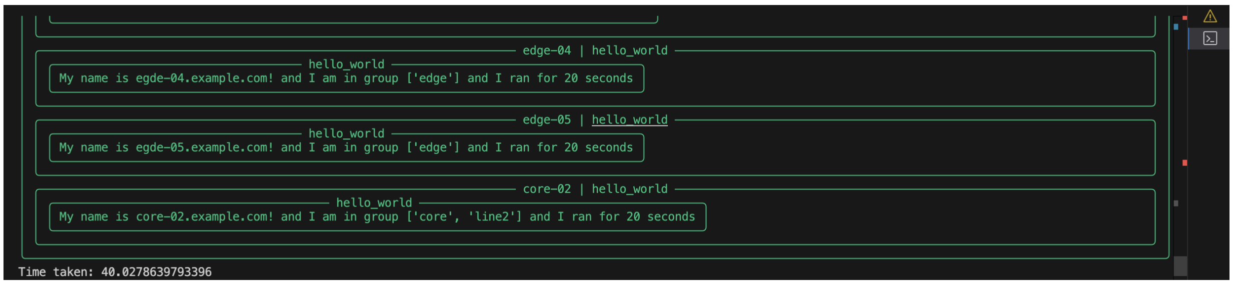
[4] Nornir Filter: December 11, 2024 https://nornir.readthedocs.io/en/latest/howto/advanced_filtering.html



```
distribution-02 | hello_world
hello_world
My name is distribution-02.example.com! and I am in group ['distribution', 'line2'] and I ran for 20 seconds

core-01 | hello_world
hello_world
My name is core-01.example.com! and I am in group ['core', 'line1'] and I ran for 20 seconds

core-02 | hello_world
hello_world
My name is core-02.example.com! and I am in group ['core', 'line2'] and I ran for 20 seconds

Time taken: 20.0219800047225952
```

Figure 6.4.: Threaded runner 20s task - all run in parallel

The ConditionalRunner has about the same overhead (28 milliseconds) to start a task as the threaded runner, as shown in Figure 6.5, even though the ConditionalRunner has to resolve some circular conditions and start the tasks in two batches sequentially.



```
edge-04 | hello_world
hello_world
My name is egde-04.example.com! and I am in group ['edge'] and I ran for 20 seconds

edge-05 | hello_world
hello_world
My name is egde-05.example.com! and I am in group ['edge'] and I ran for 20 seconds

core-02 | hello_world
hello_world
My name is core-02.example.com! and I am in group ['core', 'line2'] and I ran for 20 seconds

Time taken: 40.0278639793396
```

Figure 6.5.: Conditional runner 20s task - many conditions causing two batches

6.7. Testing

Testing the ConditionalRunner was crucial to ensure its reliability and stability. We used the Pytest framework to write unit tests for the runner, covering various scenarios and edge cases. The tests are available in the GitHub repository[5]. The test coverage of 98.8% is highlighted as a badge on the repository to build confidence with new users.

To ensure we only test our runner code and not the Nornir code, we used `unittest.mock` to mock the Nornir tasks and results.

6.8. Publishing and CI/CD

The ConditionalRunner is published on PyPi and available for download using pip. This is done using the GitHub Actions CI/CD pipeline[6], which automatically tests the package for Python 3.8 - 3.13, builds, and publishes it to PyPi when a new release/version is created on GitHub. The code is formatted and linted using ruff and checked with mypy for type hints. Additionally, the code coverage is checked with pytest-cov and automatically uploaded to a JSON on GitHub Pages[7] which is used by the coverage badge.

[5]GitHub repository - test: December 11, 2024 https://github.com/InfrastructureAsCode-ch/nornir_conditional_runner/blob/main/tests/test_conditional_runner.py
[6]GitHub Actions CI/CD pipeline: December 11, 2024 https://github.com/InfrastructureAsCode-ch/nornir_conditional_runner/blob/main/.github/workflows/main.yaml
[7]GitHub Pages branch: December 11, 2024 https://github.com/InfrastructureAsCode-ch/nornir_conditional_runner/tree/gh-pages

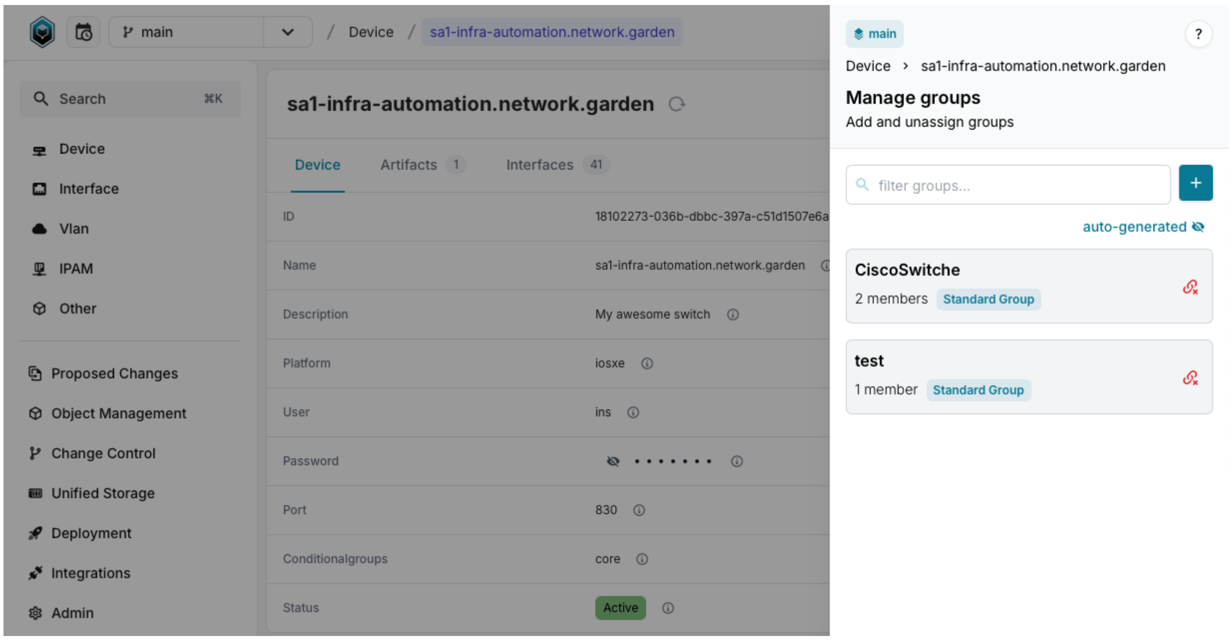
Additionally, we placed the ConditionalRunner in the official Nornir Plugin List[8] and it appears that users are responding positively.

6.9. Integration into Our Project

Integrating the ConditionalRunner into our project was straightforward as we could replace the threaded runner with our conditional runner and set the group limit to 1.

```python
    nr: Nornir = InitNornir(
        runner={
            "plugin": "ConditionalRunner",
            "options": {
                "num_workers": 2,
                "group_limits": {
                    "CiscoSwitches": 1,
                },
                "group_fail_limits": {
                    "CiscoSwitches": 1,
                },
                "skip_unspecified_group_on_failure": True,
            },
        },
    )
```

In Infrahub, a device can be added to groups that it belongs to, as shown in Figure 6.6.



sa1-infra-automation.network.garden

ID	18102273-036b-dbbc-397a-c51d1507e6a
Name	sa1-infra-automation.network.garden
Description	My awesome switch
Platform	iosxe
User	ins
Password	••••••••
Port	830
Conditionalgroups	core
Status	Active

Device › sa1-infra-automation.network.garden

Manage groups

Add and unassign groups

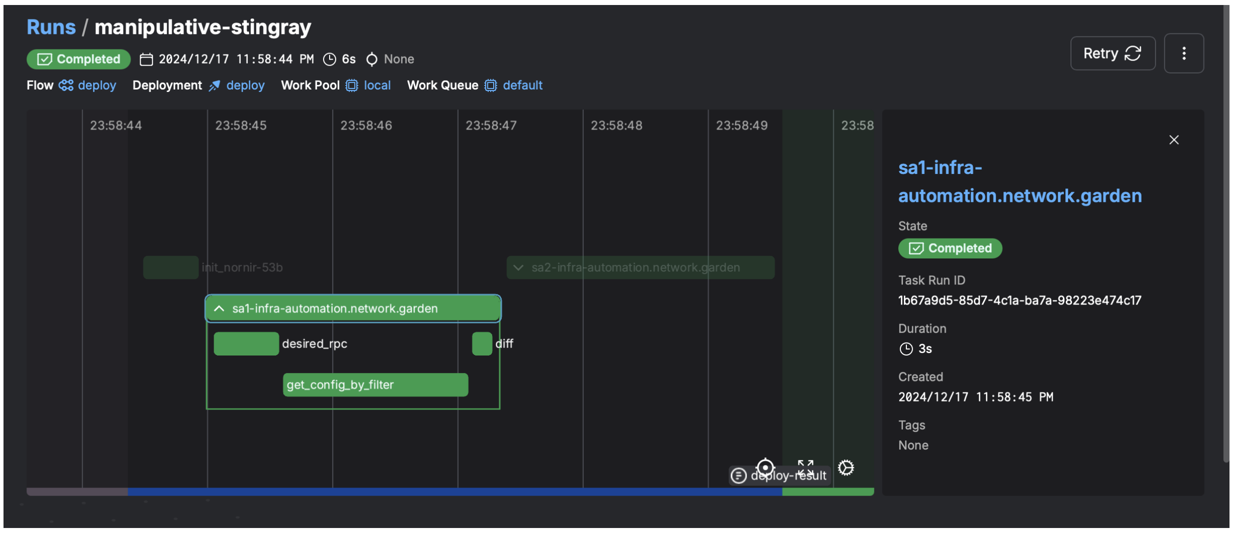
CiscoSwitche
2 members Standard Group

test
1 member Standard Group

Figure 6.6.: Group definition in Infrahub

As a result, it is clearly seen that the task for each device starts one after the other, as shown in Figure 6.7.

[8]Official Nornir Plugin List: December 11, 2024 https://nornir.tech/nornir/plugins/


Runs / manipulative-stingray

Completed 2024/12/17 11:58:44 PM 6s None

Flow deploy Deployment deploy Work Pool local Work Queue default

sa1-infra-automation.network.garden

State
Completed

Task Run ID
1b67a9d5-85d7-4c1a-ba7a-98223e474c17

Duration
3s

Created
2024/12/17 11:58:45 PM

Tags
None

Figure 6.7.: Prefect with conditional runner

6.10. Conclusion of the ConditionalRunner Component

Developing the ConditionalRunner for Nornir was a great experience. We learned a lot about the Nornir framework, threading, and concurrency in Python. Additionally, we valued the opportunity to contribute to the open-source community and connect with the bright minds in the network automation community.

7. Infrastructure

Our infrastructure is separated into two parts: the development environment using Docker and the production environment on Kubernetes.

7.1. Overview

This document outlines the infrastructure setup for our project. Our infrastructure is divided into two main environments: the development environment using Docker and the production environment on Kubernetes. This separation allows us to develop and test our application in an isolated environment before deploying it to production.

7.1.1. Development Environment

The development environment is designed to provide a consistent and isolated setup for each developer. We use Docker to create a devcontainer that includes all necessary tools and dependencies. This setup ensures that all developers work in the same environment, reducing the chances of environment-specific issues.

7.1.2. Production Environment

The production environment is hosted on a Kubernetes cluster, providing scalability and reliability. We use various tools and configurations to deploy our application, ensuring it runs smoothly in production.

7.2. Development Setup

We decided to first build the infrastructure in a development environment (docker-environment) and then move to production. This was done to create an easy playground for each of us to test and develop the application rapidly without interrupting each other.

7.2.1. Devcontainer

The devcontainer is used for our convenience, as it allows us to have a consistent development environment. The code can also be run within a virtual environment on the host machine, as described in the ReadMe.md[1]. However, the devcontainer takes care of installing all the CLI tools and environment variables for us. Additionally, if we break something during development, we can simply delete the container and start over. We need Python and Docker-in-Docker in the devcontainer. Additionally, we automatically install Prefect, Task, Poetry, and all the dependencies of our project.

7.2.2. Task

Task[2] is used to make it easy to set up and run the whole project with a few commands. The command `task setup` creates a virtual environment, installs all the dependencies, starts Infrahub, the Prefect server, and the Prefect pool. The command `task load-data` loads the Prefect deployments into Prefect and the data of our Cisco switches into Infrahub. There are also tasks

[1]GitLab repository - ReadMe: December 11, 2024 https://gitlab.ost.ch/ins-stud/sa-ba/sa-hs24-iac/ins_network_automation/-/blob/main/README.md?ref_type=heads
[2]Task Website: December 11, 2024 https://taskfile.dev/

like start, stop, prefect-stop, and destroy, which take care of all the other needs we had during development.

7.2.3. Poetry

Poetry[3] is used to manage the dependencies of our project. It is a great tool for managing dependencies and the virtual environment. A package is created from our project and groups the dependencies to separate the development dependencies from the production dependencies. This way, we can easily install the production dependencies in the production environment and only the development dependencies in CI jobs. Poetry also takes care of creating a CLI shortcut for us, so we can run the project with `iac_sa` and register the connection plugin for `NcDiff`. Additionally, the rule set for Ruff[4] and MyPy[5] are defined in the `pyproject.toml` file.

7.2.4. Docker-Compose

A stable version of the Infrahub docker-compose file is slightly modified so the Infrahub task worker is able to run `lxml`, as we need it for the transfer function of the Infrahub artifact.

7.2.5. Prefect

We use our own instance of the Prefect server in the test setup, unlike in production. This way, we can easily test the Prefect flows and the Prefect workers and break things in Prefect without any consequences. Additionally, the first version of Infrahub did not use Prefect as a backend, so we had to use our own instance of Prefect.

7.3. Production

The production environment is hosted on a Kubernetes cluster. The following tools are used to deploy our application.

7.3.1. Infrahub Helm

The Infrahub Helm chart is used to deploy the Infrahub application on the Kubernetes cluster. The default Helm chart values are slightly modified to fit our needs. HTTPS `extraTLS` is added, which works with a wildcard certificate. The package `lxml` is added to the Infrahub worker by adding the install command to the args of the pod. In the future, this could be improved by using a custom Docker image with the package already installed. Additionally, we had to give the RabbitMQ pod more resources and adjust the startup probes, as it was taking too long to start up and Kubernetes was killing the pod. This was suggested to us by an ObsMill engineer in their Discord channel.[6] We did not set it to unlimited, as the pod could potentially take up all the resources of one node. We just gave it a few more resources and adjusted the startup probes. The discrepancy between the default values probably comes from the fact that our Kubernetes cluster does not have as much IOPS all the time as required by ObsMill.[7]

[3]Poetry Website: December 11, 2024 https://python-poetry.org/
[4]Ruff bleeding edge formatter and linter: December 11, 2024 https://docs.astral.sh/ruff/
[5]MyPy static type checker: https://www.mypy-lang.org
[6]ObsMill Discord: December 13, 2024 https://discord.com/channels/1212332642801025064/1301914405176475729/threads/1312826976905723905
[7]ObsMill Website: December 13, 2024 https://docs.infrahub.app/topics/hardware-requirements/

In an older version of Infrahub (v1.0.8), we had to replace the NFS provisioner with a different one, as the one in the chart was not working. The stable/nfs-client-provisioner chart[8] is used, and the values are changed to `disable` the Infrahub chart. This was fixed in the newer version of the NFS provisioner. This error probably also came from the fact that ObsMill is still rapidly developing Infrahub and changed the NFS provisioner to a `ReadWriteOnce PersistentVolume`.

7.3.2. Cert-Manager

We use cert-manager[9] to manage the certificates for our application. Cert-manager is a Kubernetes add-on to automate the management and issuance of TLS certificates from various issuing sources. We use Let's Encrypt as the issuer for our certificates. The certificates are stored in a secret and mounted to the Infrahub ingress. The certificates are automatically renewed by cert-manager. We had to use the ACME DNS-01 challenge, as our website is not publicly reachable.

7.3.3. Prefect and Ingress

In production, we did not use our own instance of Prefect, as we could simply reuse the one from Infrahub and save some resources. Additionally, we gained some insight into the inner workings of the event-based job starts of Infrahub. For this, we just added a custom ingress to the Prefect server, which could reuse the same wildcard certificate.

7.3.4. Custom Prefect Worker and Dockerfile

To run the `Nornir NETCONF deployment` code on the Kubernetes cluster, we had to create a custom Docker image.

At first, we experimented with Prefect's Kubernetes working pool.[10] However, as all the documentation was just for AWS, GCP, or Azure clusters, we looked at the chart to add a Prefect worker to Kubernetes.[11] In this chart, we found a locally hosted option, but it was protected by `"accountId"` and we figured, as it was not documented, it is not included anymore in the free tier. We managed to run a Prefect worker pool on the local machine with access to the Kubernetes config, which was able to create new pods for flow runs, but Prefect is designed to install dependencies on the worker using `Environment Variables`. This means it always installs the Python package dependencies at flow start. Alternatively, Prefect provides the option to use a custom Docker image but only from a public registry for Kubernetes pools, and one still needs a machine that runs all the time and has access to create pods on the Kubernetes cluster.[12]

At this point, inspired by the fact that a worker can also pull just code from a private git repository, we figured we could also use a custom Docker image running a small Python Prefect script which we developed ourselves by combining a few Prefect functions we found in the Prefect documentation.[13] This way, we only pull the code into a running container and do not have to pull a Docker image from a registry every time a flow wants to run, and we do not need to build and publish a new container every time the code changes, decoupling the infrastructure from the code. This ensures that the infrastructure is just there to support the code's requirements and

[8]NFS Service Provisioner: December 13, 2024 https://github.com/kubernetes-sigs/nfs-ganesha-server-and-external-provisioner/tree/HEAD/charts/nfs-server-provisioner
[9]Cert-Manager Website: December 13, 2024 https://cert-manager.io/
[10]Run Prefect flows on Kubernetes: December 13, 2024 https://docs.prefect.io/v3/deploy/infrastructure-examples/kubernetes
[11]Prefect Helm Chart: December 13, 2024 https://github.com/PrefectHQ/prefect-helm
[12]Prefect Kubernetes Worker: December 13, 2024 https://docs.prefect.io/v3/deploy/infrastructure-examples/kubernetes
[13]Run Prefect flows in Docker container: December 13, 2024 https://docs.prefect.io/v3/deploy/infrastructure-examples/docker

dependencies, as it is Prefect's goal. First, the `gitOps.py`[14] loads the source code from the git repository using an access token secured in a Prefect secret. Then it converts the source code containing flows to Prefect deployments and serves the deployments to the Prefect server. The validate flow has a schedule of 30 minutes.

The `Dockerfile`[15] we created is based on a Python bookworm image, installing all the dependencies we need for the code to run with the same `project.toml` and `poetry.lock` as the `nornir NETCONF deployment` code repository. This Docker container is automatically built and pushed to the GitLab registry on every merge to the main branch. This Docker image is used by the `iac-prefect-worker` deployment in the Kubernetes cluster, which can also run in multiple replicas. The second replica will take over once the first goes down and is recreated by Kubernetes, as we quickly found out by testing it, deleting an `iac-prefect-worker` pod.

7.4. Conclusion of Infrastructure

In conclusion, our infrastructure setup allows us to develop and deploy our application efficiently. The development environment using Docker provides a consistent and isolated environment for each developer, while the production environment on Kubernetes ensures scalability and reliability. The use of tools like Task, Poetry, and Prefect streamlines our workflow and makes it easy to manage dependencies and automate tasks.

[14]GitLab repository - gitOps.py: December 13, 2024 https://gitlab.ost.ch/ins-stud/sa-ba/sa-hs24-iac/deployment-repo/-/blob/main/gitOps.py?ref_type=heads
[15]GitLab repository - Dockerfile: December 13, 2024 https://gitlab.ost.ch/ins-stud/sa-ba/sa-hs24-iac/deployment-repo/-/blob/main/Dockerfile?ref_type=heads

8. Technical Issues and Obstacles

During the implementation of our system, we faced several technical challenges that impacted the development process. In this chapter, we describe these obstacles and the solutions we devised to overcome them. By addressing these issues, we were able to refine our approach, improve system performance, and ensure a more robust and efficient solution.

8.1. NETCONF XML Exporter Class

Problem The namespaces in a NETCONF XML must match exactly what the Cisco switch expects. However, Pydantic often compresses and aliases namespaces, which causes mismatches. For example: **Correct Namespace:** "http://openconfig.net/yang/vlan"
Pydantic Alias: "openconfig-vlan:vlan-id"

Solution

1. Split the Alias: We split the Pydantic alias into two parts at the colon symbol. The right side (e.g., "vlan-id") becomes the XML tag name.
2. Map the Left Side: We split the left side (e.g., "openconfig-vlan") at the dash and use a match-case statement to map each part to the correct namespace (e.g., "http://openconfig.net/yang").
3. Manual Mapping: We manually define each alias compression performed by Pydantic to ensure the correct namespace is used.

This approach ensures the namespaces in the generated XML match exactly what the Cisco switch expects. You can find more details and the code implementation in Chapter 4.2.4.

8.2. Nornir NETCONF Deployment

Problem Comparing two XML files is challenging because the order of the XML tags can vary. This makes it difficult to determine if two XML files are identical. Initially, it always compared two different interfaces because the order of the tags was different or the content was similar (deep-diff behavior). Additionally, the XML diff returned by the `xmldiff` library is not very useful for a user-friendly report.

Solution

1. Use IDs for the XML tags: We use IDs for the XML tags to ensure that the order of the tags does not affect the comparison. This allows us to compare the content of the XML files rather than the order of the tags.
2. Write a Custom XML Diff: We write a custom XML diff that compares the content of the XML files and generates a user-friendly report. This allows us to attach the report to the email notification and provide the user with a clear overview of the differences between the XML files.

More information can be found in the **XML Diff** section 2.1.4.

8.3. Nornir Replace Operation

Problem The YANG model OpenConfig interfaces implementation on the Cisco switches does not support the replace operation on a complete interface.

Solution We use the merge operation to update the interface configuration and only replace the subcontainer of the interface. This allows us to update the interface configuration without affecting the rest of the configuration.
More information can be found in the **Validate or Deploy Configuration** section 2.1.3.

8.4. Conversion to Cisco Native Model

Problem During the integration and testing of all the components, we realized that the Cisco switches have a poor implementation of the OpenConfig YANG model in the trunk VLAN list. The Cisco implementation is not fully compliant with the OpenConfig YANG model. OpenConfig VLAN has multiple VLANs represented as multiple `<trunk-vlan>` tags.

```
<config>
    <interface-mode>TRUNK</interface-mode>
    <trunk-vlans>100</trunk-vlans>
    <trunk-vlans>200</trunk-vlans>
</config>
```

If we configure the trunk-vlans manually on the switch, the switch will reply with the above XML if we run `get_running_config()`. However, if we configure the trunk-vlans using the same NETCONF XML again, the switch ends up configuring just one VLAN. The second one overwrites the first one. To verify this was not a fault of our system, although it was unlikely after the test described above, we conducted a test with virtual Arista switches. The Arista switches did not have this problem. They were able to configure multiple VLANs with the same XML. As you can see in the Testing section 4.4.

Solution Apparently, there is no other way than to switch to the native Cisco YANG model. We wanted to avoid this because the OpenConfig model is less vendor-specific and much easier to use. However, the Cisco model is better implemented on the Cisco Catalyst devices.

1. **Nornir NETCONF Deployment**: To switch to the native Cisco model, we simply had to change the `defaults.yaml` file to use the Cisco model instead of the OpenConfig model and figure out how to use multiple IDs for the same model as the Cisco native model creates a container for each interface speed (e.g., `GigabitEthernet` and `TenGigabitEthernet`) as you can see in snippet 3. Additionally, we had to figure out the correct tags for the IDs and improve the self-developed `diff()` function to work with multiple IDs. The `defaults.yaml` file can be found in Appendix 3.

2. **Python Transformation**
    a) **Structure** In the Python Transformation, the basic structure of the Cisco Native Interface Model is significantly different from the structure of the OpenConfig Model. Because of this, we had to rebuild the function to fit the structure of the new model. The following is an example of the code we had to change:

```
# Code Example from Python Transform with OpenConfig Vlans
InterfaceListEntrys = InterfaceListEntry(
    name=intfname,
    config=ConfigContainer2(
```

```python
            name=intfname,
            type=typeIntf,
        ),
        ethernet=EthernetContainer(
            switched_vlan=SwitchedVlanContainer(
                config=ConfigContainer7(
                    trunk_vlans=trunkvlanidList
                )
            )
        )
    )
```

```python
    # Code Example from Python Transform with Cisco Native Trunk Vlans
    interfacelistentrys = InterfaceListEntry(
        name=interfacename,
        switchport_config=SwitchportConfigContainer(
            switchport=SwitchportContainer(
                mode=ModeContainer(
                    trunk=TrunkLeaf()
                ),
                trunk=TrunkContainer(
                    allowed=AllowedContainer(
                        vlan=VlanContainer2(
                            vlans=",".join(map(str, trunkvlanidlist))
                        )
                    )
                ),
            )
        ),
    )
```

More details on the Python Transformation are in Chapter 5.2.3.

b) **Interface Names** One main difference is the naming of the interfaces. In OpenConfig, we had simply a name tag like `'FortyGigabitEthernet1/0/1'`, but Cisco Native requires its own tag for the interface type `'FortyGigabitEthernet'` and the name as `1/0/1`. As we decided to leave the data in Infrahub as the interface name `'FortyGigabitEthernet1/0/1'`, we split this namespace with a regex into two parts. You can find the implementation in Chapter 5.2.3.

c) **Register Namespace** As a follow-up to the switch from the ElementTree Python library to the LXML Python library, we had to register the namespace in the Python Transformation. The assignment of the namespace happens in the XMLExporter function, but in resolving a bug, the ElementTree Element loses the reference as soon as the function is over. To solve this, we registered the namespace in the Python Transform function with $ETree.register_namespace('nc',' urn:ietf:params:xml:ns:netconf:base:1.0')$.

3. **XML Exporter**

a) **Change to LXML** A significant part of this change was to switch from the ElementTree XML API to LXML. We were forced to make this change because of its `getparent()` function, allowing us to loop over all parents. In OpenConfig, it wasn't necessary because a new namespace wasn't introduced by two neighbors. To clarify, here is an XML with the Cisco Native Model:

```xml
    <switchport>
        <mode xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-switch">
```

```
            <!-- Some Containers -->
        </mode>
        <trunk xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-switch">
            <!-- Some Containers -->
        </trunk>
    </switchport>
```

The problem is that the `mode` tag and the `trunk` tag have the same namespaces, are neighbors, and use a new namespace different from the root tag. In the figures below, we created flowcharts showing how the XMLExporter handles the namespaces, with the old and the new function.

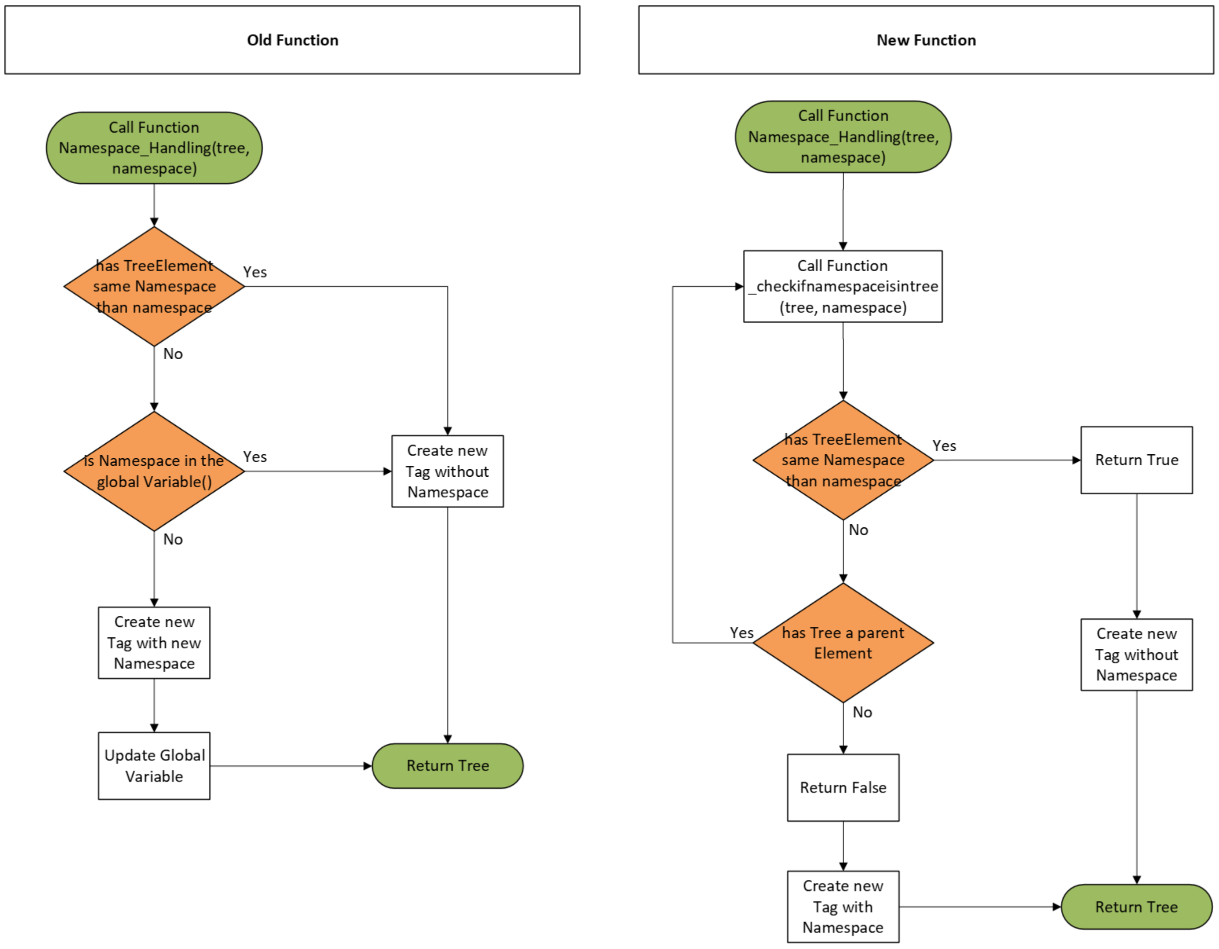

Old Function
Call Function Namespace_Handling(tree, namespace)
has TreeElement same Namespace than namespace
Yes
No
Is Namespace in the global Variable()
Yes
No
Create new Tag without Namespace
Create new Tag with new Namespace
Update Global Variable
Return Tree
New Function
Call Function Namespace_Handling(tree, namespace)
Call Function _checkifnamespaceisintree (tree, namespace)
has TreeElement same Namespace than namespace
Yes
No
Return True
Create new Tag without Namespace
has Tree a parent Element
Yes
No
Return False
Create new Tag with Namespace
Return Tree

Figure 8.1.: Flowchart Namespace Handling Function

More information about Namespace Handling is in Chapter 4.2.4.

b) **Namespace Handling** OpenConfig and Cisco Native have different namespaces and styles. `'openconfig-interfaces:interfaces'` `'Cisco-IOS-XE-native:native'`

In the code, we had to change the handling of the alias. We still split the alias into two parts by the ':', and use the left side as the namespace. However, the differences are:

i. URLs are different.

ii. Usage of '-'.

We solved this by adding an if clause to check if the left side contains 'Cisco-IOS-XE'. If yes, we create a list with the first entry being the Cisco URL and the second entry being the part of the alias. In the OpenConfig namespace, we need to actively change the value because Pydantic creates the alias with the values in YANG, but the Cisco switch requires different values in some exception cases. More information about the implementation is in Chapter 4.2.4.

4. **YANG Model** The Cisco native model is extensive. We had to select just a small part of the Cisco native interfaces YANG model to be able to configure the trunk and access VLANs. We found the appropriate branch of the native model with `pyang -f tree -p Cisco-IOS-XE-native/native/interface ./yang/sa1-infra-automation.network.garden/Cisco-IOS-XE-native.yang`. Initially, we tried to use the interface subtree of the Cisco native model. However, the Pydantic tool ran overnight, used up 40GB of memory, and created 495,939 lines of code. Only once we deleted containers that use the type `Decimal64Value`, which was not yet supported by Pydantic, did it become feasible. Making the subtree more specific was apparently not possible due to the unusual design of the Cisco native model. As we already knew we could delete containers from the subtree, we decided to delete all the containers that we do not need to make the output of Pydantic smaller and customize the YANG model to our needs, as shown in 3. In the end, we reduced the model to less than 500 lines of code.

5. **Pydantic Model** The change of the YANG model also had an impact on the Pydantic model, which we couldn't solve by only modifying the YANG as described at the point 4. Unfortunately, Pydantify creates aliases with the namespace derived from the YANG model. OpenConfig works well because each container with the required namespace resides in the correct YANG file. However, Cisco Native YANG models inherit the namespace from the main YANG file, resulting in all containers sharing the same namespace, and thus the same alias.

   To create the correct namespaces, we manually reviewed a valid Cisco NETCONF XML file using the command `show running-config | format 'netconf-xml'` and manually changed the namespaces.

   Fortunately, since the YANG file wasn't very large after our modifications, we only needed to change every alias from the container 'SwitchportContainer' upwards from `Cisco-IOS-XE-native` to `Cisco-IOS-XE-switch`:

```python
# Everything on a lower level has alias="Cisco-IOS-XE-switch"

class SwitchportContainer(BaseModel):
    mode: Annotated[ModeContainer, Field(None,
        alias="Cisco-IOS-XE-switch:mode")]
    access: Annotated[AccessContainer, Field(None,
        alias="Cisco-IOS-XE-switch:access")]
    trunk: Annotated[TrunkContainer, Field(None,
        alias="Cisco-IOS-XE-switch:trunk")]
    native: Annotated[NativeContainer, Field(None,
        alias="Cisco-IOS-XE-switch:native")]

class SwitchportConfigContainer(BaseModel):
    switchport: Annotated[SwitchportContainer, Field(None,
        alias="Cisco-IOS-XE-native:switchport")]

# Everything on a higher level has alias="Cisco-IOS-XE-native"
```

For more information, see Chapter 3.2.

8.4.1. Conclusion of the Switch to the Cisco Native Model

The switch to the native Cisco model was a necessary evil. The OpenConfig model would have been much easier to use, but the Cisco switches do not support it well. The Cisco native model is more vendor-specific and harder to use, but it is better implemented on the Cisco Catalyst devices. We had to make some adjustments to the YANG model, the Pydantic structure, and the XML exporter to accommodate the switch to the Cisco native model. However, the switch to the Cisco native model was necessary to ensure that the system works correctly with the Cisco Catalyst devices. In the end, it made our solution more robust and proved that we can handle different YANG models, even large and complex ones like Cisco native.

Part III.

Project Documentation

1. Results

Key Achievements

The *Network Configuration Automation with Infrahub and Nornir* project successfully delivered a comprehensive and automated system for VLAN configuration management. The key achievements are as follows:

1. **Automation of Network Configuration:** Utilized the Nornir framework, a Python-based automation tool, to deploy network configurations efficiently. This automation reduced manual intervention, minimized errors, and improved deployment speed.
2. **Integration with Infrahub:** Employed Infrahub as the single source of truth for managing and versioning network configurations, ensuring consistency and centralized oversight of network states.
3. **Dry-Run Validation:** Implemented a dry-run feature to validate configurations before deployment, providing a safe mechanism to preview changes and minimize disruptions.
4. **GitLab Integration:** Integrated GitLab for robust version control and collaboration, enabling efficient change tracking and auditing.
5. **Custom Tools for Stability:** Developed the Conditional Runner plugin for Nornir to enforce concurrency limits and maintain network stability by managing controlled task execution across devices.
6. **Validation and Transformation with Pydantic Models:** Integrated Pydantic for validating network configurations derived from YANG schemas and created XML generators for NETCONF compatibility.
7. **Workflow Orchestration with Prefect:** Used Prefect for defining, scheduling, and monitoring workflows, simplifying the management of complex processes and enhancing scalability.
8. **Scalability and Compatibility:** Designed the system to support multiple YANG models, enabling compatibility across various device types such as Cisco, Arista, and Nokia. Optimized performance by adopting lightweight plugins like NornirNETCONF.
9. **Configuration Consistency:** Implemented XML diff tools and validation scripts to ensure the alignment of current and desired states, reducing discrepancies and enforcing consistency.
10. **Improved Network Reliability:** Minimized human errors, reduced manual efforts, and increased the overall reliability of network operations, allowing network teams to focus on higher-value, strategic tasks.

2. Conclusion

We have presented a novel approach to configure VLANs across the entire network.

The project has demonstrated that it is possible to abstract the configuration from the devices and present it in a user-friendly GUI inventory system. The reconciliation of the desired and running configurations is functioning well and is readable.

The project has shown that it is possible to configure VLANs across the entire network using a few clicks in our GUI inventory system, which is quite effective. We are confident that the project can be expanded to include more use cases and more devices. Although we need to be aware of the limitations of working with YANG models, which we have to use to build correct configurations for the devices. They support fast and reliable reconciliation once they work, but YANG models can be complex due to device deviations and vendor-specific implementations. OpenConfig YANG models alleviate this limitation to some extent, but as we encountered, they are not always correctly implemented on the devices.

2.1. Further Improvements

Future work could focus on integrating support for additional device types and expanding the whole system to accommodate more complex network configurations. Additionally, there is potential to simplify the handling of YANG models by developing tools that automatically adjust for vendor-specific deviations, thereby enhancing the overall expandability and scalability of the system.

For further improvements, our project could be enhanced by:

- Extending functionality beyond Virtual Local Area Network (VLAN) to include IP management, routing protocols, and more.
- Centralizing support for additional network device types beyond Cisco, such as Juniper, Arista, Nokia, or Huawei.
- Evaluating whether comparing the desired state with the current state of the network could be improved by using Pydantic models.

3. Project Planning

3.1. Processes

Our project is structured using an agile project management framework. This approach is particularly suitable for our needs, as we work with Epics and have a meeting with our advisor every week. This allows us to stay flexible for changes.

We have organized our project into Phases (in our Jira called Phases), which help us manage our tasks and maintain clarity throughout the project lifecycle. Each Phase represents a significant component of our work and is further broken down into Tasks and Subtasks. This structured approach enables us to effectively track progress and ensure that all aspects of the project are addressed.

3.2. Architectural Roles

We designated **Architecture Agents** to leverage each team member's expertise in different technologies, fostering knowledge sharing throughout the project. Initially, we held a meeting to discuss the architecture and assign specific responsibilities. A later chapter will detail the reasoning behind our architectural decisions.

1. **Simon Linder:** Implements and manages the Nornir automation framework, handling device configuration tasks, including fetching the current network state and applying changes.

2. **Polina Lisetska:** Develops and maintains Infrahub, the single source of truth for network configurations and user interactions.

3. **Ramon Stutz:** Responsible for Pydantify, creating Python Pydantic models from YANG definitions, and developing an XMLExporter to generate Netconf XML for device configurations.

This distribution leverages each member's strengths while promoting collaborative decision-making within our team of three developers. It's essential that everyone contributes their technical expertise without feeling overruled, collectively shaping the project's architecture. Although primary tasks are assigned based on experience (Simon with **Nornir** automation, Polina with **Infrahub** development, and Ramon with **Pydantify** for YANG model handling), we work closely together. This vertical slicing allows efficient domain-specific development while ensuring individual systems integrate seamlessly. We update each other at least twice weekly on current statuses and challenges, ensuring mutual support and meaningful contributions, especially in architectural decisions or cross-domain challenges.

3.3. Meetings

We conduct weekly check-ins with our supervisor to address any challenges, deviations, or uncertainties that may arise. Additionally, our team aims to meet at least once a week to foster collaboration and support one another. If necessary, we can hold more frequent meetings to ensure we stay aligned and address any pressing issues promptly.

3.4. Phases

In our project, we have established a structured approach to planning by defining Phases as overarching goals, complemented by high-level tasks. Each Phase is further decomposed into specific tasks and subtasks, facilitating a clear path toward project completion. The timeline allocated for each Phase ranges from 1 to 3 weeks, with each Phase assigned to a designated team member responsible for its execution.

Phase-ID	**Name**	**Description**	**Due Date**	**Assignee**
ACENIWIPN-9	Setup Environment	We set up our environment and build our architecture to test the dependencies.	13.10.2024	Simon Linder
ACENIWIPN-1	Create Infrahub schema	Write an Infrahub schema that matches the requirements	03.11.2024	Polina Lisetska
ACENIWIPN-7	Developing Netconf Script	Develop Netconf script which pulls information from the current network devices and changes their configurations.	03.11.2024	Simon Linder
ACENIWIPN-5	XML-Export	Develop a generic Python script to translate a Pydantic model into a Netconf valid XML	03.11.2024	Ramon Stutz
ACENIWIPN-8	Create Working Prototype	Sets up a working prototype	17.11.2024	Ramon Stutz
ACENIWIPN-2	Create Infrahub Artifacts	Create an Infrahub Artifact Definition	17.11.2024	Polina Lisetska
ACENIWIPN-6	Nornir Conditional Runner	Create a Nornir Runner which runs its Runner parallel but with conditions	22.11.2024	Simon Linder

Table 3.1.: Project Phases Overview

Phase-ID	**Name**	**Description**	**Due Date**	**Assignee**
ACENIWIPN-3	Extend the Python Script to load data from Netconf XML	Extend the Python script to load the data from a Netconf XML into a Pydantic model.	01.12.2024	Ramon Stutz
ACENIWIPN-4	Pull Information from Infrahub	Create a function which pulls information from the Infrahub RESTCONF API and loads it into the Pydantic structure	01.12.2024	Polina Lisetska
ACENIWIPN-69	Setup Prototype on K8s Cluster	Deploy the prototype to the K8s cluster	07.12.2024	Simon Linder
ACENIWIPN-19	Testing	Test our script	15.12.2024	Polina Lisetska, Ramon Stutz, Simon Linder
ACENIWIPN-12	Documentation	Create and document our project.	18.12.2024	Polina Lisetska, Ramon Stutz, Simon Linder
ACENIWIPN-14	Project organization	Organize our projects	18.12.2024	Ramon Stutz

Table 3.2.: Project Phases Overview 2

3.4.1. Time Table

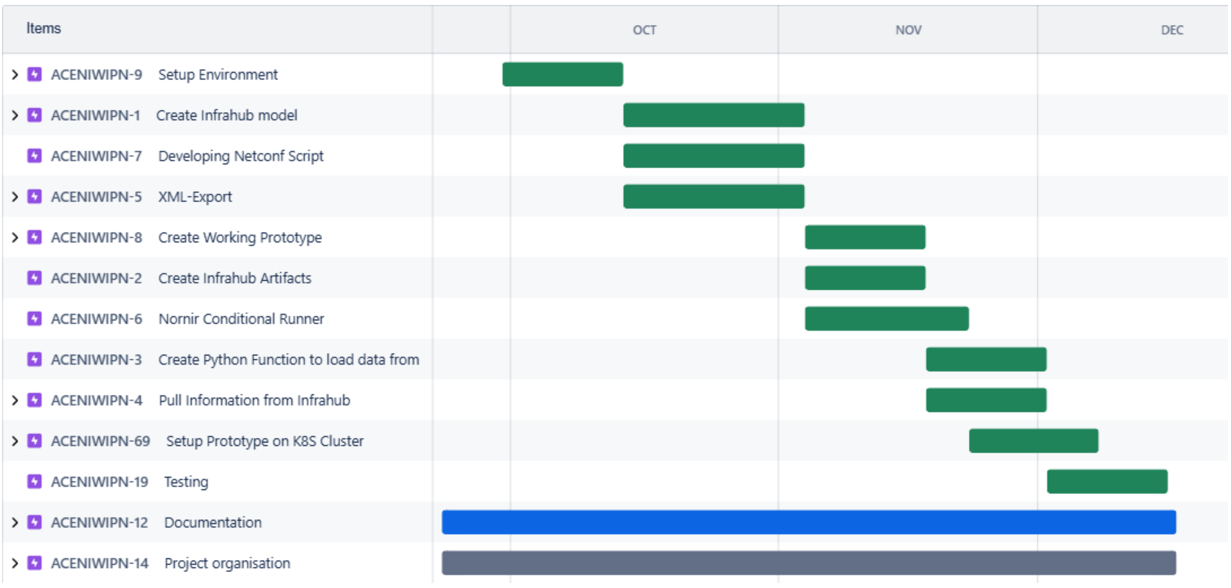


Figure 3.1.: Time Plan of the project

3.5. Risk Management

Like in every other project, risks are always part of it. Our job is to assess, analyze, manage, and if possible, minimize as many risks as possible. Undetected risks could pose a major threat to the success of the project.

3.5.1. Risks

1. **Scope Creep:** Uncontrolled changes or additions to project requirements can lead to scope creep, causing delays or overruns.

2. **Team member falls out:** A team member is absent and isn't capable of working on the project.

3. **Team dynamics:** Poor communication, collaboration issues, or lack of cohesion within the project team can hinder progress and affect project morale.

4. **Quality assurance challenges:** Ineffective testing practices or inadequate quality assurance measures can result in undetected defects, leading to product failures or customer dissatisfaction.

5. **Project management challenges:** Inadequate processes for updating our team project status or updating the timetable can lead to misunderstandings and hinder progress.

6. **Poor Requirements Management:** Inadequate gathering, documentation, or management of project requirements can lead to misunderstandings, rework, and dissatisfaction with the final product.

7. **Technical challenges:** Complex technical requirements, dependencies, or limitations can pose challenges during development, leading to delays or compromised quality.

3.5.2. Risk Countermeasures

Scope Creep

1. Define clear project requirements and objectives from the outset.
2. Regularly review project scope with stakeholders to ensure alignment.
3. Educate stakeholders about the impact of scope changes on project timelines and budgets.

Poor Requirements Management

1. Document requirements clearly and comprehensively.
2. Discuss and review requirements with stakeholders to ensure alignment.

Technical challenges

1. Break down complex tasks into smaller, manageable components.
2. Seek input from subject matter experts and consider alternative solutions.
3. Implement a light version of our desired architecture first to ensure functionality between the versions.
4. Allocate sufficient resources and time for addressing technical challenges.
5. Conduct a comprehensive technical feasibility study before project initiation.

Team dynamics

1. Foster open communication and collaboration within the team.
2. Address conflicts and misunderstandings promptly and constructively.
3. Provide opportunities for team-building activities and training.
4. Assign roles and responsibilities clearly to avoid ambiguity.
5. Meet weekly to discuss the tasks and problems.

Quality assurance challenges

1. Develop a comprehensive testing strategy and plan.
2. Don't review your own work; review each other's work.
3. Plan enough time for testing.
4. Do reviews during the project, not all at the end.

Project management challenges

1. Assign a role to a team member who takes time to update the project management tool (Jira)
2. Weekly meetings to discuss and assign active and new tasks.

Team member falls out

1. Open conversation and weekly updates on the status of tasks so someone could take over.
2. Properly and up-to-date documentation of the tasks.

3.5.3. Risk Matrix

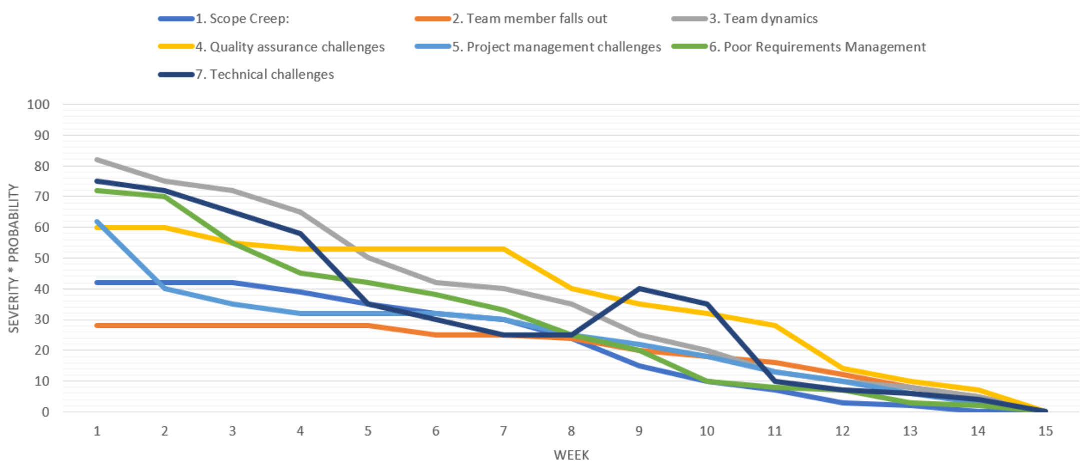


Figure 3.2.: Risk Management Matrix, also showing the technical challenge "conversion to Cisco native model"

3.5.4. Risk summary

In the first weeks of the project, we identified and assessed potential risks. We took some risk measures to minimize them, but unfortunately one risk occurred at week 9 - Technical challenges. We had to adapt our code to the Cisco Native Model. Because of the risk measures we took, we were able to manage this risk and successfully finish our project.

3.6. Planning Tools

3.6.1. JIRA

We use Jira because all team members are familiar with it, allowing us to manage our projects effectively using Epics, Tasks, and Subtasks. Jira also enables us to create timetables, ensuring clear visibility into project timelines and responsibilities. This familiarity and structured approach enhance collaboration and streamline our project management processes.

3.6.2. Clockify

We use Clockify because it integrates seamlessly with Jira, providing us easy time tracking on our tasks. Its graphical reports and filter functions allow us to analyze how time is spent on various tasks and promote accountability.

3.6.3. Overleaf

Overleaf is an online collaborative platform that simplifies the process of writing and publishing documents using LaTeX. It allows multiple authors to work simultaneously on a document, making it ideal for academic and technical writing. Overleaf's built-in templates and real-time preview features streamline the formatting process, enabling us to focus on content creation while ensuring high-quality output suitable for publication.

List of Tables

List of Figures

Acronyms

ACME	Automated Certificate Management Environment. 61
API	Application Programming Interface. 9
CI/CD	continuous integration and continuous deployment. iv, vi, viii, 14, 15, 17, 19, 27, 29, 52, 56
CLI	Command Line Interface. 26, 59, 60, 79
CPU	Central Processing Unit. 55
DB	Database. 11
DNS	Domain Name System. 61
Enum	Enumeration. 33
GUI	Graphical User Interface. 26, 42, 71
HTML	Hypertext Markup Language. 23
HTTPS	Hypertext Transfer Protocol Secure. 11, 60
IaC	Infrastructure as Code. iii, 9, 10, 11, 27, 29, 79
IP	Internet Protocol. 43, 71
JSON	JavaScript Object Notation. 12, 56
NETCONF	Network Configuration Protocol. i, ii, iii, viii, 3, 8, 9, 11, 12, 13, 15, 19, 20, 21, 22, 26, 29, 34, 37, 38, 50, 63, 64, 70
NFS	Network File System. 61
SDK	Software Development Kit. 21
SMTP	Simple Mail Transfer Protocol. 9
TLS	Transport Layer Security. 61
UI	User Interface. 9, 11, 44
URL	Uniform Resource Locator. 67
VLAN	Virtual Local Area Network. i, iv, 2, 6, 11, 16, 17, 24, 30, 31, 36, 41, 42, 43, 50, 51, 52, 53, 64, 67, 70, 71, 79, 86
XML	Extensible Markup Language. i, vii, viii, 9, 11, 12, 13, 16, 19, 20, 21, 22, 26, 33, 34, 35, 36, 37, 38, 39, 40, 41, 50, 63, 64, 65, 68, 70, 72, 79

Glossary

GitLab GitLab is a web-based DevOps lifecycle tool that provides a Git repository manager, wiki, issue-tracking, and CI/CD pipeline features, provided by our university.. 42, 45

GraphQL GraphQL is a data query and manipulation language for APIs that allows a client to specify what data it needs ("declarative data fetching"). Also known under the pattern name "WishTemplate Pattern".. 8, 11, 45, 46, 50, 51

INS Institute of Network and Security, a research institute at the University of OST, which is our project partner.. 2, 4, 25, 54

NcDiff NcDiff is a tool that compares two NETCONF configuration files and generates a patch file that can be used to update the first file to match the second.. 22

PEP8 PEP8 is a style guide for Python code. It is a set of rules that specify how to format Python code for maximum readability.. vi, 14, 15

YANG YANG is a data modeling language used to model configuration and state data manipulated by a NETCONF agent.. i, ii, iii, vii, 9, 11, 12, 13, 19, 20, 22, 25, 26, 29, 30, 31, 32, 33, 34, 35, 37, 64, 67, 68, 70, 71, 72, 85

Part IV.

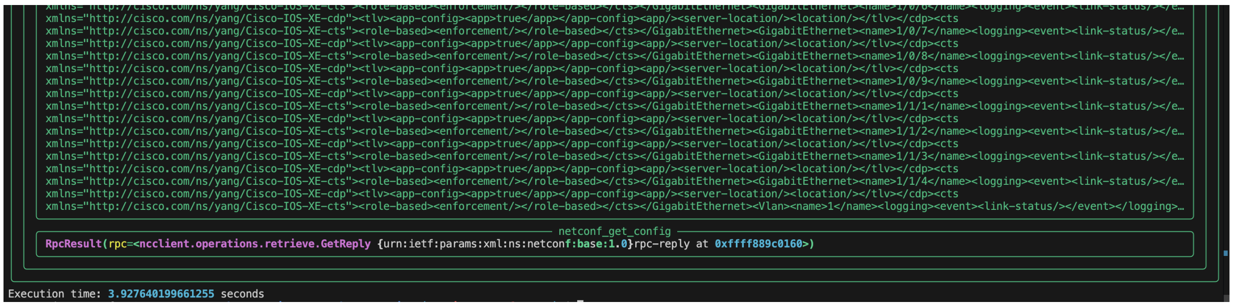
Appendix

1. Nornir Connection Plugin Analysis

1. The connection plugin should be able to communicate with our CISCO IOS-XE devices as well as Arista EOS devices, if possible, as we plan to conduct integration tests on virtual Arista devices.
2. It should manage the opening and closing of the connection independently.
3. It should be performant and reliable.
4. It should be easy to use and integrate with the Nornir framework.
5. It should be well-documented.
6. It should be actively maintained.
7. It should support methods to get a configuration, a part of a configuration, lock a configuration, unlock a configuration, edit a configuration, commit a configuration, and validate a configuration, so we can implement dry run logic.
8. It should return the output in a structured format, such as XML or JSON and attach the rpc oject for later use of the connection or advanced use.

We looked at 3 NETCONF plugins and libraries, such as Nornir-Netconf, NCDiff, and Scrapli.

1. **Nornir-Netconf:** Is a plugin for the Nornir automation framework that provides a high-level API for interacting with network devices using the NETCONF protocol.
   - **Supported for / Tested on:** Integration Tests Devices with full integration tests with ContainerLab
     - Nokia SROS - TiMOS-B-21.2.R1
     - Cisco IOSxR - Cisco IOS XR Software, Version 6.1.3
     - Cisco IOSXE - Cisco IOS XE Software, Version 17.03.02
     - Arista CEOS - 4.28.0F-26924507.4280F (engineering build)
   - **Transport:** SSH
   - **Supported Methods:**
     - netconf_capabilities - Return server capabilities from target -> Result.result -> RpcResult
     - netconf_edit_config - Edits configuration on specified datastore (default=”running”) -> Result.result -> RpcResult
     - netconf_get - Returns state data based on the supplied xpath -> Result.result -> RpcResult
     - netconf_get_config - Returns configuration from specified configuration store (default=”running”) -> Result.result -> RpcResult
     - netconf_get_schema - Retrieves schemas and saves aggregates content into a directory with schema output -> Result.result -> SchemaResult
     - netconf_lock - Locks or Unlocks a specified datastore (default=”lock”) -> Result.result -> RpcResult



```
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cts"><role-based><enforcement/></role-based></cts></GigabitEthernet><GigabitEthernet><name>1/0/6</name><logging><event><link-status/></e…
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cdp"><tlv><app-config><app>true</app></app-config><app/><server-location/><location/></tlv></cdp><cts
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cts"><role-based><enforcement/></role-based></cts></GigabitEthernet><GigabitEthernet><name>1/0/7</name><logging><event><link-status/></e…
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cdp"><tlv><app-config><app>true</app></app-config><app/><server-location/><location/></tlv></cdp><cts
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cts"><role-based><enforcement/></role-based></cts></GigabitEthernet><GigabitEthernet><name>1/0/8</name><logging><event><link-status/></e…
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cdp"><tlv><app-config><app>true</app></app-config><app/><server-location/><location/></tlv></cdp><cts
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cts"><role-based><enforcement/></role-based></cts></GigabitEthernet><GigabitEthernet><name>1/0/9</name><logging><event><link-status/></e…
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cdp"><tlv><app-config><app>true</app></app-config><app/><server-location/><location/></tlv></cdp><cts
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cts"><role-based><enforcement/></role-based></cts></GigabitEthernet><GigabitEthernet><name>1/1/1</name><logging><event><link-status/></e…
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cdp"><tlv><app-config><app>true</app></app-config><app/><server-location/><location/></tlv></cdp><cts
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cts"><role-based><enforcement/></role-based></cts></GigabitEthernet><GigabitEthernet><name>1/1/2</name><logging><event><link-status/></e…
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cdp"><tlv><app-config><app>true</app></app-config><app/><server-location/><location/></tlv></cdp><cts
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cts"><role-based><enforcement/></role-based></cts></GigabitEthernet><GigabitEthernet><name>1/1/3</name><logging><event><link-status/></e…
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cdp"><tlv><app-config><app>true</app></app-config><app/><server-location/><location/></tlv></cdp><cts
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cts"><role-based><enforcement/></role-based></cts></GigabitEthernet><GigabitEthernet><name>1/1/4</name><logging><event><link-status/></e…
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cdp"><tlv><app-config><app>true</app></app-config><app/><server-location/><location/></tlv></cdp><cts
xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-cts"><role-based><enforcement/></role-based></cts></GigabitEthernet><Vlan><name>1</name><logging><event><link-status/></event></logging>…

netconf_get_config
RpcResult(rpc=<ncclient.operations.retrieve.GetReply {urn:ietf:params:xml:ns:netconf:base:1.0}rpc-reply at 0xffff889c0160>)

Execution time: 3.927640199661255 seconds
```

Figure 1.1.: Nornir_Netconf Result

- netconf_validate - Validates configuration datastore. Requires the validate capability. -> Result.result -> RpcResult
- netconf_commit - Commits a change -> Result.result -> RpcResult
- netconf_rpc

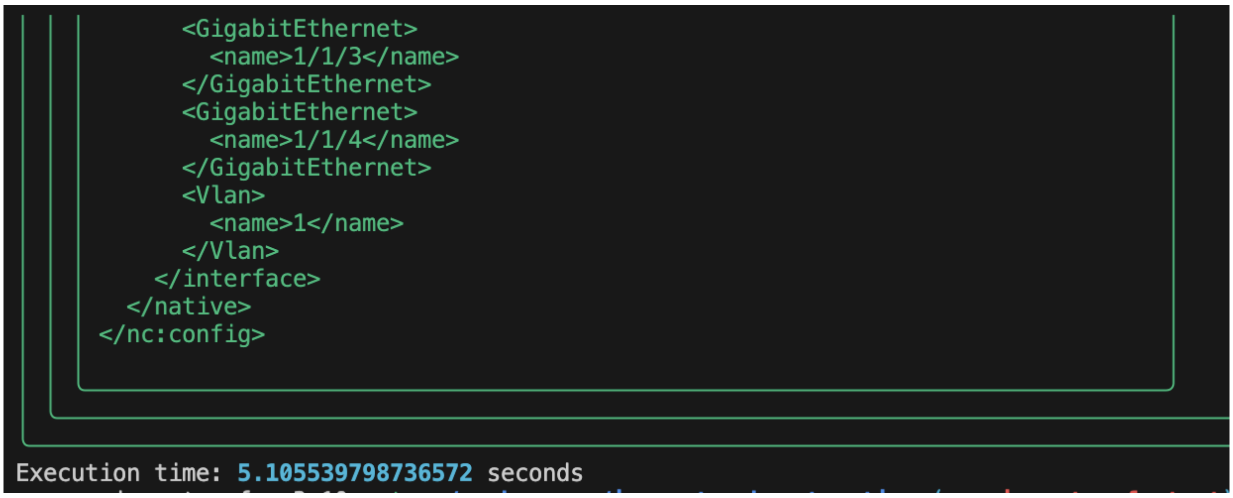
discard is missing but can be implemented with RPC.

- **Performance:**
  - Execution time: 3.8691527843547534 seconds
  - Execution time: 3.896733045578003 seconds
  - Execution time: 3.880706310272217 seconds
  - Execution time: 3.850162982940674 seconds
  - Execution time: 3.915595769882202 seconds
- **Result:** Not formatted, but all data is present in
- **Notes:** The library is based on ncclient, which is a well-known library for NETCONF.

  Nice global locking over multiple tasks is possible.

  Many tests and examples available.

2. **NCDiff:** Is a library that provides a simple way to compare two NETCONF configurations.
   - **Supported for / Tested on:**
     - Suported devices IOSXE and many others
   - **Transport:** SSH
   - **Supported Methods:**
     - netconf_edit_config
     - netconf_get
     - netconf_get_config
     - netconf_lock
     - netconf_unlock
     - netconf_validate
     - diff of models
   - **Performance:** Slow at firs run as if downloads all YANG models
     - Execution time: 249.23009967803955 seconds

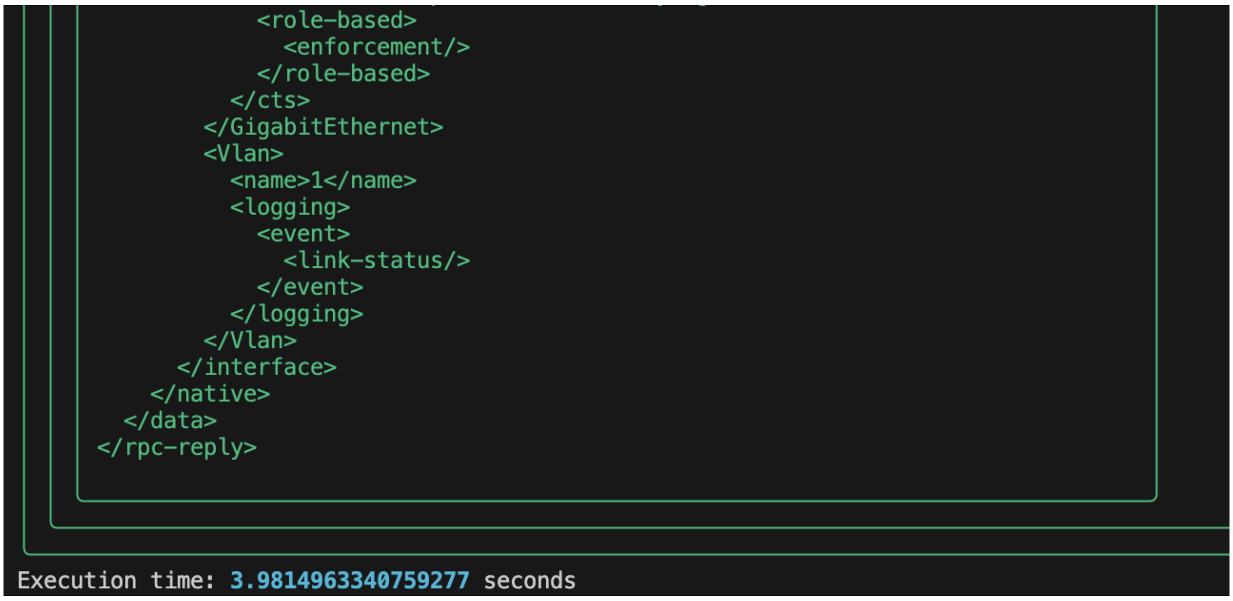


```
        <GigabitEthernet>
          <name>1/1/3</name>
        </GigabitEthernet>
        <GigabitEthernet>
          <name>1/1/4</name>
        </GigabitEthernet>
        <Vlan>
          <name>1</name>
        </Vlan>
      </interface>
    </native>
  </nc:config>

Execution time: 5.105539798736572 seconds
```

Figure 1.2.: NCDiff Result

    - Execution time: 5.105539798736572 seconds
    - Execution time: 5.8046042919158936 seconds
    - Execution time: 5.217963695526123 seconds
    - Execution time: 5.310622453689575 seconds
  - **Result:** nice XML output
  - **Notes:** Not perfect for our use case, as it downloads all YANG models at the first run, which is slow. We only need the models for VLAN.

3. **Scrapli Netconf:** Is a library that provides a simple way to interact with network devices, NETCONF is a plugin for the Scrapli which makes it more complex.
   - **Supported for / Tested on:**
     - Cisco IOS-XE (tested on: 16.12.03) with Netconf 1.0 and 1.1
     - Cisco IOS-XR (tested on: 6.5.3) with Netconf 1.1
     - Juniper JunOS (tested on: 17.3R2.10) with Netconf 1.0
   - **Transport:** SSH2 (Plugin for Scrapli; difficult to get running (Key alg))
   - **Supported Methods:**
     - get - with "subtree" (default) or "xpath" filter
     - get_config - complete config
     - edit_config
     - lock
     - unlock
     - commit
     - discard
     - Delete config
     - RPC
     - capabilities



```
              <role-based>
                <enforcement/>
              </role-based>
            </cts>
          </GigabitEthernet>
          <Vlan>
            <name>1</name>
            <logging>
              <event>
                <link-status/>
              </event>
            </logging>
          </Vlan>
        </interface>
      </native>
    </data>
  </rpc-reply>

Execution time: 3.9814963340759277 seconds
```

Figure 1.3.: Scrapli Netconf Result

Validate is missing but can be implemented with RPC

- **Performance:**
  - Execution time: 3.7733654975891113 seconds
  - Execution time: 3.775163412094116 seconds
  - Execution time: 3.9814963340759277 seconds
  - Execution time: 3.8708913326263428 seconds
  - Execution time: 3.8404557704925537 seconds
- **Result:** nice XML output
- **Notes:** Difficult to configure, as it first runs in to a timeout and then needs to be reconfigured. SSH2 does not work with python 3.12 only with 3.10 (not documented only found this out as pytest skips ssh2 unittests with python 3.12). Cipher missmatch error -> different devcontainer needed.

2. Nornir Configuration

2.1. Installation

```
pip install nornir-conditional-runner
```

2.2. Usage

Replace the default Nornir runner with ConditionalRunner in your configuration:

```
from nornir import InitNornir

nr = InitNornir(
    runner={
        "plugin": "ConditionalRunner",
        "options": {
            "num_workers": 10,
            "group_limits": {
                "core": 1,
                "distribution": 2,
                "edge": 3,
            },
            "group_fail_limits": {
                "core": 1,
                "edge": 2,
            },
            "conditional_group_key": "conditional_groups",
            "skip_unspecified_group_on_failure": True,
        },
    },
    inventory={
        "plugin": "SimpleInventory",
        "options": {
            "host_file": "demo/inventory/hosts.yaml",
            "group_file": "demo/inventory/groups.yaml",
        },
    },
)

def my_task(task):
    return f"Running on {task.host.name}"

result = nr.run(task=my_task)
print(result)
```

2.2.1. Host Example

Hosts can define custom groups in their data dictionary using the `conditional_group_key` provided in the runner options. The runner will use these groups to enforce the `group_limits`.

```
host1:
data:
    conditional_groups:
```

```
    - core
host2:
data:
    conditional_groups:
    - distribution
```

If the `conditional_group_key` is not provided, the runner will default to using the host groups.

```
host1:
groups:
    - core
host2:
groups:
    - edge
```

3. Nornir Configuration

The final configuration of the Nornir tool `defaults.yaml`:

```yaml
# yamllint disable rule:line-length
data:
  mail:
    smtp_server: "smtp.ost.ch"
    port: 25
    sender: "iac_sa@ost.ch"
    to: "Simon.linder@ost.ch"
  yang:
    - interface:
      filter: |
        <native xmlns="http://cisco.com/ns/yang/Cisco-IOS-XE-native">
            <interface>
                <GigabitEthernet>
                    <name></name>
                    <switchport-config>
                    </switchport-config>
                </GigabitEthernet>
                <TenGigabitEthernet>
                    <name></name>
                    <switchport-config>
                    </switchport-config>
                </TenGigabitEthernet>
                <TwentyFiveGigE>
                    <name></name>
                    <switchport-config>
                    </switchport-config>
                </TwentyFiveGigE>
                <FortyGigabitEthernet>
                    <name></name>
                    <switchport-config>
                    </switchport-config>
                </FortyGigabitEthernet>
            </interface>
        </native>
      ids:
        - tag: "{http://cisco.com/ns/yang/Cisco-IOS-XE-native}GigabitEthernet"
          id_tag: "{http://cisco.com/ns/yang/Cisco-IOS-XE-native}name"
        - tag: "{http://cisco.com/ns/yang/Cisco-IOS-XE-native}TenGigabitEthernet"
          id_tag: "{http://cisco.com/ns/yang/Cisco-IOS-XE-native}name"
        - tag: "{http://cisco.com/ns/yang/Cisco-IOS-XE-native}TwentyFiveGigE"
          id_tag: "{http://cisco.com/ns/yang/Cisco-IOS-XE-native}name"
        - tag: "{http://cisco.com/ns/yang/Cisco-IOS-XE-native}FortyGigabitEthernet"
          id_tag: "{http://cisco.com/ns/yang/Cisco-IOS-XE-native}name"
      replace:
        - <switchport-config>
    - vlan:
      filter: |
        <vlans xmlns="http://openconfig.net/yang/vlan">
        </vlans>
      ids:
        - tag: "{http://openconfig.net/yang/vlan}vlan"
          id_tag: "{http://openconfig.net/yang/vlan}vlan-id"
      replace:
        - <vlans xmlns="http://openconfig.net/yang/vlan">
  # config
  xmldiff: false
  etreediff: true
  ncdiff: false
  debug: false


```