POLITECNICO DI TORINO

Master degree course in Computer Engineering

# Improving Observability in Large Enterprise Networks with NetBox And SuzieQ


**Supervisor**
prof. Fulvio Risso

**Candidate**
Riccardo Lucifora

**Internship Tutor**
dott. ing. Luca Nicosia


Anno accademico 2023/2024

This work is subject to the Creative Commons Licence

# Summary

Modern networks are as **complex** as it gets to meet the needs of the market, which is placing ever-greater demands on storage space, speed, flexibility, and availability. However, they have never really been simple, as they are populated by very diverse devices that do not have a global view of the entire network. This design, together with the trend just mentioned, has meant that ever-larger networks cannot be managed efficiently from a single privileged access point.

We can list various "problems" in computer networks, such as their complexity, opacity, and unpredictability. Some concepts have emerged that aim to solve some of these problems: Observability and Automation. This thesis will focus on **network observability**, which is defined as the ability to understand the internal state of a complex system based on external outputs. When a system is observable, a user can identify the causes of a performance problem by looking at the data it produces, without additional testing and coding.

In particular, we will analyze two tools that aim to make networks more observable: SuzieQ and NetBox. **SuzieQ** stands out for its ease of use, its performance, and its ability to summarize heterogeneous information in a normalized "universal" format that is the same regardless of the input. **NetBox** strong suits are the completeness and granularity of the stored information and the great potential it offers in terms of integration with other tools. The graphical user interface is also worth mentioning: it is pleasant to use and complete, which is unusual in this field.

The market is interested in **integrating the two tools**: they can complement each other, enrich the information provided to the user, and mitigate some shortcomings. This work aims to study their data models and then propose a solution to integrate the tools so that two main functions are offered: Synchronization and Validation. The main **challenges** lay in the

architecture and code design, data normalization, and defining the scope of the solution.

All the phases of the study will be outlined, alongside the architecture of the solution and its implementation, the challenges to face, and the reasons for the taken decisions. In the end, the **results** from a qualitative and quantitative perspective – respected standards and elapsed time measurements – will be presented as well as future work, as the core logic behind this solution will be part of a market product.

# Acknowledgements

I want to thank prof. **Fulvio Giovanni Ottavio Risso**, my supervisor. He made me passionate about the subject through his courses and provided me with this outstanding opportunity at **Stardust Systems**. On that, I also want to mention the warm welcome that I found. From the boss to coworkers, I found such a great and genuine team in a relaxed yet dynamic environment. They never let me down when needed and left me all the space I needed for studying. It was never about exploiting me but always about empowering me. I hope my work reflected that.

On a more sentimental note, I wish to extend my special thanks to **my family**, who supported me through these years, between peaks and valleys. I would not be here without them. Last but not least, my appreciation goes to **my friends**. Andreas and Vanessa are almost family, but also Fabio and Isabella left an unparalleled dent in my heart, which will never disappear. There are a lot of people I left out, but I feel grateful for each one of them, and that's why I want them with me when I'm celebrating, as they were there for me when I was at my lowest: thanks to you all.

# Contents


**I Introduction** **9**

**1 General Concepts** 11
1.1 The Logical Level . . . . . . . . . . . . . . . . . . . . . . . . . 11
1.2 Diverse Hardware . . . . . . . . . . . . . . . . . . . . . . . . . 13
1.3 Diverse Software: Examples . . . . . . . . . . . . . . . . . . . 15

**2 Problem Statement** 19
2.1 The Challenges . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.2 A Solution: Network Observability . . . . . . . . . . . . . . . 21

**II Network Observability Tools** **23**

**3 Related Work** 25
3.1 The Market . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.2 Cisco ThousandEyes . . . . . . . . . . . . . . . . . . . . . . . 25
3.3 Paid Alternatives . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.4 Free Open-Source Alternatives . . . . . . . . . . . . . . . . . . 32
3.5 Integration For Automation: NetBox And Ansible . . . . . . . 33

**4 NetBox** 37
4.1 Functional Scope . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.2 Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.3 Data Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

**5 SuzieQ** 41
5.1 Main Components And Functions . . . . . . . . . . . . . . . . 41
5.2 Data Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

**III NetBox And SuzieQ Integration** **45**

**6 Preliminary Studies** 47
6.1 Reasoning . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
6.2 Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . 49

**7 Architecture** 51
7.1 Functions and Scope . . . . . . . . . . . . . . . . . . . . . . . 51
7.2 Methodology And Complexity . . . . . . . . . . . . . . . . . . 52
7.3 Logical Modules . . . . . . . . . . . . . . . . . . . . . . . . . 55

**8 Implementation** 59
8.1 Language and Libraries . . . . . . . . . . . . . . . . . . . . . 59
8.2 Files And Directory . . . . . . . . . . . . . . . . . . . . . . . 61
8.3 The Facade . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
8.4 Main Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
8.5 Translation Logic . . . . . . . . . . . . . . . . . . . . . . . . 65
8.6 Code Logic And Methods . . . . . . . . . . . . . . . . . . . . 66
8.7 Fine Tuning . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

**IV Results** **73**

**9 Evaluation** 75
9.1 Performance And Scalability . . . . . . . . . . . . . . . . . . 78
9.2 Good Programming . . . . . . . . . . . . . . . . . . . . . . . 82

**10 Conclusions and Future Works** 85
10.1 Future Works . . . . . . . . . . . . . . . . . . . . . . . . . . 85

# Part I

# Introduction

# Chapter 1

# General Concepts

On the surface, network operations may **seem** straightforward - numerous devices communicating seamlessly through a standardized 'language' defined by protocols. The network is always up and running and we struggle to ask ourselves: how? There's an intricate complexity that lies beneath, making any network a **multifaceted heterogeneous system**. To get to the bottom of this, we must revisit the fundamentals.

## 1.1 The Logical Level

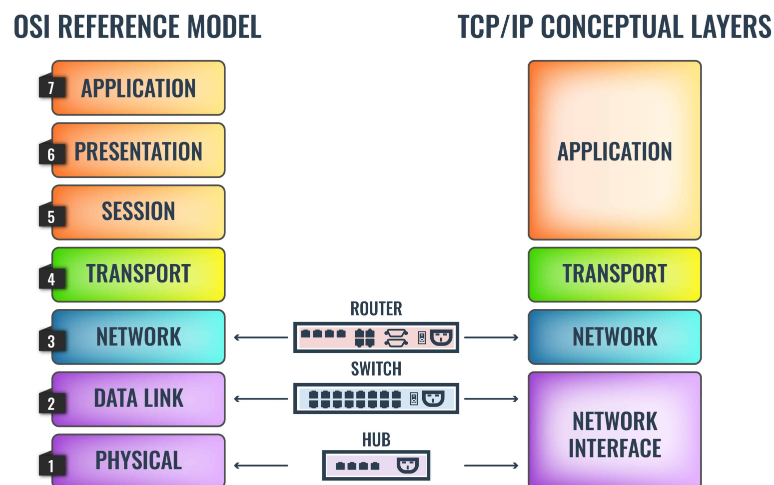



Figure 1.1. Traditional OSI stack vs TCP/IP stack. (1)

A network can be seen as a layered structure, following the OSI Model (1):

1. **Application (level 7)** - this is the most abstract layer, at the top of the stack. It serves as a way for users to interact with the system. Some known protocols from here are DNS (Domain Name System), HTTP (Hyper Text Transfer Protocol), SSH (Secure Shell), POP (Post Office Protocol);

2. **Transport (level 4)** - this layer ensures a reliable data transmission. Its key protocols are TCP (Transmission Control Protocol) which is slower and more reliable and UDP (User Datagram Protocol) which is faster but less reliable. Using one or the other depends on the task at hand: UDP is preferred for real-time tasks where reliability is not crucial and data loss is tolerable e.g. streaming, VoIP; TCP is used when reliability is a key point e.g. data transfer, email;

3. **Network (level 3)** - this layer decides the most efficient routes for packets. This operation is called routing. The main protocols here are IP (Internet Protocol), probably the most well-known, in its versions (IPv4 and IPv6); OSPF (Open Shortest Path First); BGP (Border Gateway Protocol);

4. **Data link (level 2)** - this layer makes frames flow between nodes reliably, on the same physical network layer over a shared medium. The main protocols here are MAC (Medium Access Control); LLDP (Link Layer Discovery Protocol); ATM (Asynchronous Transfer Mode); and MPLS (Multiprotocol Label Switching). MAC won over more reliable protocols because of its simplicity and reduced costs;

5. **Physical (level 1)** - this is the most concrete layer, at the base of the stack. It transmits bits over the physical medium e.g. cables, optical fibers, and the air using different kinds of signals: electrical, optical, and electromagnetic.

**The bottomline** - We have just outlined what's useful to our discussion, hence the omission of levels 5 and 6 should not be surprising. The point is that we have distinct **layers**, each one equipped with its own **protocols**; each layer interacts with the ones immediately above and below it through specific methods.

As the OSI Model provides a comprehensive, high-level perspective of the **logical** layers involved in network design, it's important to note that we're

just scratching the surface. The complexity deepens when we closely analyze the **diverse range of devices, protocols, and formats** that come into play. Within each one of these aspects, there's also a multitude of commercial alternatives to take into account.

## 1.2 Diverse Hardware

Let's dig deeper into the hardware part of the picture. A modern network can include:

1. **Routers** - Routers, operating at the 3rd level of the OSI Model, direct data traffic based on the destination IP address. This process utilizes a Routing Table, which is designed to require as little space as possible;

   Pure routing devices are rare, as they often incorporate **additional functionalities**. These include security e.g. embedded firewall and filtering software; Wi-Fi capabilities; device connection e.g. printers reachable through the network, IP phones;

   Routers can also significantly **vary their behavior** in terms of performance, hardware ports, and protocols. For instance: wired routers share data over cables, while wireless routers use antennas; in large-scale networks, core routers require higher performances;

2. **Switches** - Switches channel data to the correct device within a network. This operation requires using a data structure called a Filtering Database. Unlike routers, switches operate within a single network;

   Network switches have experienced **significant advancements and diversification** over time, to meet the changing needs of networking. Not all the ideas that emerged over the years made the cut, but they all helped shape the range of modern products we can see on the market. It's also interesting to notice some crossover between networking and different fields, with vendors and researchers trying to figure out how to progress further and further;

   For example, one innovation that was anticipated but didn't fully materialize (due to high costs) was the use of **Network Processing Units** (NPUs). The concept of NPUs, though, is tightly related to GPUs as they both rely on heavy parallelization.

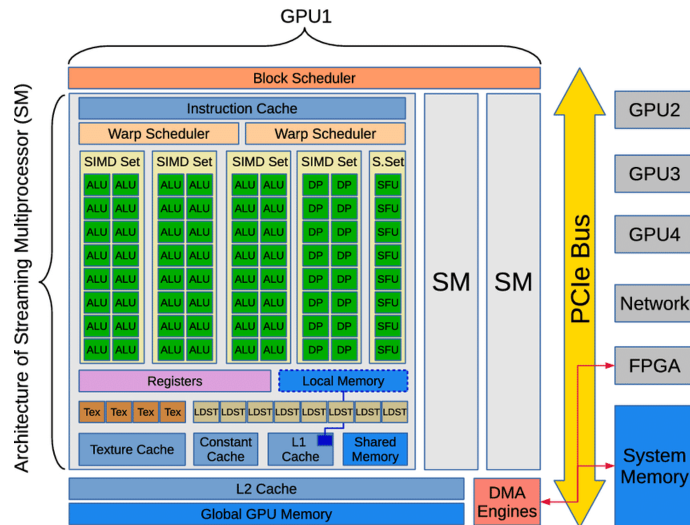



Figure 1.2. Generalized scheme of GPU architecture.

Even **network accelerators** for switches have been influenced by the design of GPUs. For instance, NVIDIA's Spectrum Ethernet switches, used to accelerate hyperscale generative AI fabrics, are a testament to this influence. Accelerators are hardware components that make sense the most for easy, repetitive operations that would be costly to perform in software. A fine example is security accelerators, which perform encryption and decryption.

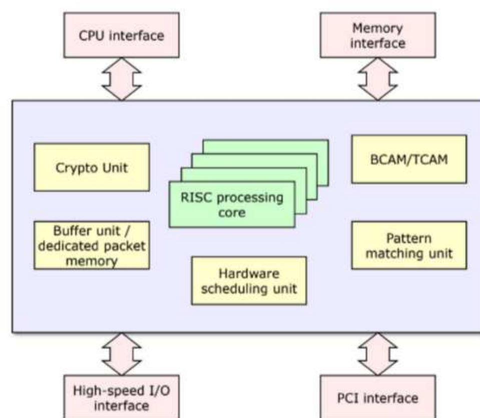



Figure 1.3. A switch and its accelerators.

3. **Servers** - A server is a hardware or software component that provides functionalities ("Services") to other components, called "clients". There's a many-to-many relationship between clients and servers. Servers are often stacked in data centers, providing a powerful platform to **share data** as well as other resources and **distribute workloads**, moving jobs around transparently to users. (2)

   The **client-server model** is ubiquitous on the Internet: millions of servers are always connected, up and running, and virtually every user action requires some degree of interaction with one or more servers.

   Servers can hugely vary based on their **purpose** e.g. database servers, file servers, mail servers, computing servers, web servers, etc., and on their **hardware requirements** even if generally they're more powerful and expensive than the clients that connect to them.

4. **Much more** - A network also includes firewalls (security guards); dedicated storage units; cables from electrical to optical ones; power units etc.

## 1.3 Diverse Software: Examples

Networking devices mostly rely on CLIs (Command Line Interface) to be interacted with. That means a lot of work for network administrators to understand how to use the same commands on different devices from different vendors. Also because of that, using CLIs is typically **error-prone**. GUIs (Graphical User Interface) sure are more intuitive, but they're quite rare in this field. Let's see some examples.

**Display the IP interface brief:**

- Cisco IOS `show ip interface brief`
- Juniper Junos OS `show interfaces terse`
- Huawei `display ip interface brief`
- Arista EOS `show ip interface brief`

**Display the routing table:**

- Cisco IOS `show ip route`
- Juniper Junos OS `show route`
- Huawei `display ip routing-table`
- Arista EOS `show ip route`

**Display detailed information about an interface:**

- Cisco IOS `show interfaces GigabitEthernet0/0`
- Juniper Junos OS `show interfaces ge-0/0/0 extensive`
- Huawei `display interface GigabitEthernet 0/0/0`
- Arista EOS `show interfaces Ethernet1 detail`

**Configure an IP address on an interface:**

- Cisco IOS
```
interface GigabitEthernet0/0
ip address 192.168.1.1 255.255.255.0
no shutdown
exit
```
- Juniper Junos OS
```
edit interfaces ge-0/0/0
set unit 0 family inet address 192.168.1.1/24
commit
exit
```
- Huawei
```
interface GigabitEthernet 0/0/0
ip address 192.168.1.1 255.255.255.0
undo shutdown
quit
```
- Arista EOS
```
interface Ethernet1
ip address 192.168.1.1/24
no shutdown
exit
```

Please keep in mind that the exact commands can vary slightly depending on the specific model and software version of the network device.

# Chapter 2

# Problem Statement

## 2.1 The Challenges

Having revisited the basics, it should be clear now how complex the situation is: we have very different devices that must work together to pursue a common goal, and even within a category, different commands are used and different hardware and software capabilities must be taken into account.

We will analyze the drawbacks that this complexity brings, one by one, and, in the next chapter, also try to understand how we can solve at least some of those, to design networks that are increasingly smart, reliable, and easy to administrate.

1. **Complexity** - A network administrator has to coordinate many different devices that may have different protocols, architectures, and functions. This makes it hard to design, implement, and maintain a distributed system that works reliably and efficiently.

   Configuration is also a pain: CLIs vary significantly by vendor and frequent changes to the CLI structure and syntax make it difficult to maintain CLI scripts. The output of commands is also structure-agnostic, unpredictable, and prone to changes, causing great difficulties in automatically parsing CLI scripts.

2. **Opacity** - We may think of a network as a whole, but devices only have a partial and local perspective and may use different formats and units. This means that: it's difficult to aggregate and compare their resources and capabilities; identifying and isolating the source and impact of a problem can be quite challenging.

3. **Unpredictability** - There's a huge variety of failures, delays, and errors that are hard to anticipate and handle. Moreover, they're detrimental to the performance, correctness, and security of the system. Some examples:

- Device crashing;
- Messages being lost or corrupted;
- Network links being congested or broken;
- Malicious attacks.

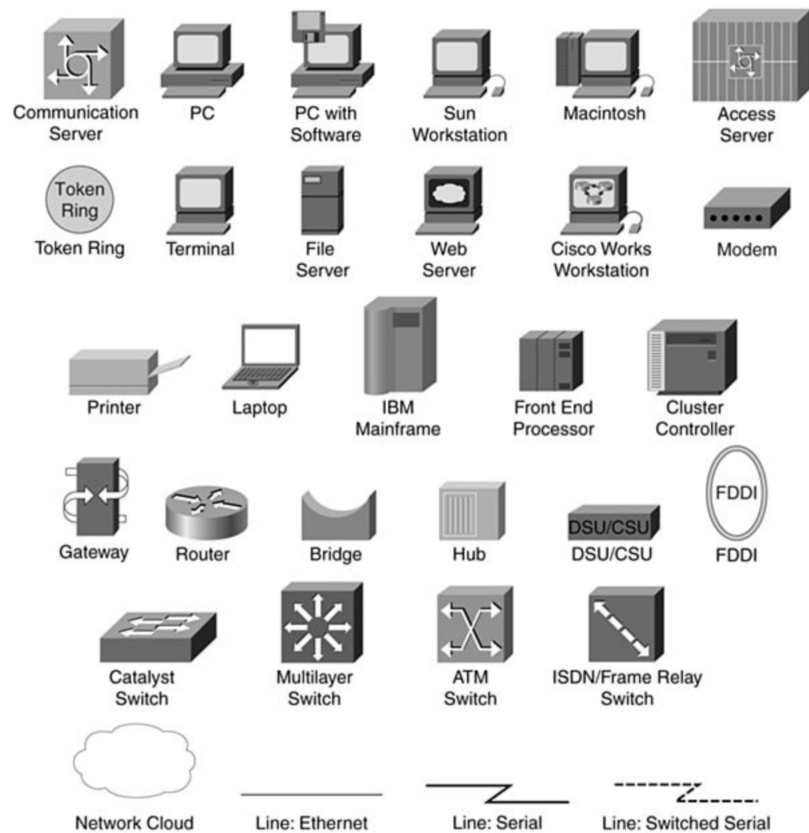



Figure 2.1. Complexity - a single network, a wide array of devices. (3)

## 2.2 A Solution: Network Observability

Network observability can mitigate the aforementioned problems. Network observability tools utilize multiple data sources to provide quick, meaningful insights into the underlying functions of the network. This makes it easier for network operators to **remediate issues** and **improve performance** because they don't have to understand the raw data.

This approach is in stark contrast to the traditional approach, where data is only collected and presented without additional context and with low granularity. The deeper, **more powerful architectural design** of network monitoring solutions makes the market for this kind of solution very large. (4) (5)

In addition, these tools play a crucial role in **proactive network management**: by analyzing the data collected, they can help predict potential problems before they occur and outline the necessary preventative measures. This can help to reduce downtime and increase the overall reliability of the network.

Network observability tools can also provide valuable insights into **network usage patterns**. This information can be used to make informed decisions about capacity planning and infrastructure investment. Last but not least, by observing common patterns, these tools can identify **security threats** that violate these patterns.

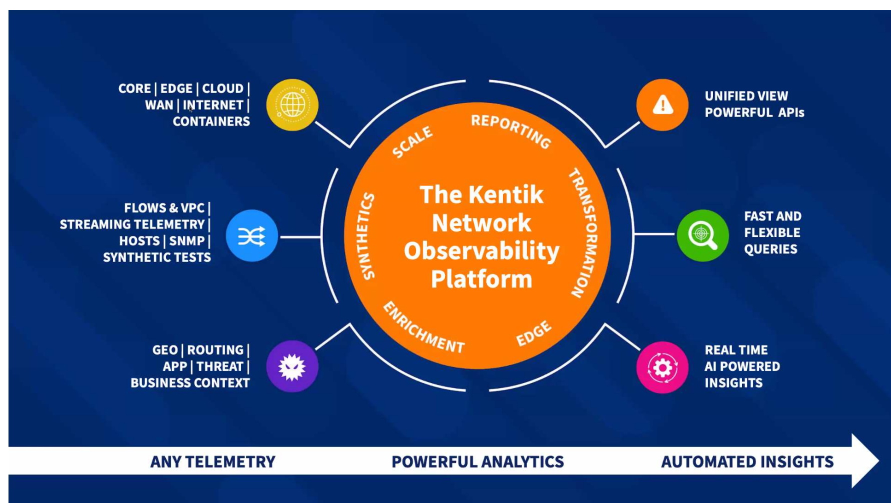



Figure 2.2. Network Observability Platforms and their capabilities. (6)

# Part II

# Network Observability Tools

# Chapter 3

# Related Work

Of course, the choice of SuzieQ and NetBox was implied when writing a thesis at Stardust Systems. Be that as it may, it's still worth analyzing what is out there in this field. We will first take a look at other network observability suites - both free and paid - and then at a quite **similar software solution**: NetBox integration within Ansible.

## 3.1 The Market

The market for observability platforms is expected to show **astonishing growth** in the medium term. Increasing adoption of cloud-based solutions for service virtualization, containerization, and other purposes by various small to large enterprises is a cornerstone for the growth of the market. (5)

Lower cost of setting up and maintaining process automation coupled with high demand for improved operational efficiency and automated implementation of business processes across various verticals are driving the market for observability platforms.

However, the complexity of the observability platform implementation process and the lack of standardized solutions are expected to reduce the potential growth of the market.

## 3.2 Cisco ThousandEyes

Let's start with Cisco, which is considered one of the market leaders in this field (4) also because of recent heavy investments they made. (7)

ThousandEyes is a cloud-based networking suite that allows for real-time, end-to-end views across all internal, external, carrier, and internet networks for large enterprises.

**Features** - The product offers various features such as:

- Browser-based Real User Monitoring - Monitor the performance of web applications from the user's perspective via a browser plugin. This feature enables the automatic collection of user interactions for various applications such as Salesforce or Office 365 Suite. Detailed metrics are displayed along with end-to-end network connectivity, including WiFi access points, to understand the user's entire journey to business-critical applications.

- On-demand and scheduled network synthetics - Proactively run synthetic monitoring tests to stay on top of issues and achieve the right business outcome. Visualize end-to-end Layer 3 network connections between users and SaaS and internal applications. Receive alerts on connectivity and availability issues so you can de-escalate them faster and increase user satisfaction.

- Wireless network visibility - With users constantly on the move, it can be difficult to detect and isolate WiFi issues. Knowing some key metrics about their connection, such as signal quality, connection speed, congestion, etc., can help you speed up troubleshooting and provide a good digital experience for your employees, no matter where they are.

- Automated session testing - Gain deep insights into the real-time performance of business-critical applications with automated session testing. Because collaboration applications dynamically select the best infrastructure components to connect to users based on different metrics, monitoring this environment must be dynamic. By automatically creating tests for each user and session with the active infrastructure components, you can monitor and troubleshoot the system way faster.

**Pros and cons** - While the product is very powerful, complete, and user-friendly with a well-designed dashboard - as can be seen in Figure 3.1 - the pricing is not transparent and can be quite expensive. The documentation is also difficult to navigate. (8)

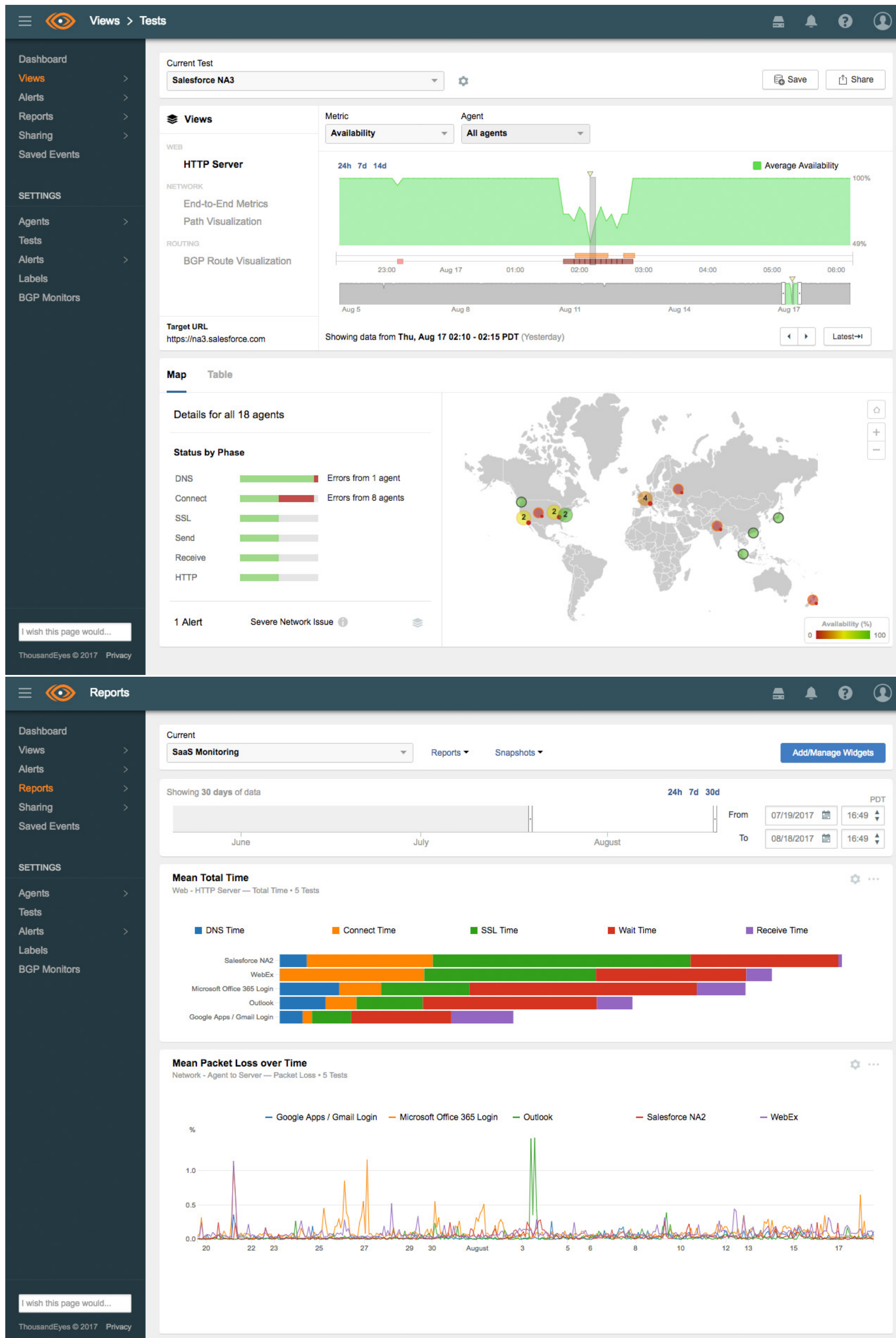



Figure 3.1. Schreenshots from Cisco ThousandEyes. (8)

## 3.3 Paid Alternatives

As already mentioned, the market for network observability suites is quite lively, so you can choose between a variety of paid products. (9) Some of the most commonly used and/or well-documented ones will be presented here, to see how they differ in terms of proposed features, price, and ease of use.

**HoneyComb** - HoneyComb is designed to give engineering teams the visibility they need to troubleshoot distributed systems. It is a cloud-based tool with support for events, logs, and traces. If your code is not yet instrumented, Honeycomb provides an automatic instrumentation agent called Honeycomb Beelines that takes care of that. It also supports OpenTelemetry for instrumentation data.

**Pros and Cons** - HoneyComb enables granular insights because it captures detailed data. Sampling is dynamic to make operations more efficient depending on network conditions. It also provides shared workspaces to encourage team collaboration and solve problems faster. However, the learning curve can be a problem for users unfamiliar with the concepts of observability and the library is quite limited. The price can also be a problem for smaller teams. (10)

**Pricing** - Honeycomb offers a free service, while its pro tier starts at 100$. The pricing is based on data retention and the volume of events captured. (10)

**Auvik** - Auvik is a network management software that automates the discovery, mapping, inventory, and documentation of networks. It allows users to see the network topology and fine details about devices, including their data lifecycle, in real-time. It also alerts users of issues and helps with troubleshooting by providing insightful information. Last but not least, SaaS management allows users to track SaaS usage, spending, and security across the network.

**Pros and Cons** - While Auvik is a very flexible and powerful tool that is great for network troubleshooting and mapping, it can be quite expensive and requires a lot of attention. This makes it best suited for larger organizations. (11)

**Pricing** - Auvik doesn't list pricing on its website. Interested users can request a custom quote. (11)

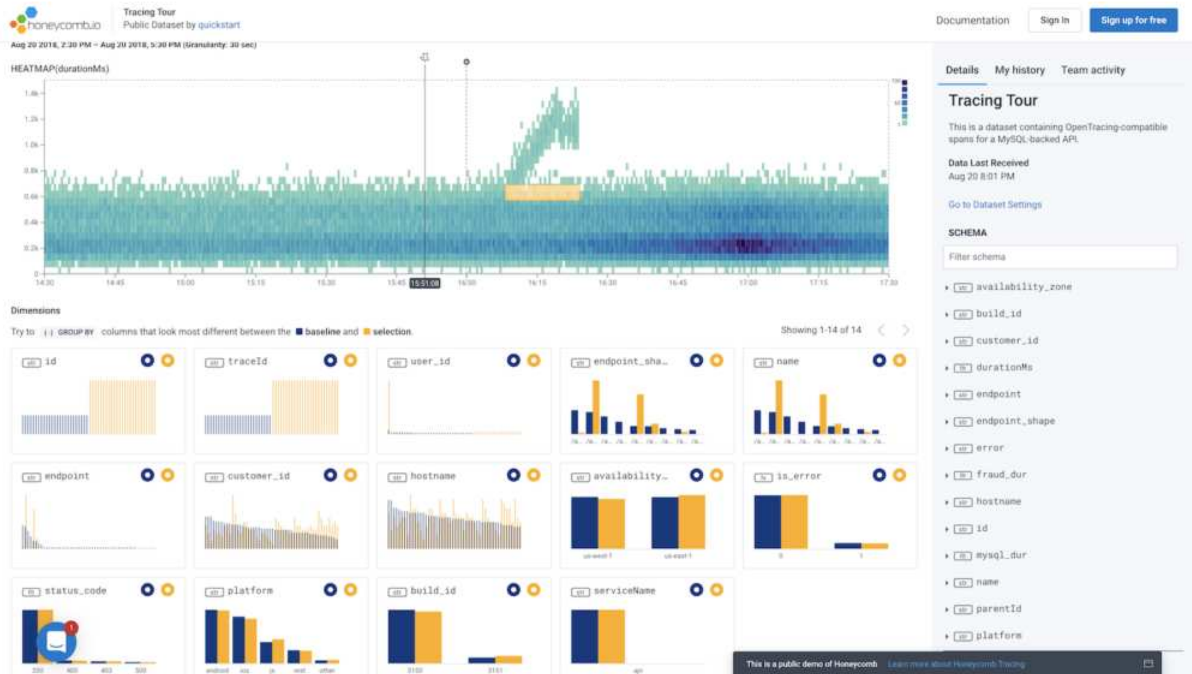



Figure 3.2. HoneyComb Dashboard.

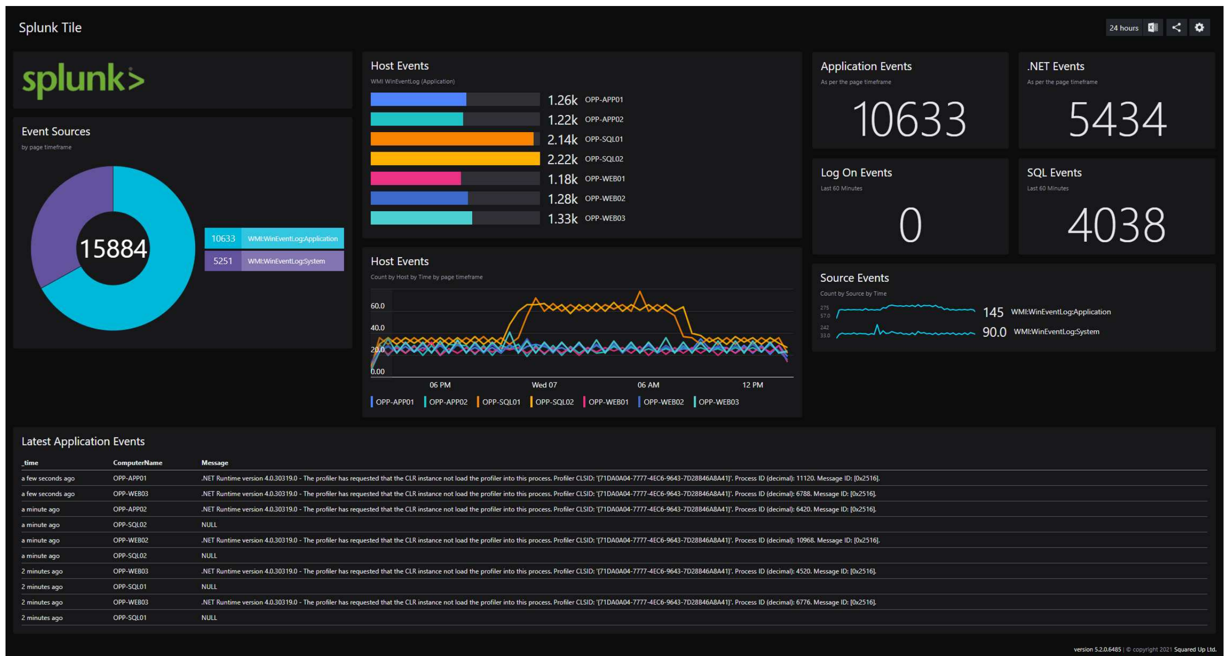



Figure 3.3. Splunk Dashboard.

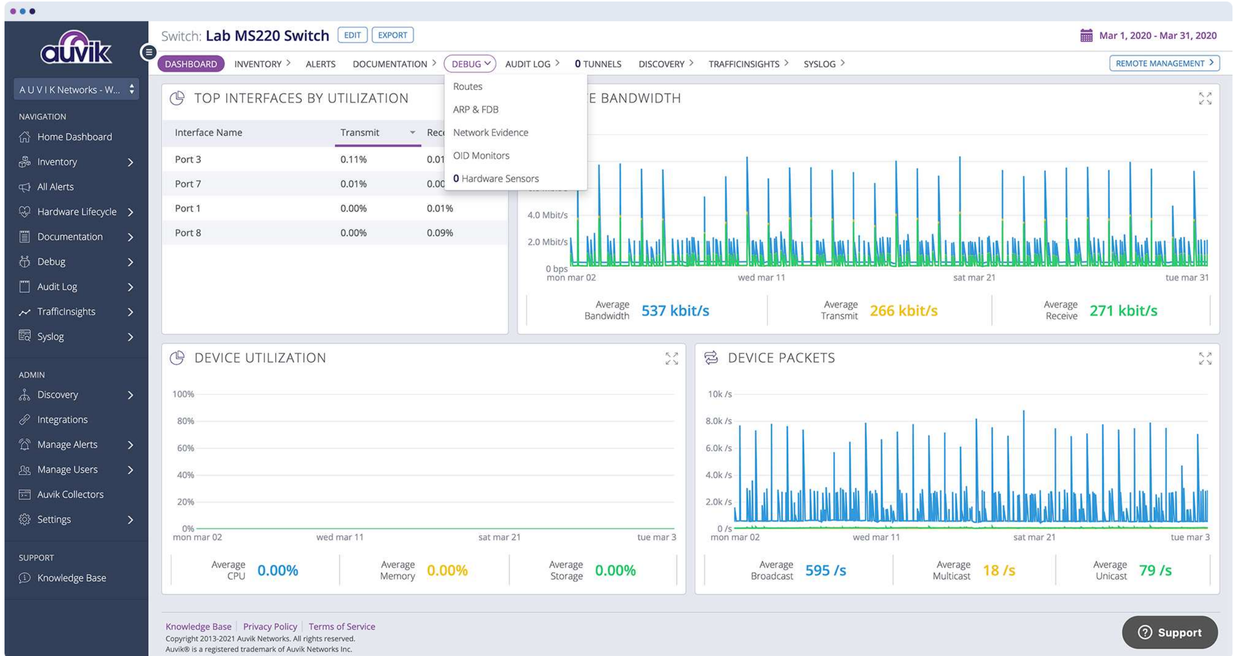



Figure 3.4. Auvik Dashboard.

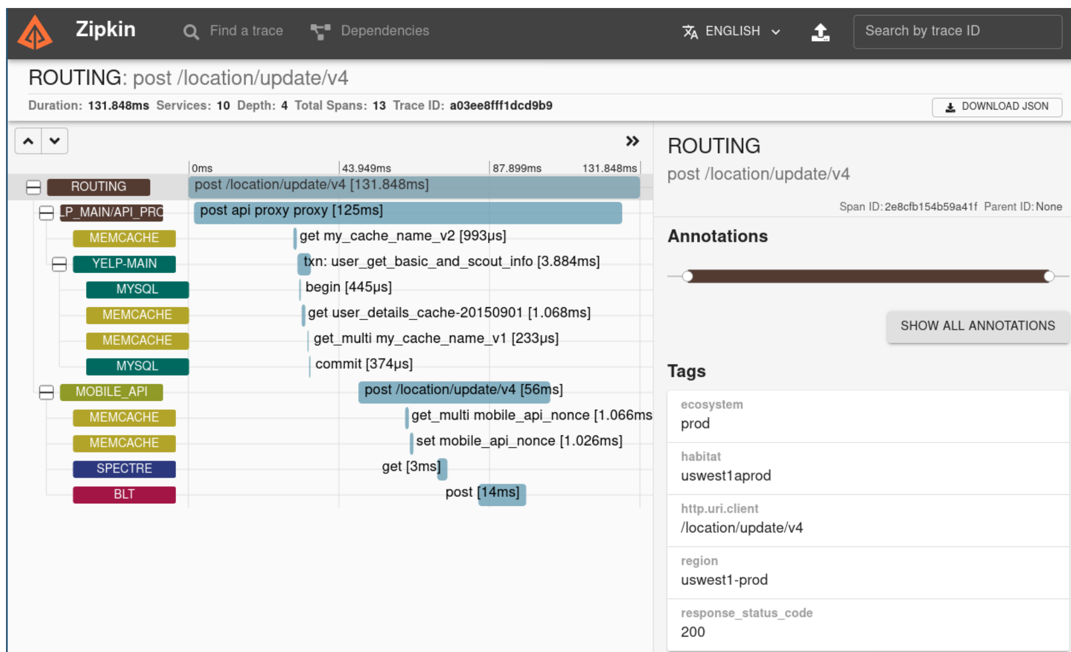



Figure 3.5. Zipkin Dashboard.

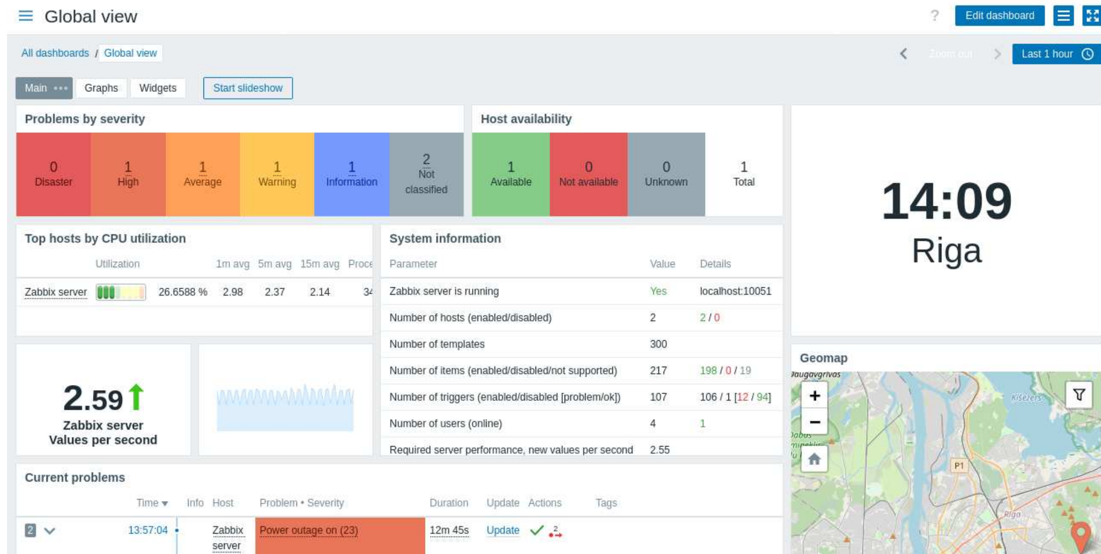



Figure 3.6. Zabbix Dashboard.

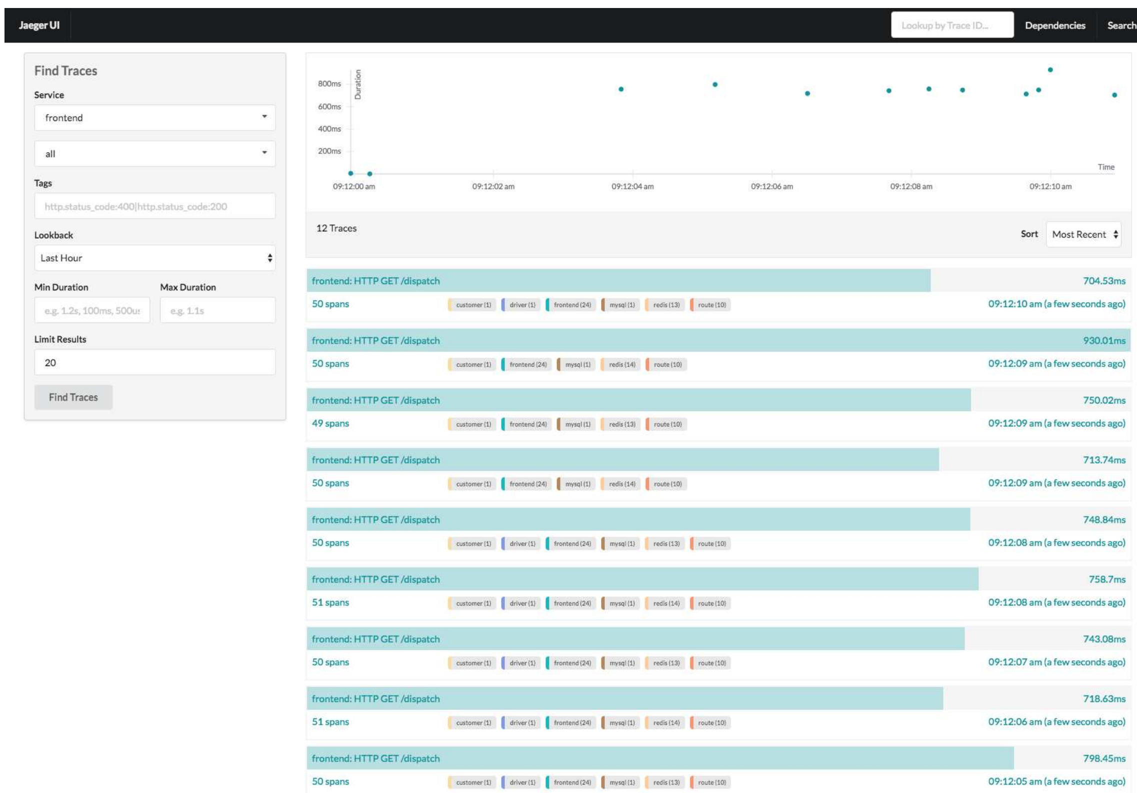



Figure 3.7. Jaeger Dashboard.

**Splunk** - Splunk is a software for searching, monitoring, and analyzing machine-generated big data via a web-based interface. It captures, indexes, and correlates real-time data in a searchable repository from which it can create charts, reports, alerts, dashboards, and visualizations. It was included in this section because it is very popular with users and appears in pretty much every article on the topic. It even was the Top Rated on TrustRadius in 2020. (12)

**Pros and Cons** - Splunk is very good at collecting and aggregating log messages from different sources so that users can easily access and analyze log data in a centralized location. Splunk's reporting features are very powerful and easy to use, powered by intuitive, well-made dashboards. Other than that, the GUI can be quite confusing, the integration with Excel is limited and its architecture is so complex that it can discourage new users.

**Pricing** - Splunk's observability Cloud for Enterprise editions starts at 95$ per host per month if billed annually.

## 3.4 Free Open-Source Alternatives

**Zipkin** - Zipkin is an open-source APM tool used for distributed tracing. Zipkin collects timing data needed for debugging latency issues in service architectures. Indeed, in distributed systems, it is quite difficult to trace user requests across different services. If a request fails or takes too long, distributed tracing helps to understand what caused it. Zipkin includes Reporters that send data, which is then collected by Collectors. The data can then be easily queried via APIs and the included user interface.

**Pros and Cons** - Zipkin is a mature platform with broad industry support and a large and active community. Because of that, it has a wide range of extensibility options and tool integrations. However, since it uses a less modular and more centralized architecture, it's slower and less flexible than its newer competitors, especially at scale. Also, its UI is limited, but you can use add-ons for better analytics and visualizations.

**Jaeger** - Jaeger can be seen as a modern version of Zipkin. In addition to Zipkin's feature set, Jaeger also offers dynamic sampling, a REST API, a ReactJS-based user interface, and support for Cassandra and Elasticsearch

in-memory data stores. To achieve that, Jaeger takes a decentralized approach, with a client that sends traces to an agent that waits for incoming spans and forwards them to the collector. The collector then validates, transforms, and persists the spans. Not all generated traces and spans are collected, as Jaeger takes a dynamic representative sample of the monitored data.

**Pros and Cons** - Jaeger's documentation is very well done and offers a wide range of possible applications. Jaeger's original way of collecting data handles sudden traffic spikes better than other tools, which increases Jaeger's overall performance even on a large scale. Like the main advantages, the main disadvantages have to do with its modern approach and relative immaturity: Jaeger is written in Go, for example, which is far less popular than Java and Python, so you may need to learn a new language. Also, its architecture is much more complex and difficult to maintain.

**Zabbix** - Zabbix is an enterprise-grade open-source monitoring solution known for its scalability and extensibility. It offers features for network, server, and application monitoring such as agentless and agent-based monitoring, real-time alerting and notifications, performance and availability reporting, and support for SNMP and IPMI.

**Pros and Cons** - Zabbix is scalable for large environments, very customizable, and flexible, as it supports various data sources. It also has a very active community, and its documentation is well-written and complete. It can be quite complex for some setups, though.

## 3.5 Integration For Automation: NetBox And Ansible

As mentioned in the summary, modern efforts to make networks more efficient focus not only on observability but also on automation. We will now briefly explain how you can achieve this by integrating NetBox with another tool: Ansible.

Ansible is a widely used open-source, command-line automation tool. It can be used to configure systems, deploy software, and orchestrate advanced workflows for application deployment, updates, etc. Ansible is very **secure and reliable**. This is achieved by using vanilla SSH to communicate with the hosts without the need for bootstrapping. Ansible uses human-readable

language, making it easy to use right from the start.

Let's now talk about NetBox - we will add details in the next chapter - and how it can be integrated with Ansible to achieve network automation. (13)

**A Network Source of Truth** - The Source of Truth (SoT) is the place where you can retrieve the intended state of the device. You should have a single Source of Truth per data domain, often referred to as the System of Record (SoR). You can aggregate the data from the physical site Source of Truth into other data sources for automation.

NetBox is an excellent choice because of its main advantages, which made it the de-facto standard for SoT in the industry. We will discuss them in the next chapter.

To create a framework for network automation you must first identify the Source of Truth for the data to be used in future automation. Every time you need a configuration data point for automation, it is **inefficient** to read the configuration from the device and it should be given that it is as it should be and not simply left there by mistake. When it comes to providing data to teams outside of the network organization, providing an API can help speed up data collection without having to log into the device first.

The NetBox Collection allows you to quickly add information to a NetBox instance. You just need to supply an API key and a URL to get started. With this Collection, a base inventory, and a NetBox environment you can quickly populate your SoT. **My solution** will allow users to quickly populate NetBox from SuzieQ while allowing SuzieQ to use NetBox as its SoT in the process.

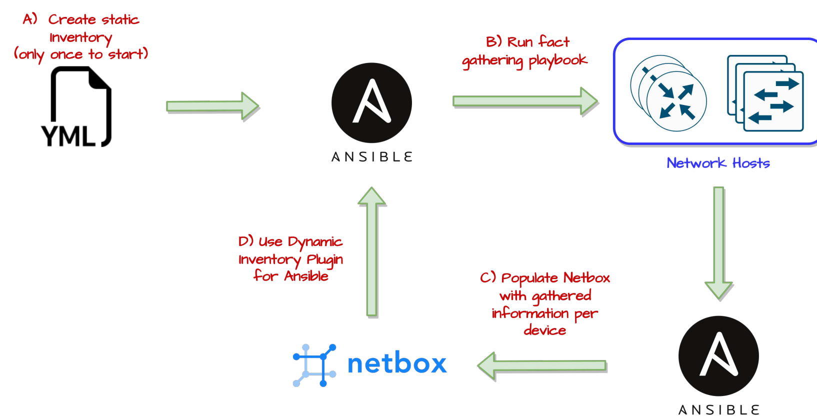



Figure 3.8. Diagram of interactions between Ansible and NetBox. (14)

# Chapter 4

# NetBox

NetBox is the market-leading solution for modeling and documenting modern networks. It combines IP address management (IPAM) and data center infrastructure management (DCIM) with powerful APIs and add-ons, making it the ideal SoT for network automation. Unlike other CMDBs, NetBox has a data model that is specifically tailored to the needs of network administrators. For example, you can document racks, hierarchical regions, cables, AS numbers, and much more.

## 4.1 Functional Scope

NetBox was developed for network operators, and this is reflected in its main functions. Among other things, it offers:

- IPAM with full IPv4/IPv6 parity;
- Automatic IP provisioning;
- VLANs with variably-scoped groups; ASN management;
- Device modeling;
- VPN tunnels;
- User-defined fields to extend the data model;
- Much more.

The NetBox's feature set is **limited** to ensure that development focuses on what is needed and that the exposure surface is minimized. To overcome

this, it can be integrated with other tools. On its own, it cannot be used for **network monitoring** - which is why SuzieQ is so well suited for integration - DNS or RADIUS server, configuration, and facility management.

## 4.2 Design

NetBox's design is focused on providing a data model that can reflect a real-world network. For instance, IP addresses are assigned to interfaces, which will be then attached to a device. This kind of choice makes it very precise but also kind of convoluted.

NetBox represents the **desired state** of a network versus its operational state. It can then be used to fill monitoring and provisioning systems with a high level of confidence. However, NetBox itself does not perform any scanning of network resources, but SuzieQ takes care of this task in our solution.

The code itself is kept **as simple as possible**, striving for a reasonable compromise between complexity, optimization, and provided functionalities.

## 4.3 Data Model

Each object type in NetBox, such as a device or IP address, is represented by a model. Each model is defined as a Python class and has its own SQL table. Tables in NetBox are implemented as discrete SQL tables, each representing a model. Each attribute of a model exists as a column within its table.

NetBox supports six types of custom fields:

1. Text - Free-form text (up to 255 characters);
2. Integer - A whole number (positive or negative);
3. Boolean - True or False;
4. Date - A date in ISO 8601 format (YYYY-MM-DD);
5. URL - This will be presented as a link in the web UI;
6. Selection - A selection of one of several pre-defined custom choices;

7. Multiple selection - A selection field that supports the assignment of multiple values.

NetBox also supports **object nesting**, as shown in Figure 4.1. This is achieved through the use of foreign keys, which allow a field in one table to reference an entire object from another table. This is a common practice in relational databases and is used extensively in NetBox to create relationships between different types of objects.

For example, a device object in NetBox might have a site field that references a site object. This allows the device to be associated with all the information contained in the site object, such as its name, location, and other attributes. Moreover, NetBox introduced a new type of custom field that enables referencing a related NetBox object.

While this allows for a high degree of flexibility in modeling complex relationships, it also requires **careful management** to ensure data integrity and avoid confusion.

All in all, NetBox is an excellent open-source tool, with a very high level of granularity, APIs that are comprehensive and easy to use, a well-polished GUI, and a high degree of customization and extensibility.

```
{
    "count": 54,
    "next": "https://demo.netbox.dev/api/dcim/devices/?limit=50&offset=50",
    "previous": null,
    "results": [
        {
            "id": 138,
            "url": "https://demo.netbox.dev/api/dcim/devices/138/",
            "display": "ANSIBLE-SWITCH01",
            "name": "ANSIBLE-SWITCH01",
            "device_type": {
                "id": 7,
                "url": "https://demo.netbox.dev/api/dcim/device-types/7/",
                "display": "C9200-48P",
                "manufacturer": {
                    "id": 3,
                    "url": "https://demo.netbox.dev/api/dcim/manufacturers/3/",
                    "display": "Cisco",
                    "name": "Cisco",
                    "slug": "cisco"
                },
                "model": "C9200-48P",
                "slug": "c9200-48p"
            },
            "role": {
                "id": 4,
                "url": "https://demo.netbox.dev/api/dcim/device-roles/4/",
                "display": "Access Switch",
                "name": "Access Switch",
                "slug": "access-switch"
            },
            "device_role": {
                "id": 4,
                "url": "https://demo.netbox.dev/api/dcim/device-roles/4/",
                "display": "Access Switch",
```


Figure 4.1. A NetBox device with nested objects.

# Chapter 5

# SuzieQ

SuzieQ is an open-source, multi-vendor network observability application that focuses on improving your understanding of your network. It allows you to analyze your network using a range of **vendor-independent queries and methods**. For example, with SuzieQ you can ask questions such as: What was the state of an interface 30 minutes ago? What changes have been made to a VLAN between now and 10 am yesterday? What software version is running on our devices? Is LLDP running correctly on the network? SuzieQ is primarily intended for network engineers.

## 5.1 Main Components And Functions

SuzieQ follows this process:

- Data is collected from routers, bridges, and Linux servers with an agentless model using either SSH or REST API as the transport. Various providers are supported, such as Arista EOS, Cumulus Linux, Cisco's IOS, IOS-XE, and IOS-XR platforms, etc.

  The **Poller** is the component taking care of this. Periodically, it collects data from the list of nodes specified in the inventory file. Then, those lists fill a centralized global inventory.

  This **Inventory** can be split and distributed to a customized number of workers. Some sources are dynamic - Netbox - so the node list might change over time. The Poller can dynamically track these changes and provide updated inventory chunks to the workers.

- The data is normalized into a vendor-agnostic format. This allows consistent analysis and the seamless integration of data from different sources. The **Data Normalizer** takes care of this.

- All data is stored in files using Parquet, a popular Big Data format.

  The **Coalescer** is part of the SuzieQ data processing pipeline. After the data is collected and normalized, it is timestamped and stored in the **Database**. At this point, the Coalescer reorganizes this data - belonging to a certain time window - to ensure time-efficient further analysis. The Coalescer accepts fine-grained periods such as 10m, 2h, etc.

- The data is then made accessible via a command line interface (CLI), a graphical user interface (GUI), a REST API, or Python. It can be analyzed using simple, intuitive commands. Data can be visualized and exported in various formats, from plain text to JSON, CSV, and Markdown.

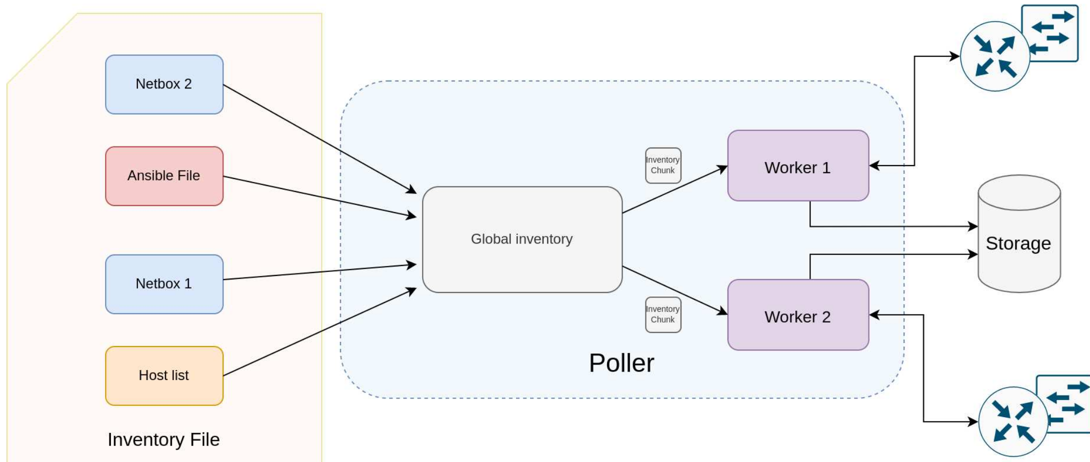



Figure 5.1. Poller Architecture.

## 5.2 Data Model

SuzieQ gathers data and stores it in tables, the most basic structures of the SuzieQ database. Each table corresponds to a specific service that collects

data. For instance, the BGP table contains the BGP data that the BGP service collects from routers.

To check what kind of information is collected for each table, you can use the `<table> describe` command through the SuzieQ CLI. For example, to see the details of the fields in the BGP table, you can run the command `bgp describe`. Fields can vary depending on the type of collected data.

```
suzieq> device show namespace=nxos
   namespace    hostname                     model      version   vendor                          architecture status       address
bootupTimestamp
0        nxos     dcedge01                vqfx-10000    19.4R1.10  Juniper                                      alive  10.255.2.250 2021-04-21
06:52:09.329000-07:00
1        nxos       exit01  Nexus9000 C9300v Chassis       9.3(4)    Cisco  Intel Core Processor (Skylake, IBRS)  alive  10.255.2.253        2021-04-21
06:53:03-07:00
2        nxos       exit02  Nexus9000 C9300v Chassis       9.3(4)    Cisco  Intel Core Processor (Skylake, IBRS)  alive  10.255.2.254        2021-04-21
06:53:08-07:00
3        nxos   firewall01                        vm  18.04.3 LTS   Ubuntu                                x86-64  alive  10.255.2.249        2021-04-21
06:52:12-07:00
4        nxos       leaf01  Nexus9000 C9300v Chassis       9.3(4)    Cisco  Intel Core Processor (Skylake, IBRS)  alive  10.255.2.189        2021-04-21
15:22:51-07:00
5        nxos       leaf02  Nexus9000 C9300v Chassis       9.3(4)    Cisco  Intel Core Processor (Skylake, IBRS)  alive  10.255.2.188        2021-04-21
15:23:03-07:00
6        nxos       leaf03  Nexus9000 C9300v Chassis       9.3(4)    Cisco  Intel Core Processor (Skylake, IBRS)  alive  10.255.2.190        2021-04-21
15:23:16-07:00
7        nxos       leaf04  Nexus9000 C9300v Chassis       9.3(4)    Cisco  Intel Core Processor (Skylake, IBRS)  alive  10.255.2.191        2021-04-21
15:23:33-07:00
8        nxos    server101                        vm  18.04.3 LTS   Ubuntu                                x86-64  alive  10.255.2.204        2021-04-23
05:53:01-07:00
9        nxos    server102                        vm  18.04.3 LTS   Ubuntu                                x86-64  alive   10.255.2.39        2021-04-23
05:53:01-07:00
10       nxos    server301                        vm  18.04.3 LTS   Ubuntu                                x86-64  alive  10.255.2.140        2021-04-23
05:53:01-07:00
11       nxos    server302                        vm  18.04.3 LTS   Ubuntu                                x86-64  alive  10.255.2.114        2021-04-23
05:53:01-07:00
12       nxos      spine01  Nexus9000 C9300v Chassis       9.3(4)    Cisco  Intel Core Processor (Skylake, IBRS)  alive  10.255.2.119        2021-04-21
06:53:10-07:00
13       nxos      spine02  Nexus9000 C9300v Chassis       9.3(4)    Cisco  Intel Core Processor (Skylake, IBRS)  alive  10.255.2.120        2021-04-21
06:53:14-07:00
```

Figure 5.2. Poller Architecture.

# Part III

# NetBox And SuzieQ Integration

# Chapter 6

# Preliminary Studies

## 6.1 Reasoning

After analyzing our tools, their functions, and their main components, we can now remember the thesis goal: integrating them. But the question is, why would anyone want to do that?

The **market interest** for this function is high, as Stardust Systems has received several requests from customers. Why is this the case? Surprisingly, it has as much to do with the limitations and potential for improvement of the two tools as it does with their benefits.

**NetBox perspective** - NetBox is widely used because, as a Network Source of Truth (NSoT), it provides a consistent and reliable source of data while decoupling our intent from the actual infrastructure in a flexible, granular, intuitive way. Nonetheless, after analyzing user feedback on forums and the NetBox GitHub page, it's clear that there's still room for improvement.

- NetBox is very capable, but that comes with a steep learning curve (15). SuzieQ clients could be encouraged to use NetBox if they could use some of its benefits without having to deal with its complexity;
- NetBox is commonly used to manage VLANs and sites so we will include them in our solution;
- Filling NetBox manually can be tiresome and error-prone. Also, it requires a lot of passages in some cases e.g. setting an IP address as the main address on a device. SuzieQ could be used to bulk push data in a quick, convenient way;

- According to the 2022 Community Survey, users are asking for "topology diagram generation" and that's something SuzieQ can provide;
- Users may update the network state time after time, leaving NetBox behind. Periodic automatic updates could be a welcome change. User attention should be requested anyway to check if everything makes sense;
- LLDP is not supported by default on NetBox, but it's supported by SuzieQ. Cables can be dinamically pushed from SuzieQ to NetBox exploiting this.

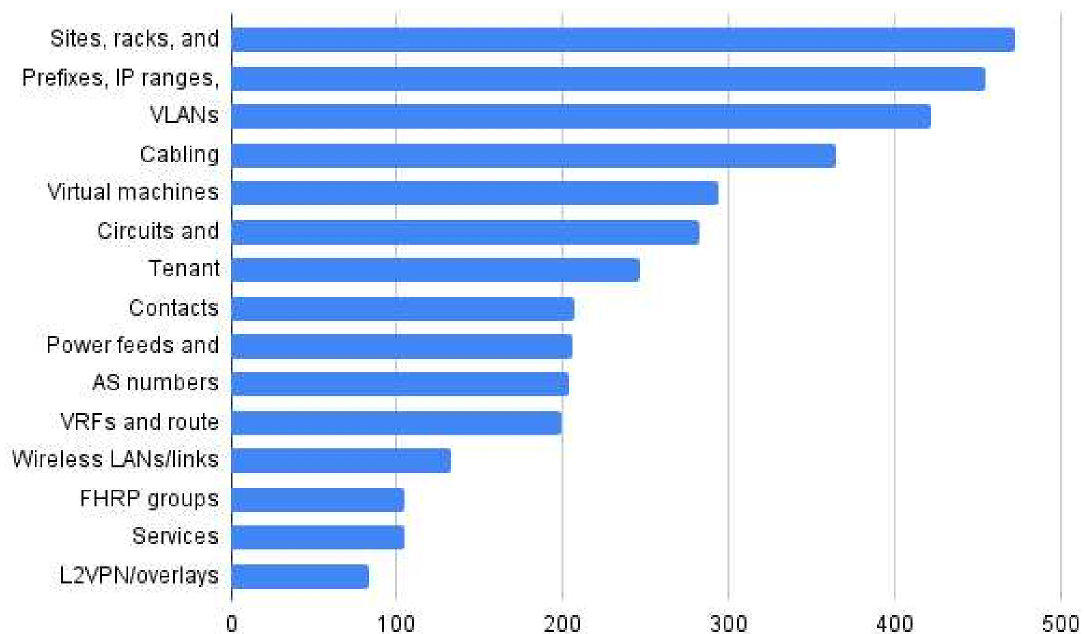



Figure 6.1. Most used objects in NetBox.

**SuzieQ Perspective** - SuzieQ aims to offer a wide variety of tools to handle your network and make it work consistently to support your business applications. Using NetBox functionalities seamlessly can enhance its capabilities: let's see how.

- **Observability** - If there's a mismatch between the intended and actual state, we can understand why based on data. Where is the problem? What is causing it? We can now answer all those questions.

- **Granularity** - NetBox documents a wide range of aspects about your network: the chosen devices (device, device type), their duties (device role), where they are (rack, site, region), how they're connected (interfaces status), and much more. Some information could integrate and enrich what's present in SuzieQ.

- **Feature completition** - SuzieQ can only pull devices from NetBox. We want to close the circle by writing a sturdy code foundation to allow users to push devices and compare the NetBox view (intended state) with the SuzieQ view (actual state).

**Business Perspective** - At last, let's see why Stardust Systems itself is interested in adding this feature to its suite:

- **Growth** - Covering more use cases, they could interest new users in adopting our system;

- **Low costs** - Other than some added complexity, there are no real additive costs here. Indeed, the new function will run independently without impacting the overall computational load;

- **User retention** - Users will be encouraged to keep using SuzieQ as new possibilities continue emerging, thanks to the team's effort to bring in more functionalities.

## 6.2 Requirements

After utilizing the tools, I presented my coworkers with a list of requirements to visualize the user experience and solution. Together, we choose the more relevant ones.

**Functional** - What should be possible:

1. See if the actual state matches the desired state (validation);
2. Update NetBox to match the current state (synchronization);
3. Enable periodical execution;
4. Enable execution on demand;
5. Immediately signal errors;

6. The status should always be visible to the user;
7. Store the operation's results in a log file;
8. Select the items you want to synchronize;
9. Updating (even deleting) selected fields;
10. Tracking device added from SuzieQ via a tag.

**Non-Functional** - How should the function behave:

1. Usability and learning curve - Seamless integration with SuzieQ: existing commands should be used as much as possible, and new commands should be intuitive;
2. Usability and learning curve - The system output should be as human-friendly as possible. For instance, "500: Unknown server error" is not okay;
3. Usability and learning curve - Logs should be rich enough in information, complete but not overdone. They should include timestamps, operation results, the reasoning behind errors and decisions (see FR 5, 6);
4. Scalability and performance - It shouldn't slow down other operations;
5. Scalability and performance - The operation should be completed within some minutes;
6. Security - The synchronization operation can be automatic or on demand: the user should have the possibility to choose to avoid unwanted, unreversible changes;
7. Security - The system should be as secure as it was before the function was introduced;
8. Accountability - The user should know what pertains to SuzieQ when looking at their data in NetBox. We will tag data accordingly;
9. Reliability - The operation should be always available;
10. Reliability - If the operation fails, it should be possible to retry it after.

# Chapter 7

# Architecture

## 7.1 Functions and Scope

After studying and using the two tools and their data models, two tables to start with are selected: **devices** to close the loop and **VLANs** to start supporting IPAM management. Further reasons for this are their relevance and the fairly simple translation between SuzieQ and NetBox.

**My Approach** - First, you must understand which fields are compulsory in NetBox and then which SuzieQ fields are needed to match those. As you can see from Tables 7.1, 7.2, and 7.3, the mapping is not 1:1, but that will be discussed later.

The **functional scope** of the solution was divided hierarchically into tiers. This allowed me to include only the most important parts in the final version of the code and limit the impact of time constraints.

1. Tier 1) Basic Synchronization and Validation functions - Push and compare whole tables;
2. Tier 1) Basic user customization - Enable the user to choose the tables and the granularity of the operations;
3. Tier 1) Performance optimization - Execute the task in <5 minutes;
4. Tier 2) Basic information logging - Provide information on how many items succeeded in comparison and push.;
5. Tier 2) Complexity hiding - Hide the specific implementation of the solution as much as possible (we'll see more on that later);

6. Tier 2) Future-proof coding - Make the solution as open as possible for future addictions and improvements;
7. Tier 3) Updating - Enable the user to update items;
8. Tier 3) Allow horizontal and vertical filtering on tables on which the operations will be applied.
9. Tier 4) User interaction - Add user interaction through CLI and GUI.

**Prospects** - Following a similar process, potential advancements for the feature are envisioned. However, as you will see in the final chapter of this thesis, my understanding of the implementation challenges and domain of the feature improved significantly over time, resulting in more precise and concrete changes. Initially, it was believed that future versions could also include:

- **Intent networking** - A new approach that aims to replace manual configuration and issue response with a deeper level of intelligence and intended state. That would require some integration with Ansible;
- **A full change control system** - Change control is a well-organized method for managing any alterations to a product or system. It ensures that no superfluous changes are made, all modifications are properly documented, and resource usage doesn't disrupt services. Users asked for this in the 2023 NetBox Community Survey.

## 7.2 Methodology And Complexity

While the **device table** is more complex on the NetBox side - as it contains many external references - the **VLAN table** is the real star of the show, as it will require accessing the interface table to be bound to a device. Also, VLANs are not unique in SuzieQ and NetBox in the same way. That will be discussed later.

**Different data models** - In SuzieQ, all information relevant to a table is provided locally in that table as a string, boolean, integer, etc. In NetBox, instead, any field can be another object from another table. Moreover, relevant information could be hidden by the GUI or entirely located on another table referencing the original one.

**Compare** - When comparing NetBox and SuzieQ, you either:

| | | | | |
|---|---|---|---|---|
| **Interface** | device | name | type | |
| **Device** | device_type | role | site | status |
| **VLAN** | name | status | vid | site |
| **Site** | slug | name | status | |
| **Manufacturer** | slug | name | | |

Table 7.1. Needed fields for tables of interest.

| | | | | |
|---|---|---|---|---|
| **SQ Device** | namespace | model, vendor | - | status |
| **NB Device** | **Site** | **Device type** | **Device role** | status |
| **SQ Namespace** | name | lower(name) | status | |
| **NB Site** | name | slug | status | |
| **SQ Device** | vendor | - | model | lower(model) |
| **NB Dev type** | **Manufacturer** | height | model | slug |
| **SQ -** | - | - | - | |
| **NB Dev role** | name | slug | color | |
| **SQ Device** | vendor | lowercase vendor | | |
| **NB Manufacturer** | name | slug | | |

Table 7.2. Mapping between SuzieQ and NetBox - Device Table.

| | | | |
|---|---|---|---|
| **SQ VLAN** | vname | state | VLAN |
| **NB VLAN** | name | status | vid |

Table 7.3. Mapping between SuzieQ and NetBox - VLAN Table.

- Compare two plain values;
- Convert values from a fixed set using dictionaries and then compare them. For example, a device can be "active" in SuzieQ but "alive" in NetBox, which is the same;
- Flatten NetBox nested objects to a plain value. For example, flatten a NetBox site to its name to match namespace information in the SuzieQ device table;
- Compare a composite value in NetBox with different plain values in SuzieQ. For example, a NetBox device may be built as SQ model + SQ vendor.

**Push** - When pushing SuzieQ objects to NetBox, you have to:

- Know in advance if some fields are references to external objects;
- In that case, do a lookup in NetBox data, fetch the object's ID, and use it instead of its name or other information;
- If that object is not in NetBox, it must be pushed manually. However, it's possible that you may not have all the required information to do so. For instance, if you want to push an interface and a device is not present, you need details such as its model and vendor. Unfortunately, these details are not available in the interface table;
- After all required external objects are in NetBox and their IDs are fetched, the "main" object can also be pushed.

**Limited by design** - After conducting preliminary studies, I decided to prioritize quality over quantity by implementing limitations on less polished functions. This does not mean that future versions cannot relax those limitations.

**The "interface situation"** - Since the software solution was designed around the "device" and "VLAN" tables, the "interface" table is only used to bind VLANs with devices/interfaces. Therefore, we cannot independently push interfaces just yet. This will be discussed further in the last chapter.

**No updates, only additions** - To avoid accidental data corruption or loss, any addition must be explicitly allowed. If you need to edit existing data,

you should evaluate each case carefully and make changes only if necessary. Updating is more disruptive than pushing, so we won't provide any automatic update mechanism for now. Instead, the user will be able to compare SuzieQ and NetBox using the comparison function, see the differences between them, optionally add mismatching objects, or fill up an empty instance of NetBox.

**VLANs 1:1 mapping** - In NetBox, VLANs are unique by name and site. In SuzieQ, there's a record for each name/hostname/namespace combination. Hence, to ensure that each SuzieQ VLAN has a corresponding NetBox VLAN to be compared to when pushing VLANs to NetBox, some wrapper information with the VLAN name, site, and device must be included.

## 7.3 Logical Modules

The implemented operations can be classified into different logical modules. However, this is just a **high-level classification** to isolate different operations and study them separately. There is no guarantee that all these operations will be separated coding modules. Each operation could be a method, an entire class, or a small part of another method. Some modules may already exist and need enhancements or adaptations, while others may be completely new.

Let's list them:

- **Storage** - Where data is stored. It can be a remote repository, offline folder, etc.;
- **Ingress Converter** - A module to convert external (NetBox) data so that it can be compared with internal (SuzieQ) data;
- **Egress Converter** - A module to convert internal (SuzieQ) data so it can be pushed to external tools (NetBox);
- **Configuration File** - A configuration source with custom user parameters;
- **Configuration Manager** - A module to enforce the user configuration on code;

- **Logger** - A module to extract and show the output in the most human-friendly way possible. It could even store some information in a storage medium;
- **Comparator** - A module to compare internal and external data. The output is then processed by the Logger;
- **Loader** - A module to take internal data from SuzieQ storage - after conversion and some intermediate processing - and push it to an external tool's storage. This may include some querying logic;
- **Mapper** - A module containing mapping information between the two tools e.g. dictionaries;
- **Coordinator** - A module to call smaller modules in the right sequence, ensuring all needed data is passed to them.

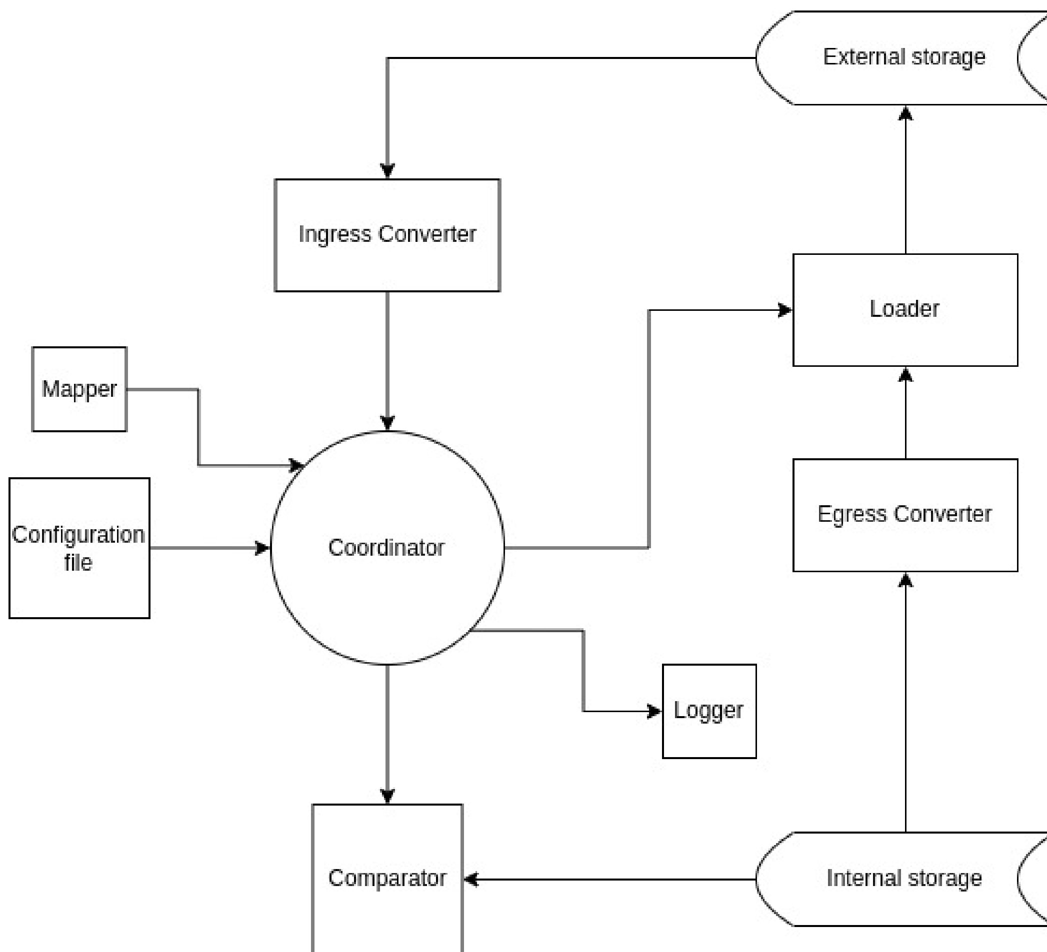



Figure 7.1. Diagram of interactions between the main modules.

# Chapter 8

# Implementation

## 8.1 Language and Libraries

Python, Pandas, and Pydantic were chosen because they're used in Stardust Systems. Here we will briefly explain what they are and what advantages and disadvantages they have.

**Python** - Python is a widely used high-level, general-purpose programming language. (16) (17) Some characteristics:

- Python uses indentation to designate code blocks, improving readability. Whitespaces (tabs or spaces) connect statements within the same block. To nest blocks, indent to the right;
- Python supports multiple programming paradigms, such as procedural, object-oriented, and functional. My solution makes heavy use of the latter;
- Python is dynamically typed and garbage-collected. This ensures a good grade of flexibility that must managed carefully. There are typing libraries for that;
- Its comprehensive standard library makes it the perfect programming language for writing simple programs without the aid of external libraries.

Let's understand what made Python the right choice for Stardust or anyone else in the market and which drawbacks must be considered.

**Pros** - Python's simple syntax is perfect for beginners. It's free to use and distribute and has a vibrant community behind it, leading to wide availability of libraries for any use case. For this reason, Python is suitable for applications such as web development, data science, and machine learning.

**Cons** - Python is an interpreted language, so it's a lot slower than compiled languages like C/C++ and Rust. Python is also heavy on memory, and some liberties it takes - for instance, dynamic typing - make it less safe and more difficult to debug than strictly typed languages.

Python's popularity is growing also in modern domains such as data analysis and machine learning. So, using it is convenient and future-proof.

**Pandas** - As we mentioned, Python is seeing rapid adoption for data science/data analysis and machine learning tasks. Pandas is an open-source Python package that is used for this. Since massive amounts of data will be processed, Pandas is perfect. (18)

**Pros** - The Pandas library allows developers to represent and process (wrangle) data in various ways with a comprehensive list of methods. It makes code very dense, fast, and efficient: a welcome improvement over "plain" Python iterative loops.

**Cons** - The learning curve is fairly steep, as it can take quite some time to understand which combination of methods you should use and why. The documentation doesn't mitigate this at all.

Pandas is a beloved library with a vibrant community behind it and a lot of users all over the world.
Pandas is an essential library with a vibrant, dynamic community. It just requires some effort to get accustomed to it and make better choices.

**Pydantic** - Pydantic is a Python library for data validation and setting management. With it, you can create nested models, validate data automatically - or in a custom way - against your models, ensure data integrity, and much more. (19)

**Pros** - Pydantic makes both writing code - using Python's type annotations - and debugging it - with clear error messages - very intuitive. It also ensures that your data is consistent and can be integrated with Modern Web Frameworks.

**Cons** - Pydantic is Python-specific, so it's not the best if you're working in

a multi-language environment. Pydantic can also be slow when validating datetime objects.

Pydantic is powerful and simple enough for ingress data and configuration file validation, and its drawbacks don't affect me.

## 8.2 Files And Directory

A directory was made just for the NetBox integration modules. There are three auxiliary files, called `netbox_constants.py`, `nb_cfg.yml` and `nb_utils.py`. The first contains the needed mapping data, while the second contains the configuration parameters for users to customize their experience. The third file is a Pydantic Validator for the second file.

Two other files, called `test_main.py` and `test_main.py` are used to test the code. While the first passes the right parameters to the second, the second is a wrapper calling the methods in the right order, one after another.

The two main modules are `netbox_integration.py` and `suzieq_facade.py`. The first contains the NetBox-specific classes and methods. The second contains all the classes and methods which are specific to SuzieQ. The methods are hidden under a facade to enable their seamless use by other tools in the future.

## 8.3 The Facade

The facade pattern is a software design pattern used in object-oriented programming. As mentioned, a facade serves as a front-facing interface masking the low-level complexity of the underlying code.

What problems can the Facade design pattern solve?

- It makes a complex subsystem easier to use;
- It decouples the high-level use of relevant functions from their low-level implementation.

Two versions of a hypothetical facade were designed.

1. **Naive approach** - The Facade will include all the modules inside the SuzieQ part, as long as it involves schema conversion and content comparison. The client specifies what operations to perform, which tables should be involved, how often, etc. via the configuration file.

   The Facade will do everything, from creating intermediate objects like NetBox tables to converting them to the SuzieQ format to compare their content. This approach has a limitation: the conversion can be wildly different from one case to the other. In some cases, it's just a matter of resolving some references, flattening the DataFrame, and there you are; in other cases, there's also a conversion to be performed on the content.

   Converting data may require several steps such as mapping values, changing units, and modifying the data types. We could fill the constants file with dictionaries and update them as time passes and more components need it, but it's unclear how scalable or feasible that would be.

   **Bottomline** - Conversion can vary so much that trying to do it within the same component, trying to fit every scenario, may not be feasible as it could lead to unoptimized, dirty code. Even if we conditioned what's being executed based on which kind of objects are being converted, we would end up with a monster class with redundant methods.

2. **Conservative approach** - Under the facade we compare different tables and validate ingress data to be compared with SuzieQ. Another module - in our case, the NetBox one - performs its specific conversion operations and then submits it to the facade for comparison. To be accepted, ingress data must match the expected schema.

   If something is wrong, the comparison won't happen. With this approach, the conversion methods can be used by different app-specific modules without the user noticing. The developer has to write the app-specific module and nothing else. There is a single entry point for the comparison operation.

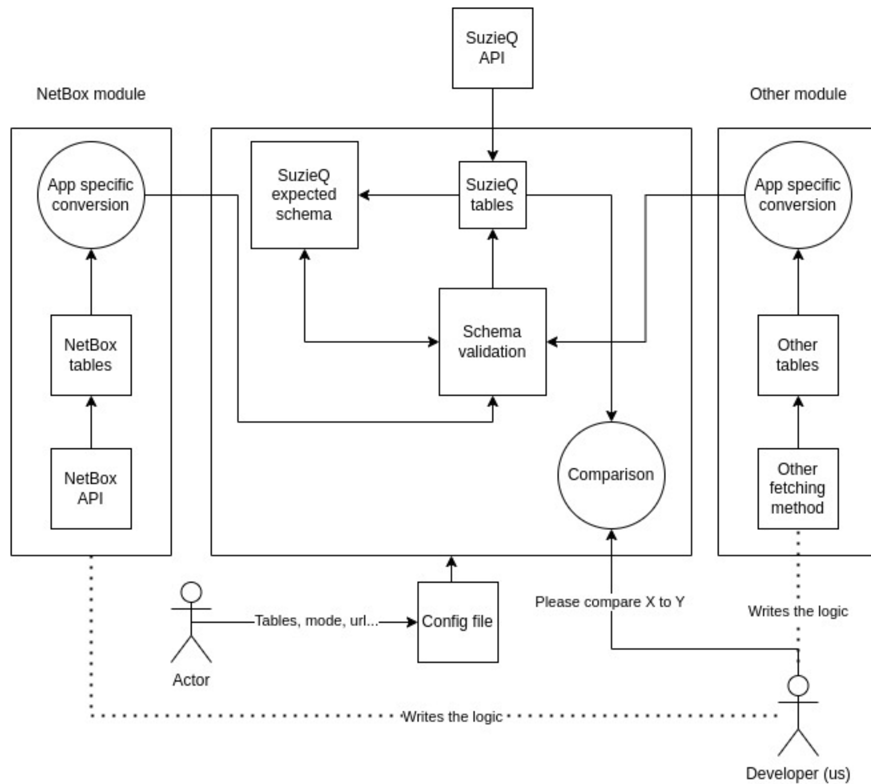



Figure 8.1. The winner: Conservative Facade schema, comparison function.

## 8.4 Main Classes

Initially, a basic version of the code was developed to figure out the low-level logic. This version lacked a clear structure and was essentially a long block of text. Later, it was restructured and separated into different files and code blocks. A combination of top-down and bottom-up approaches was chosen by designing the main classes and methods beforehand and then building the sub-classes as needed.

The main classes:

- **NetboxTable** - This is the main class of the `netbox_integration.py` module. It contains all basic methods e.g. `__init__(...)` to initialize a NetBox table, `get_data(...)` to perform GET requests to the NetBox API servers and `push_obj(...)` to perform POST requests.

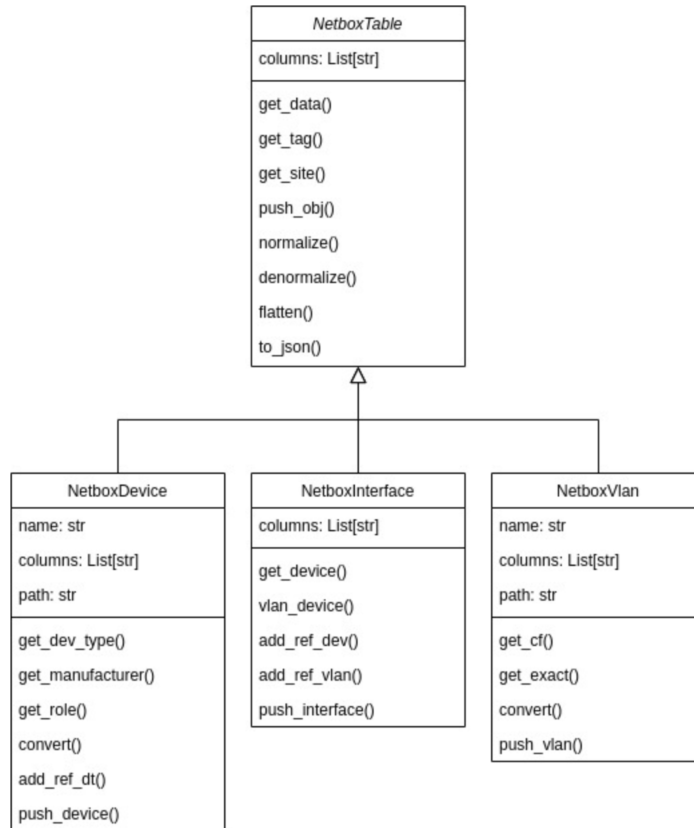



Figure 8.2. NetBox classes diagram.

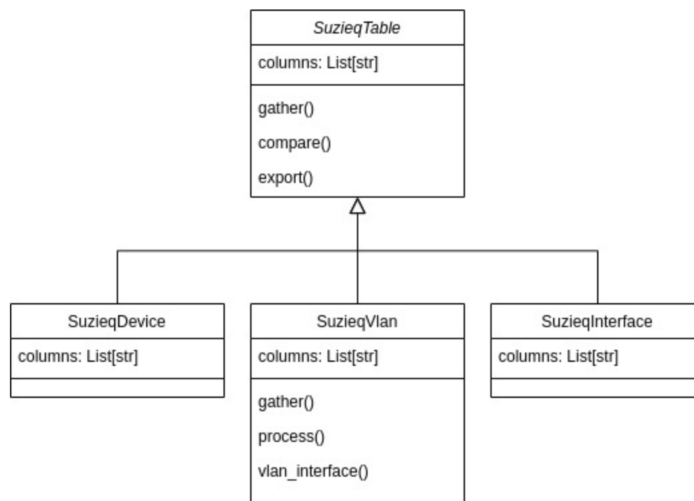



Figure 8.3. SuzieQ classes diagram.

Any subclass can use the common methods included in it. Those encompass two main types of operations: data wrangling e.g. `flatten(...)`, `normalize(...)`, `to_json(...)` and more specific versions of data GET e.g. `get_site(...)` and `get_tag(...)`.

**Subclasses** - This class has one subclass for each kind of table that we want to work with: `NetboxDevice`, `NetboxVLAN` and `NetboxInterface`. They contain methods divided into data-wrangling methods and GET/POST wrapper methods. They also include methods to convert ingress data to enter the Facade.

**Superclass** - This class is actually a subclass of `NetboxConn`, which makes connecting to NetBox possible.

- **SuzieqTable** - This is the main class of the `suzieq_facade.py` module. It contains the `__init__(...)` method, the `gather(...)` to fetch data from SuzieQ and the methods which are the fulcrum of this thesis: `compare(...)` to fully perform the comparison with ingress data and the `export(...)` method, to prepare data for exiting the Facade, in different formats (to make it future-proof). As already discussed, as it can be quite specific, the push function - for any external tool - will be handled by the application-specific component.

  **Subclasses** - This class has one subclass for each kind of table that we want to work with: `SuzieqDevice`, `SuzieqVLAN` and `SuzieqInterface`. Usually we only overwrite some parameters, while the `SuzieqVLAN` class includes an overwritten `gather(...)` method and two methods to do the binding with interfaces and devices: `process(...)` and `VLAN_interfaces(...)`.

This module includes methods meant only for internal use. The most important is probably `df_validator(...)` which ensures that ingress data is in the correct format and can be compared with SuzieQ data.

## 8.5 Translation Logic

As discussed in Chapter 7, it was necessary to devote some time to understand NetBox's and SuzieQ's data models and how the device and VLAN tables could be converted, compared, and pushed from SuzieQ to NetBox.

This was an **impervious process**, with multiple challenges: once the needed fields were selected, there was a chain of dependencies and different uniqueness constraints between SuzieQ and NetBox. Also, some objects are present in NetBox but not in SuzieQ - and vice versa - so I had to figure out how to convert those by taking information from different tables. Figure 8.4 and 8.5 try to summarize that.

## 8.6 Code Logic And Methods

In this section, the code logic and the used methods will be briefly explained step by step. **Actual code won't be shown** here or in any other section because of the non-disclosure agreement I signed with Stardust Systems.

1. It all starts in the `test_main` module, where the needed configuration files are passed to the `wrapper_main` module. This mimics a user passing the configuration file paths to the NetBox integration command using the CLI.

2. The `wrapper_main` coordinates all the next steps. It performs the same operations for each table, with some exceptions. First, it initializes the `NetboxTable` object and gets its data with a `NetboxTable.get_data` method.

3. If there's data in NetBox for that table, it converts it using the `NetboxTable.convert` method and a comparison using the `SuzieqTable.compare` method. The comparison also includes ingress data validation thanks to the `df_validator` method. The comparison outputs all failing objects on SuzieQ and NetBox parts, separated. If there's no data in NetBox, no comparison will take place.

4. If NetBox is empty, all data from the selected SuzieQ tables must be pushed. A `SuzieqTable` object is initialized and its data is fetched using the `SuzieqTable.gather` method. If we're handling the VLAN table, the `build_VLAN` method decides which fields are needed, as the code provides an option to "filter out" interfaces, as they slow up computation quite a bit.

5. Now, if the push function is allowed by the user in `nb_cfg.yml`, we first prepare data to exit the Facade, calling the `push` method, a wrapper for the `export` internal method and then use the `NetboxTable.push_tablename`

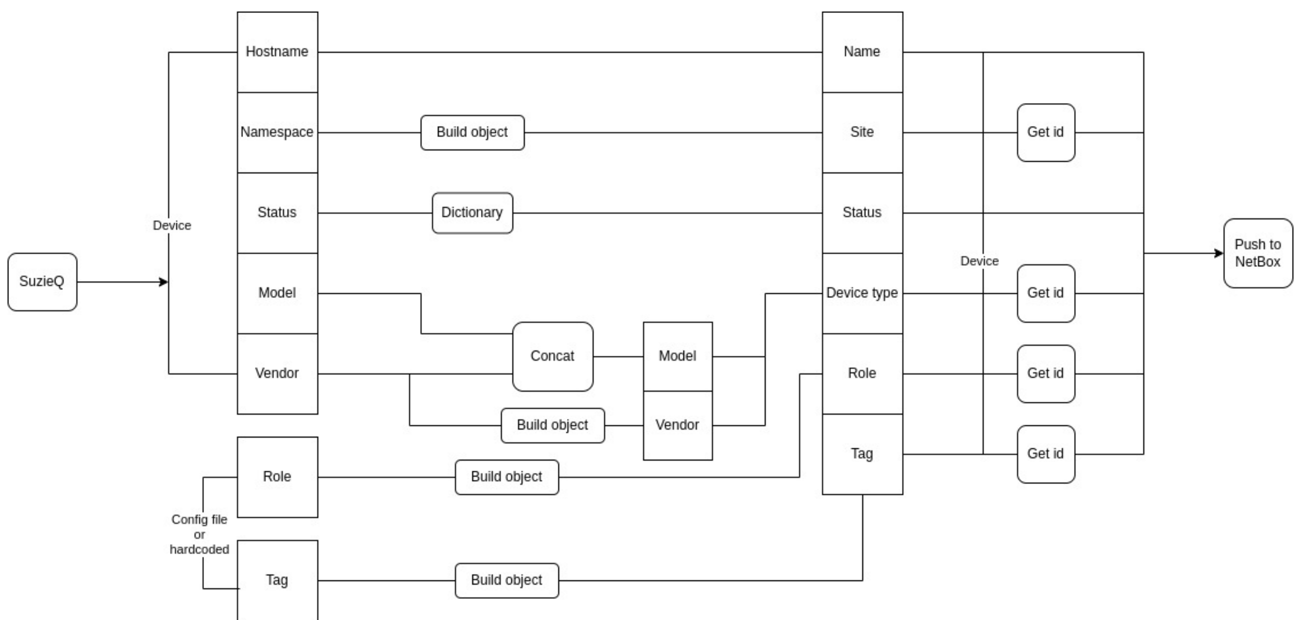



Figure 8.4. Device Push Logic. Device Comparison is similar.

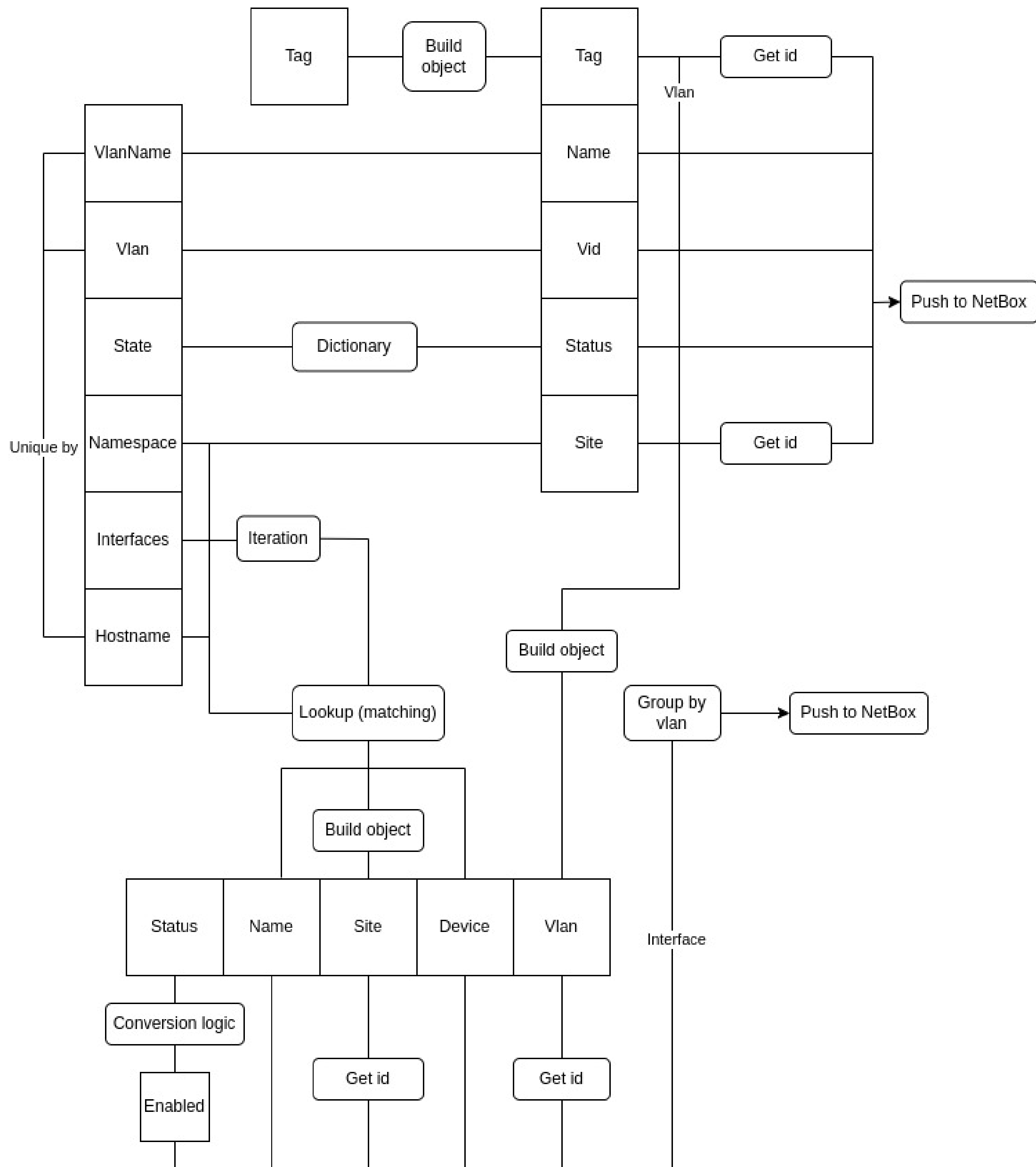



Figure 8.5. VLAN Push Logic. VLAN Comparison is similar.

method. To push VLANs with interfaces, we will retrieve only the interfaces listed in the VLAN table, using the `SuzieqVLAN.VLAN_interface` method.

6. The `NetboxTable.push_tablename` method works like this: first we understand which external objects are needed, selecting normalized NetBox data (all lowercase, no special characters) which doesn't match normalized SuzieQ data; then we push them; then we fetch their NetBox id which is unique within that table; then we push the "entire" object. The program will either push everything or nothing. Users can verify progress by running the comparison function.

## 8.7 Fine Tuning

In this section you will see all the improvements introduced along the way, from the first to the last version of my code.

**First Version** - In this version all necessary operations are performed in a brute-force fashion without regard for code structure or optimization. In this way, low-level logic can be fast understood. The code fetched all data from VLAN, device, and interface tables in SuzieQ, compared each row with NetBox data, and pushed a row at a time, regardless of which data was already in NetBox. The output was manually handled with dedicated error messages for each mismatch, making it richer but not scalable. Improvements:

1. **Code refactoring** - The code was reorganized into files, classes, and methods. This made the code much more readable and scalable. To add functions, you can add a subclass and some methods or add some dictionaries to the `netbox_constants.yml` file. You know where to add new code as everything follows a hierarchical logic, and you can look at pre-existing similar methods, which are placed and named coherently and intuitively.

2. **The Facade** - The SuzieQ part, enclosed in the `suzieq_facade` module, offers some generic methods that will always work regardless of the source of ingress data, given that it's converted in the right way. It can also export data in different formats, and you can add others following what's already in the `export` method. There also are some internal methods that could be useful for future functions.

3. **Asynchronous Operations** - The GET and POST requests are now performed asynchronously - using the `asyncio` library - to align the code with the pre-existing SuzieQ codebase. This also ensures that if the component is integrated into a part of the architecture that supports concurrent execution, we can speed up computation.

4. **Bulk Comparison** - Now we don't compare rows but `DataFrames`: we don't need to write the output manually anymore, as differences are shown regardless of which kind of data doesn't match. This is also a lot faster than iterating on rows.

5. **Bulk Requests** - We don't perform one request for one object: we first group objects and then perform requests. Examples: we don't perform a POST request for each object but for an array of objects; when fetching the unique ID of external objects to prepare devices, VLANs, or interfaces for the POST, we don't make a GET request for each external object but we fetch all objects of that kind and then filter out the one we need.

**Intermediate Version** - This version performs the needed operations on the selected tables. The code is decently organized, and testing was successful for many scenarios. The next improvements will focus on performance and better execution flow, avoiding useless operations and some extras to complete the solution.
Improvements:

1. **Data Tagging And Normalization** - SuzieQ data is labeled to distinguish it from existing data in NetBox, from data pushed by other tools or entered manually. Additionally, data is normalized before comparison to ensure that differences in case and special characters do not impact the primary functions.;

2. **VLAN Uniqueness** - The code does no longer push tags combining the VLAN name, site, and device - used so that NetBox VLANs can be 1:1 with SuzieQ VLANs - but it uses custom fields. That way, we don't clutter our NetBox instance;

3. **Device Type Handling** - The first version and the intermediate one pushed a Device Type for each device, while now it will only push one for each model/manufacturer combination. This speeds up computation and declutters our NetBox instance.

4. **Bulk Push And Optimizations** - When performing push operations, we bulk all the missing objects together and send them in a single POST request to the server. For GET operations, we apply filters to the entire NetBox tables, enabling us to perform only one request for those as well.

5. **Commented Code** - The code has been thoroughly commented to be easier to understand both for me in the future and for my coworkers, who will push this feature to market in future versions of the SuzieQ suite.

# Part IV

# Results

# Chapter 9

# Evaluation

The solution performs its main tasks: synchronization and validation. The program has undergone extensive testing and works seamlessly for devices and VLANs. It has been successfully tested several times, and it works perfectly fine in these scenarios:

- NetBox is empty and we want to push everything from SuzieQ. No comparison will be performed to speed up computation. We can push only devices, or devices and VLANs, or devices, VLANs and related interfaces;

- NetBox is not empty, so the program outputs the differences. The program only pushes mismatching SuzieQ objects when it's not too complicated;

- If we want to compare VLANs but don't care about interfaces, we can exclude interfaces to speed up the computation;

- Thanks to data normalization, similar SuzieQ and NetBox data are equal in validation and synchronization, regardless of the fields' case or use of special characters (underscores and hyphens). This is crucial when dealing with a NetBox instance that already has some objects in it because it's linked to other tools or filled manually, as the syntax conventions may vary;

- Thanks to data validation, the program won't try comparing data with mismatching formats: if the ingress data from NetBox isn't converted correctly, the program will signal it and stop its execution;

```
108        panos        server101                          vm  18.04.6 LTS    Ubuntu                               x86-64  alive
5.2.9 2021-12-13 09:34:51+01:00
109        panos        server102                          vm  18.04.6 LTS    Ubuntu                               x86-64  alive
.2.76 2021-12-13 09:34:51+01:00
110        panos        server301                          vm  18.04.6 LTS    Ubuntu                               x86-64  alive
2.241 2021-12-13 09:34:51+01:00
111        panos        server302                          vm  18.04.6 LTS    Ubuntu                               x86-64  alive
2.103 2021-12-13 09:34:51+01:00
112        panos          spine01                          VX        4.1.1   Cumulus                               x86_64  alive
2.117 2021-12-14 09:23:54+01:00
113        panos          spine02                          VX        4.1.1   Cumulus                               x86_64  alive
2.118 2021-12-14 09:23:54+01:00
114          vmx      CRP-ACC-SW01                        vmx     18.2R1.9   Juniper                                       alive
5.151 2021-08-16 11:17:22+02:00
115          vmx      CRP-DIS-SW01                        vmx     18.2R1.9   Juniper                                       alive
5.152 2021-08-16 11:17:32+02:00
116          vmx    TOR1BBN-PE-RT01                       vmx     18.2R1.9   Juniper                                       alive
5.155 2021-08-17 09:25:28+02:00
117          vmx   TOR1CRP-DGW-RT01                       vmx     18.2R1.9   Juniper                                       alive
5.153 2021-08-17 08:41:38+02:00
118          vmx   TOR4CRP-DGW-RT01                       vmx     18.2R1.9   Juniper                                       alive
5.154 2021-09-06 10:21:00+02:00
riccardo>
```

| Name | Status | Tenant | Site | Location | Rack | Role | Manufacturer | Type | IP Address |
|---|---|---|---|---|---|---|---|---|---|
| TOR1BBN-PE-RT01 | Active | — | vmx | — | — | Suzieq | Juniper | vmx | — |
| TOR1CRP-DGW-RT01 | Active | — | vmx | — | — | Suzieq | Juniper | vmx | — |
| TOR4CRP-DGW-RT01 | Active | — | vmx | — | — | Suzieq | Juniper | vmx | — |
| CRP-DIS-SW01 | Active | — | vmx | — | — | Suzieq | Juniper | vmx | — |
| CRP-ACC-SW01 | Active | — | vmx | — | — | Suzieq | Juniper | vmx | — |
| server101 | Active | — | panos | — | — | Suzieq | Ubuntu | vm | — |
| server302 | Active | — | panos | — | — | Suzieq | Ubuntu | vm | — |
| firewall01 | Active | — | panos | — | — | Suzieq | Palo Alto | PA-VM | — |

Figure 9.1. Devices in SuzieQ and NetBox, after synchronization.

## Devices

+ Add | Import | Export

Results 24 | Filters

Quick search | Configure Table

| Name | Status | Tenant | Site | Location | Rack | Role | Manufacturer | Type | IP Address |
|---|---|---|---|---|---|---|---|---|---|
| server101 | Active | — | panos | — | — | Suzieq | Ubuntu | vm | — |
| server301 | Active | — | panos | — | — | Suzieq | Ubuntu | vm | — |
| server102 | Active | — | panos | — | — | Suzieq | Ubuntu | vm | — |
| edge01 | Active | — | ospf-single | — | — | Suzieq | Ubuntu | vm | — |
| leaf01 | Active | — | ospf-single | — | — | Suzieq | Cumulus | VX | — |

```
These are the differencies:
     hostname    namespace status  vendor model
0   server102  ospf-single  alive  Ubuntu    vm
1   server103  ospf-single  alive  Ubuntu    vm
2   server101  ospf-single  alive  Ubuntu    vm
3   server104  ospf-single  alive  Ubuntu    vm
4      edge01  ospf-single  alive  Ubuntu    vm
..        ...          ...    ...     ...   ...
90   internet        junos  alive  Ubuntu    vm
91     exit02        junos  alive  Ubuntu    vm
92     edge01        junos  alive  Ubuntu    vm
93     leaf01        junos  alive  Ubuntu    vm
94     edge01        junos  alive  Ubuntu    vm

[95 rows x 5 columns]
```

## VLANs

+ Add | Import | Export

Results 28 | Filters

Quick search | Configure Table

| VID | Name | Site | Group | Prefixes | Tenant | Status | Role | Description |
|---|---|---|---|---|---|---|---|---|
| 10 | vl10 | vmx | — | — | — | Active | — | — |
| 100 | vl100 | vmx | — | — | — | Active | — | — |
| 100 | vl100 | vmx | — | — | — | Active | — | — |
| 1 | vlan1 | dual-bgp | — | — | — | Active | — | — |
| 1 | vlan1 | nxos | — | — | — | Active | — | — |
| 1 | vlan1 | ospf-ibgp | — | — | — | Active | — | — |

```
These are the differencies:
    vlanName  vlan   state    namespace
0     vlan10    24  active  ospf-single
1     vlan10    24  active  ospf-single
2     vlan10    24  active  ospf-single
3     vlan10    24  active  ospf-single
4     vlan13    13  active     dual-bgp
5     vlan13    13  active     dual-bgp
6     vlan13    13  active     dual-bgp
7     vlan13    13  active     dual-bgp
8     vlan13    13  active     dual-bgp
9     vlan13    13  active     dual-bgp
10    vlan13     1  active     dual-bgp
11    vlan13     1  active     dual-bgp
12    vlan13     1  active     dual-bgp
```

Figure 9.2. Validation output.

- Thanks to file validation, the user will be signaled if something they put in the configuration file is wrong: configuration data is in the wrong format, there was a typo, etc.;
- If a NetBox object matches a SuzieQ object except for some fields, the program will signal the difference but won't update. The reasoning behind this can be found in Chapter 6. With further studies, the update function may be introduced;
- Thanks to data tagging on NetBox, SuzieQ objects can easily be detected and deleted: this helps to troubleshoot and ensures Separation of Concerns so that if something goes wrong it doesn't affect non-related data.

## 9.1 Performance And Scalability

The solution performs its main tasks correctly. But how **fast** is it? The elapsed time for the different code sections was measured to understand what part is worth optimizing more. Testing only considered the worst-case scenario, starting with an empty instance of NetBox.

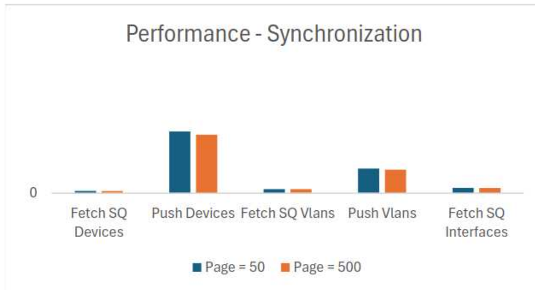
The elapsed time will be divided in:

- Time to fetch data from SuzieQ;
- Time to convert NetBox data;
- Time to compare SuzieQ and NetBox data;
- Time to push data in NetBox.

**An important note** - NetBox pages query results show only 50 objects per page by default. Larger values should reduce the number of iterations and improve the elapsed time for larger tables. As shown in the next section, this strategy was successful.

**Variables** - The results will showcase how paging can improve the performance while handling interfaces can be very detrimental because the VLAN binding with interfaces is complex. Significant data is expressed in seconds and commented.

| PUSH | Page = 50 | Page = 1000 | Absolute gain | Relative gain % |
|---|---|---|---|---|
| Fetch SQ Devices | 0,35 | 0,35 | 0 | 0 |
| Push Devices | **10** | **9,5** | 0,5 | **5** |
| Fetch SQ Vlans | 0,65 | 0,65 | 0 | 0 |
| Push Vlans | **4** | **3,8** | 0,2 | **5** |
| Fetch SQ Interfaces | 0,85 | 0,85 | 0 | 0 |
| Push Interfaces | **80** | **77** | 3 | **4** |
| Total | **95,85** | **92,15** | 3,7 | **4** |



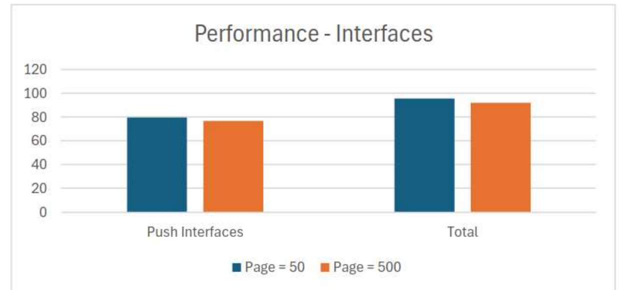
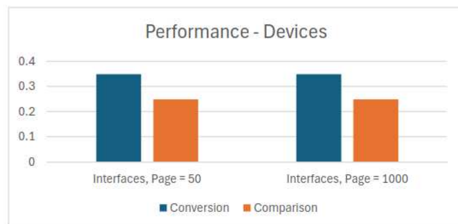




Figure 9.3. Performance for Synchronization.

| COMPARISON | Interfaces, Page = 50 | Interfaces, Page = 1000 | No Interfaces, Page = 50 | No Interfaces, Page = 1000 | Interfaces Overhead | Overhead % | Relative gain |
|---|---|---|---|---|---|---|---|
| Devices Conversion | 0,35 | 0,35 | 0,35 | 0,35 | 0 | 0 | 0 |
| Devices Comparison | 0,25 | 0,25 | 0,25 | 0,25 | 0 | 0 | 0 |
| Vlans Conversion | 26,5 | 25 | 0,85 | 0,85 | **24,8** | **96** | **3** |
| Vlans Comparison | 0,3 | 0,3 | 0,08 | 0,08 | **0,2** | **67** | **0** |
| Total | 27,4 | 25,9 | 1,53 | 1,53 | **25** | **94** | **3** |



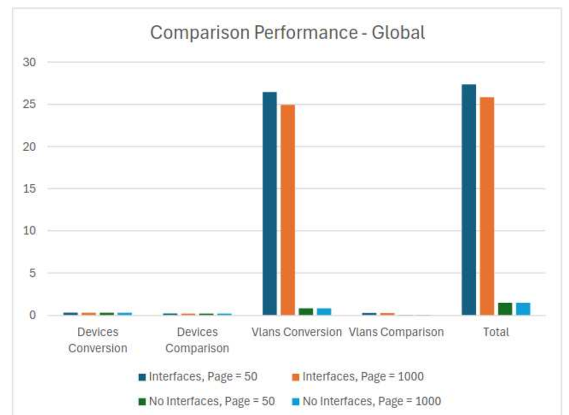




Figure 9.4. Performance for Comparison.

Let's comment on Figure 9.3 and 9.4, considering that we're dealing with 120 devices, 120 VLANs, and 740 interfaces.

**Synchronization** - As expected, paging doesn't affect fetching data from SuzieQ: that comes into play only when performing GET/POST requests to the NetBox API server. The **performance gain** is around 5% for all tables.

Fetching SQ data is so fast that it's irrelevant to the general performance. That's good, as we can expect it to be as irrelevant when increasing the tables' size. Pushing devices takes twice the time than pushing VLANs with comparable size: performance depends on the table cardinality and the number of external references. Both factors must be considered.

Pushing interfaces, as anticipated, is **very slow**: we only have a 6x multiplier when considering the cardinality, but performance is eight times worse than devices and 20 times worse than VLANs. This is probably the part that could use more optimization.

**Validation** - Interfaces do not affect devices, only VLANs. Moreover, the performance gain from paging is negligible, as the elapsed time is short and variations are within the same order of magnitude as statistical variance. Excluding the interfaces, when relevant, leads to a whopping 67% to 96% performance gain.

As expected, a different approach to the interface table could improve the main bottleneck at the cost of additional studying. At least Port Channels and interface modes should be considered.

All in all, **it takes <2 minutes** to perform both functions with Page = 1000. That's not bad considering that, as already said, this kind of operation won't be performed often. Also, fetching data from SuzieQ is not a significant overhead, and given that operations are bulked and paging can be further increased, we expect **less than linear increase** for elapsed time with increasing data.

**Scalability testing** - You will now see the results of a scalability test, trying to run the program with 10,000 devices. The device table was chosen because the VLAN tables would've required that referenced devices existed. Generating two random tables with cross references would complicate the task quite a bit.

A DataFrame with **randomly generated fields** was generated. To simulate a meaningful data set, it must be understood how many values should be provided for each field:

- Unique devices: 120;
- Unique namespaces: 9 ($\frac{1}{12}$ of total);
- Unique hostnames: 32 ($\frac{1}{4}$ of total);
- Unique statuses: we already know the permitted values and they are three;
- Unique vendors: 5 ($\frac{1}{24}$ of total);
- Unique models: 7 ($\frac{1}{17}$ of total).

A DataFrame with 2000 unique hostnames, 670 unique namespaces, three statuses, 50 models, and ten vendors was generated.

Let's comment on the last numbers: there can be up to **V*M** device types - if **V*M < D** - where D is the number of unique devices, V is the number of unique vendors, and M is the number of models. In a production network, devices are typically bought and organized in large groups, with a reasonable number of devices for each device type. For instance, 2,000 device types and 10,000 devices are not a likely scenario. It is also recommended to avoid buying from multiple manufacturers unless strictly necessary since different devices may not work well together.

| 10,000 devices | Page = 50 | Page = 2500 | Absolute gain | Relative gain % |
|---|---|---|---|---|
| Push Devices | 175 | 160 | 15 | **9** |
| Compare Devices | 1,2 | 1,2 | 0 | 0 |
| Total | 176,2 | 161,2 | 15 | **9** |

| Page = 50 | Push | Comparison | Total |
|---|---|---|---|
| 120 devices | 10,35 | 0,6 | 10,95 |
| 10,000 devices | 175 | 1,5 | 176,5 |
| Time increase | **17** | **2,5** | **16** |

Figure 9.5. Performance: 120 devices vs 10,000 devices.

Figure 9.5 shows **outstanding results**. With 83x more devices, we only have a 16x overall increase in the elapsed time. The program's behavior at scale is excellent, and we can expect it to work well for big networks. The total of three minutes is also impressive.

## 9.2 Good Programming

When coding, developers often focus on getting the job done, forgetting about all the rest. There's a lot more to be considered (20), so you will now see some good programming practices which were followed. It will be clear why they're important, what was done in that regard, and, in the next chapter, what could be improved further.

Following **standards** is crucial to attain consistency, readability, scalability, and reliability in a software solution. Let's explore some of them:

1. **Naming conventions** - using descriptive and meaningful names for variables, functions, classes, and modules enhance code readability and maintainability. When programming, choose either CamelCase or snake_case according to the conventions of your programming language. The entire codebase must consistently adhere to naming conventions. Avoid using single-letter variables, gibberish, or ambiguous abbreviations.

   **The code** - My coworkers didn't have any problem with this when studying my code: snake_case was always used, as it's Python's preferred convention, and ambiguous names were avoided. Similar functions have been named similarly, which is a good thing. However, some "tmp" variables need to be removed, and the code should be more compliant with the rest of the SuzieQ codebase.

2. **Comments and documentation** - Comments are useful to provide context and explain what their code is doing. They should be used sparingly so as not to clutter the code. Comments can help explain the purpose of your code, return values, potential errors, the role of classes, and any particularly complex steps in a function. Overall, comments can improve the readability and understanding of your code.

   **The code** - After active programming ended, the code was thoroughly commented. Each function had its parameters and purpose explained, some variables were explained, and some intricate steps were commented on step by step. My coworkers were satisfied too.

3. **Organize code** (21) - code should be grouped into blocks or functions, and comments on their purpose should be provided. Each class should only have one responsibility, and it should be possible to add functionalities without editing existing code. Subtypes should be consistent with

the base types to avoid errors, and there shouldn't be dependencies with irrelevant classes for the task.

**The code** - After creating the first working version of my code, I spent months restructuring it. As you can see, there are now two main classes and some sub-classes. The code is also organized in different files, each with a different functionalities. As this is the first time I've dealt with a project of this size alone, there is room for improvement. The main files can be further divided and decluttered, and some methods may need to be rewritten.

4. **Code reusability** - write code that can be reused efficiently. You can achieve this by creating functions that are as generic as possible, encapsulating similar yet different functions in wrappers so that it can call the appropriate one in a way that's transparent to other modules.

   **The code** - The facade component was created for this purpose specifically. Other tools could potentially utilize its methods by simply providing the necessary dictionaries.

5. **Use short line length** (22) - Using shorter line lengths, such as 80 characters, enhances readability and makes it easier to manage your code.

   **The code** - The 80-character limit is rarely exceeded, and the worst lines of code are usually around 120 characters long. There's a little room for improvement especially in the push_something functions.

6. **Version control** - Commit messages should be as informative as possible to let you and other developers keep track of the changes, why they were introduced, and how. This way, code can be studied and troubleshooted faster. A reasonable branching strategy should be applied, and pull requests should be used for seamless collaboration among coworkers.

   **The code** - The first version was fully developed offline, but that's because it was just a way to get to know the fundamentals of the program logic: it was not intended to be final in any way, and it took a reasonably short time to develop. After that, commits were pushed regularly, with exhaustive logs. A separate branch was used and a Pull Request draft was provided.

7. **Review** - Peer reviews are crucial for writing high-quality code. When someone outside reviews the code, it is easier to identify any weaknesses. Additionally, different people can provide unique insights on how to improve the code. Peer reviews can also help identify and fix issues earlier, reducing the costs associated with addressing them later on.

   **The code** - I consistently sought my coworkers' perspectives, with moderation. That led me to write code faster and better, as some choices wouldn't have been possible without an external, impartial eye evaluating the situation.

Coding standards may depend on the used programming language and evolve, but they're adaptable and likely valid in the future. As changes may occur, it is advised to **stay up-to-date** with the latest version to facilitate collaboration among developers.

# Chapter 10

# Conclusions and Future Works

In conclusion, results show that the proposed solution is good in terms of good programming, performance and scalability. That made it ready for my coworkers to put it to market as soon as possible. They also confirmed that the proposed logic was solid and didn't need any serious makeover.

Some **successful strategies** were the detailed preliminary studies - with rich documentation - and the mixed approach between top-down and bottom-up. This ensured minimal need to rewrite code, easy communication of the problem specifications to my coworkers, and a fairly fast progress pace.

Some **weak spots** were the code structure, as my coworker had to declutter and rebase it a bit, and some naive choices on my part. In particular, the facade was too opaque and the interfaces management was too complex and slow. Also, in the last coding phases documentation disappeared altogether, leaving GitHub commits alone in explaining the process.

We will now delve more into details, outlining future prospects for this feature.

## 10.1 Future Works

In the beginning, new functions to integrate into the solution were proposed along more concrete objectives that were unlikely to make it into my final iteration (Tier 3 and 4 objectives).

After the process of implementing the solution and discussed it and its limitations with my coworkers – who are currently studying the code to integrate it into the SuzieQ suite – we can be far more precise. In this section, you will see some remarks on the objectives to realize sooner or later. This kind of consideration can partially be found in the previous section, so I will not repeat myself.

**Improvements** - The existing code can be improved in some ways:

- The output shows results – failed and successful items and their numbers – as soon as they're available, but it was left barebone. For instance, it shows mismatching rows without providing a more detailed explanation. My idea: process the merge results further. The `left_only` rows should be compared with the right table's ones and vice versa to see which field was the reason for the failure;
- My coworkers found the facade too generic and kinda of difficult to understand. Although it's good that the solution's particularities are left hidden, some methods were considered too opaque.
- Even if the NetBox class already includes various common methods to get sites and tags, there is still room for further generalization. For example, URLs are heavily used, but there's no way to generate them automatically. Writing them manually is error-prone;
- VLAN management could be simplified as my 1:1 mapping approach is a little convoluted and slows down pushing. This also requires studying interfaces on a deeper level and then pushing them all instead of just the ones in the VLAN table;
- The management of table lifecycles, especially related to sessions, needs further refinement. If the program execution is stopped abnormally, an "Unclosed client session" error will occur.

**Additions** - To make this solution more complete they could add:

- Filtering on rows and columns to speed up computation;
- The "platform" information to the device table. SuzieQ holds the needed information in the "os" and "version" fields, which are already included but not used later;

- VLANs and interfaces need related devices to be already present: there's a chain of dependencies to be respected. That may apply to new tables in the future, so the configuration model should enforce a precise order in the validation and synchronization process. For instance, a user shouldn't be allowed to try to push VLANs if no devices were pushed before;
- Other tables could be supported in the future.

On top of that, you must keep in mind that studying a problem is one thing while **selling the solution to customers** is an entirely different challenge. Some details that are important to users can be invisible to developers. The first months after commercialization will be crucial to see how our considerations hold up in a real-world scenario.

# Bibliography


[1] KEARY T. The OSI Model Explained: What is the OSI Model? | Comparitech; 2018. Available from: https://www.comparitech.com/net-admin/osi-model-explained/.

[2] Contributors W. Server (computing). Wikimedia Foundation; 2019. Available from: https://en.wikipedia.org/wiki/Server_(computing).

[3] Moreno V, Reddy K. Network virtualization. Cisco Press; 2006.

[4] Townsend W. Defining Network Observability And Assessing The Market Leaders; 2023. Available from: https://www.forbes.com/sites/moorinsights/2023/06/29/defining-network-observability-and-assessing-the-market-leaders/?sh=6ca20ee16935.

[5] Saha S. Observability Platform Market; 2022. Available from: https://www.futuremarketinsights.com/reports/observability-platform-market.

[6] Coombs B. Enterprise Network Observability with Kentik - TFD27; 2023. Available from: https://www.techdoodles.co.uk/blog/2023/4/1/enterprise-network-observability-with-kentik-tfd27.

[7] Meyer D. Cisco widens its ThousandEyes vision with enhanced observability; 2023. Available from: https://www.sdxcentral.com/articles/interview/cisco-widens-its-thousandeyes-vision-with-enhanced-observability/2023/06/.

[8] Ly A. ThousandEyes Review 2024 Pricing, Features, Shortcomings; 2024. Available from: https://www.betterbuys.

com/network-monitoring/reviews/thousandeyes/#:~:text=ThousandEyes%20At%20A%20Glance%20Good%3A%20The%20solution%20offers.

[9] Anand A. Latest Top 11 Observability Tools in Spotlight - 2024's Guide | SigNoz; 2024. Available from: https://signoz.io/blog/observability-tools/.

[10] Club TC. HoneyComb Observability Tool In-Depth Review; 2023. Available from: https://thectoclub.com/tools/honeycomb-review/.

[11] Guide TS. Auvik — Pricing, Review, and FAQs; 2023. Available from: https://www.thesmbguide.com/auvik-reviews.

[12] Reviews U. Splunk Enterprise Reviews Ratings 2024; 2023. Available from: https://www.trustradius.com/products/splunk-enterprise/.

[13] VanDeraa J. Using NetBox for Ansible Source of Truth; 2020. Available from: https://www.ansible.com/blog/using-netbox-for-ansible-source-of-truth.

[14] KaonBytes. Netbox Dynamic Inventory for Ansible as a feedback loop; 2022. Available from: https://kaonbytes.com/p/netbox-dynamic-inventory-for-ansible-as-a-feedback-loop/.

[15] Barry J Jim Lopez. NetBox review: Simplifying Network Documentation - Compsmag; 2023. Available from: https://www.compsmag.com/reviews/netbox-review/.

[16] Basel K. Python Pros and Cons: What are The Benefits and Downsides of the Programming Language; 2018. Available from: https://www.netguru.com/blog/python-pros-and-cons.

[17] Contributors W. Python (programming language). Wikimedia Foundation; 2019. Available from: https://en.wikipedia.org/wiki/Python_(programming_language).

[18] S S. What Is Pandas in Python? Everything You Need to Know; 2022. Available from: https://www.activestate.com/resources/quick-reads/what-is-pandas-in-python-everything-you-need-to-know/.

[19] Kurinna O. Pydantic explained: revolutionizing Python data management; 2023. Available from: https://www.apptension.com/blog-posts/pydantic.

[20] Dixit V. Coding Standards and Guidelines: A Comprehensive Guide With Examples And Best Practices; 2023. Available from: https://www.lambdatest.com/learning-hub/coding-standards.

[21] Suvariya R. SOLID Principles — explained with examples; 2019. Available from: https://medium.com/mindorks/solid-principles-explained-with-examples-79d1ce114ace.

[22] Gregori S. Ask Hackaday: Are 80 Characters Per Line Still Reasonable In 2020?; 2020. Available from: https://hackaday.com/2020/06/18/ask-hackaday-are-80-characters-per-line-still-reasonable-in-2020/.