>>> network .toCode()

<packt>

1ST EDITION

Network Automation with Nautobot

Adopt a network source of truth and a data-driven approach to networking

JASON EDELMAN | GLENN MATTHEWS | JOSH VANDERAA
KEN CELENZA | CHRISTIAN ADELL | BRAD HAAS
BRYAN CULVER | JOHN ANDERSON | GARY SNIDER

Foreword by Mike Bushong, Vice President, Data Center at Nokia

Network Automation with Nautobot

Adopt a network source of truth and a data-driven approach to networking

Jason Edelman | Glenn Matthews | Josh VanDeraa
Ken Celenza | Christian Adell | Brad Haas
Bryan Culver | John Anderson | Gary Snider

‹packt›

Network Automation with Nautobot

Group Product Manager: Pavan Ramchandani
Publishing Product Manager: Khushboo Samkaria
Book Project Manager: Ashwin Kharwa
Senior Editor: Athikho Sapuni Rishana
Technical Editor: Nithik Cheruvakodan
Copy Editor: Safis Editing
Proofreader: Athikho Sapuni Rishana
Indexer: Hemangini Bari
Production Designer: Aparna Bhagat
DevRel Marketing Coordinator: Marylou De Mello

First published: May 2024

Production reference: 1300424

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN 978-1-83763-786-7

www.packtpub.com

To the Nautobot and wider network automation community, for showing us that change was needed and that if you focus time, energy, and development on the right areas, good things will happen. Thank you to the Nautobot community for the continued support and the fostering of an environment that is welcoming and makes it okay to challenge the status quo.

– The authors

Foreword

George Bernard Shaw wrote in his 1905 play *Man and Superman* the age-old quip "*Those who can, do; those who can't, teach*." It's no doubt a catchy line, but I think it misses the mark a bit. Theory can be understood without being practiced. But practice cannot be mastered alone by high-level engagement with theory.

I have been working on network automation since 2007. And I can tell you with absolute confidence that tools and methods have existed for literally decades at this point, but the vast majority of network operations are still painfully manual. Now, this fact hasn't been missed by networking vendors and would-be technology entrepreneurs. But if the pain is so acute, the ambition so strong, and the solutions so plentiful, why is this still such a struggle?

In my not terribly humble opinion, it's because the gap between theory and practice has never been wider. And even worse, with every new technology cycle, the gap gets bigger as technology after technology leads to promise after promise. All without an on-the-ground understanding of the actual networks and the people who manage them.

I have known Jason and members of the *Network to Code* team for years. As I have dabbled in strategy (yet more theory), they have doubled down on practice. And out of that practice has emerged a set of core principles accompanied by real-life experiences. It's these that bridge the gap between theory and practice. And frankly, it's what's allowed Jason and the team to develop a solution to a problem that has thus far proved difficult to tame.

This book represents the very best of their collective experience. They have captured details – including specific steps and the thought process required to succeed – that are unknowable by those who watch from the outside and merely opine on what ought to be done. They have transformed a product into a solution.

Most of us have heard the Man in the Arena quote made famous by Theodore Roosevelt. Jason and his team are active participants in the arena. And this book will help convert those spectators with the will to succeed into automation gladiators.

- Mike Bushong

Vice President, Data Center at Nokia

Contributors

About the authors

Jason Edelman is the founder and CTO at Network to Code. Observing how DevOps was radically changing IT operational models for systems administrators and developers, Jason saw an opportunity to combine existing technologies from the worlds of DevOps and software development within the networking infrastructure domain to create holistic network automation solutions. Prior to Network to Code, Jason spent a career in technical sales developing and architecting network solutions, with his last role leading efforts around SDN and programmability. Jason is also a coauthor of O'Reilly's *Network Programmability & Automation* book. He is a former CCIE and has a B.E. in Computer Engineering from Stevens Institute of Technology. He can be found on X as @jedelman8.

Glenn Matthews is a principal engineer at Network to Code and is the technical lead of the Nautobot project. Prior to Network to Code, he worked at Cisco Systems for more than a decade in software testing and software development roles with technical focuses including routing protocols, virtualization, and network automation, including the YANG Suite project. Glenn is committed to designing and developing quality software to help make the world a better place. His academic background includes a master's degree in computer science from the University of Georgia. He lives in Durham, North Carolina, with his daughter and a very persistent cat.

Josh VanDeraa is a 20-year networking veteran who has been doing network automation for the past 8 years. He has worked in large enterprise retail, travel, managed services, and most recently, professional services industries. He has worked on networks of all sizes, bringing multiple network automation solutions to the table to drive real value with Python, Ansible, and Python web framework solutions. Josh is the author of *Open Source Network Management* and maintains a blog site to provide additional content to those on the web.

Ken Celenza is VP of Network Automation Architecture at Network to Code. Ken is an experienced network and automation engineer with over 20 years of experience working in military, consulting, and enterprise environments. Ken leads client engagements at Network to Code as both a developer and an architect and serves as a mentor to network engineers.

Christian Adell is a network software engineer who has played multiple roles related to networking and IT automation. Currently, as principal architect at Network to Code, he is focused on building network automation solutions for diverse use cases, with great emphasis on open source software. He is passionate about learning and helping others to be happier, but also has more hobbies than hours in the day, so working remotely from Barcelona gives him the time and the space to achieve his dreams. Christian is co-author of O'Reilly's *Network Programmability & Automation* book.

Brad Haas is a seasoned professional serving as the Vice President of Professional Services at Network to Code. With a career spanning more than two decades, Brad has been instrumental in delivering innovative technology solutions, particularly in network automation and the integration of software-defined infrastructure. Brad is known for his advocacy of a data-informed approach to automation, ensuring technology aligns with business goals. Brad's career is distinguished by his achievement of numerous technical certifications, encompassing multiple CCIEs as well as a range of cloud certifications. His philosophy centers on using technology not just as a tool, but as a driving force for organizational transformation and growth.

Bryan Culver is an engineering manager at Network to Code, where he is currently enjoying building the team and platform behind Nautobot Cloud. He has served many roles in his career directly and indirectly related to network automation, from templating configs while racking data centers to deploying automation solutions in enterprise environments. He has a strong software engineering background, having worked in software development with start-ups and Fortune-sized companies. Outside of work, he enjoys time with his amazingly supportive wife and children, wielding power tools on any number of home renovation projects, traveling to beaches, and watching Formula 1 races.

John Anderson is a principal consultant at Network to Code and the Nautobot product owner, responsible for the direction of the project. John has 10 years of experience in network engineering and software development in higher education and global enterprise environments. He has been a maintainer and contributor to a number of network automation projects over the years. John lives in Charleston, SC and is working on a Ph.D. in computer science with a focus on zero trust network security, at Clemson University.

Gary Snider is a software engineer with 10 years of experience in network automation for global corporate networks and 10 years of experience in routing, switching, and network security. He has designed and maintained data center, branch office, and large campus networks for state and federal government. Gary is a core developer for the Nautobot project at Network to Code.

About the reviewers

Eric Chou is a seasoned technologist with over 20 years of experience. He has worked on some of the largest networks in the industry while working at Amazon, Azure, and other Fortune 500 companies. Eric is passionate about network automation, Python, DevOps, and helping companies build better security postures. Eric is the primary inventor or co-inventor of three U.S. patents in IP telephony and networking. He shares his deep interest in technology through his books, classes, and blog, and contributes to some of the popular Python open source projects.

I would like to thank my wife, Joanna, and my kids, Mikaelyn and Esmie, for inspiring me to be the best version of myself.

Tim Fiola, in automation since 2009, advocates for network engineers to embrace automation. Starting as a network engineer, he delved into Junos automation in 2009, crafting solutions and authoring *Day 1 – Navigating the Junos XML Hierarchy* for Juniper Networks. Starting out with Python in 2012, he went on to automate network planning for cloud providers and automated device upgrade workflows using SaltStack. Coauthoring *This Week – Deploying MPLS* showcased his expertise in RSVP and MPLS services. His open source project, pyNTM, simulates traffic failover in wide area networks. Joining Network to Code in 2021, he continues championing Python for network engineers, emphasizing the value of automating to free up time for high-value tasks.

I'd first like to thank my family, who supports me and tolerates my nerdy tendencies.

Professionally, I want to give a large shout-out to those technical experts who tolerated and still tolerate my persistent questions when I am having trouble understanding a complex topic. Without your patience and kindness, it'd have been a much tougher road.

Finally, thank you to the NTC team, where the work continues to challenge me and teach me every day.

Cristian Sirbu is a consultant, trainer, and community builder, with a particular interest in infrastructure design, automation, and solving business problems with technology. He's been in the industry for a while (getting his CCIE #43453 in the process), building, breaking, and fixing networks of various sorts and sizes. He currently lives in Ireland, helping businesses around the world understand network automation and learn about the technologies that drive it. Ever since being introduced to Linux back in high school, he has loved free and open source software. So, today Cristian's focus is on building the Nautobot ecosystem together with the talented folks at Network To Code and its worldwide community of practice.

Table of Contents

Part 2: Getting Started with Nautobot

3

Installing and Deploying Nautobot 67

4

Understanding the User Interface and Bootstrapping Nautobot 107

5

Configuring Nautobot Core Data Models 147

6

Using Nautobot's Extensibility Features 187

7

Managing and Administering Nautobot 253

Part 3: Network Automation with Nautobot

8

Learning about Nautobot APIs – REST, GraphQL, and Webhooks 311

9

Understanding Nautobot Integrations for NetDevOps Pipelines 371

10

Embracing Infrastructure as Code with Nautobot, Git, and Ansible 423

11

Automating Networks with Nautobot Jobs 453

12

Data-Driven Network Automation Architecture 515

Part 4: Nautobot Apps

13

Learning about the Nautobot App Ecosystem 551

Appendix 2

Appendix 3

Preface

In an ever-changing world that is multi-vendor, multi-domain, and multi-cloud, there needs to be a consistent and holistic approach to network automation. Having a data-first approach provides consistency from day one. Consistent and uniform data powers pervasive network automation. Moreover, the process of data curation and data management is one of the most, if not the most, time-consuming tasks and problems of network automation. Consider these questions. What data should be used in a network change? Where does that data come from? The answer is, the Network Source of Truth!

A source of truth or data-first approach changes what is possible for network automation. It attacks the problem head-on and provides the path for long-term success. Network data is the foundation of defining intent and allows users to finally answer the question, what is the intended configuration (rather than what is the current configuration)?

Data is the foundation of network automation. This is made possible by adopting a Network Source of Truth strategy that defines the intended state of the network. Having clean and quality data inside the Source of Truth results in trusted data being deployed by the automation platform and onto the network.

Nautobot is an open source Network Source of Truth for enterprises looking to adopt a data-driven approach to network automation and a platform that complements any network automation journey. Nautobot is open source and has a growing open source ecosystem of Nautobot Apps that help users all over the world take back control of their network.

Come along for the ride and learn how Nautobot can be deployed as a Network Source of Truth and network automation platform to power your network automation journey.

Who this book is for

This book is for network engineers who manage and deploy networks, network automation engineers who automate networks and support network engineers, and network developers and software engineers who create software that supports network and automation teams.

What this book covers

Chapter 1, *Introduction to Nautobot*, is a comprehensive overview of network automation, data, and sources of truth. It introduces Nautobot and its key use cases and lays the foundation for the rest of the book.

Chapter 2, *Nautobot Data Models*, dives into the built-in core data models of Nautobot, highlighting the breadth and depth of Nautobot as a Network Source of Truth. It provides an understanding of the relationships between the components that comprise a network modeled in Nautobot.

Chapter 3, *Installing and Deploying Nautobot*, explores the architecture of Nautobot and then takes you through your first Nautobot deployment. You'll learn how to install each core component (Nautobot itself, workers, scheduler, database, etc.) and start to configure and load data into Nautobot.

Chapter 4, *Understanding the User Interface and Bootstrapping Nautobot*, explains how to add devices to your fresh Nautobot installation, including learning about many other attributes and models and how they relate to your inventory.

Chapter 5, *Configuring Nautobot Core Data Models*, dives deep into adding and configuring Nautobot with IP addresses, circuits, cabling and power management, secrets, and modeling high-availability, and covers notes, tags, the changelog, and filter forms.

Chapter 6, *Using Nautobot's Extensibility Features*, demonstrates how flexible Nautobot is by leveraging its extensibility feature set, which allows users to customize Nautobot to their specific network or design. You'll learn about using Git as a data source, Config Contexts and JSON schemas, relationships, and much more.

Chapter 7, *Managing and Administering Nautobot*, focuses on Nautobot platform administration. It enables a platform admin to best administer Nautobot using the `nautobot-server` command and manage permissions, along with tips for upgrading and troubleshooting Nautobot.

Chapter 8, *Learning about Nautobot APIs – REST, GraphQL, and Webhooks*, explains how Nautobot is integrated with other tools by examining its APIs. This chapter first covers its RESTful and GraphQL APIs, then goes into webhooks, setting the stage to learn about Jobs and JobHooks in *Chapter 11*.

Chapter 9, *Understanding Nautobot Integrations for NetDevOps Pipelines*, explores Nautobot integrations with a focus on `pynautobot` and its Ansible collection, while providing an overview of its Docker, Kubernetes, Terraform, and Go projects.

Chapter 10, *Embracing Infrastructure as Code with Nautobot, Git, and Ansible*, focuses on enabling users who use both Ansible and Nautobot together. It provides a deeper look at the Ansible collection, explains how to set up dynamic inventory, and then builds a playbook using various Ansible modules to perform network automation.

Chapter 11, *Automating Networks with Nautobot Jobs*, begins with an overview and an introduction to the Django ORM, then walks through how to create Jobs, migrate scripts to Nautobot, and create self-service forms that allow anyone to execute Jobs. Beyond setup and configuration, Job permissions, logging, and scheduling, approvals are also covered.

Chapter 12, *Data-Driven Network Automation Architecture*, dives into network automation architecture and highlights why data-driven network automation is the best approach to guarantee success

in a network automation journey, and explains how this is accomplished with Nautobot and its surrounding ecosystem.

Chapter 13, *Learning about the Nautobot App Ecosystem*, demystifies the Nautobot app ecosystem and reveals all that the ecosystem has to offer, while highlighting the best is yet to come and is in the hands of the community.

Chapter 14, *Intro to Nautobot App Development*, provides an overview of the developer API that is used to extend Nautobot and create Nautobot apps, ranging from lightweight Nautobot apps that are only data models to full-blown apps that cater to specific outcomes.

Chapter 15, *Building Nautobot Data Models*, covers real-world use cases for building custom Nautobot with a case study of an organization that needs custom data models and walks through the path to create them from start to finish.

Chapter 16, *Automating with Nautobot Apps*, continues building the app from the previous chapter, showcasing how Jobs can be packaged with apps to create an end-to-end network automation solution.

Appendix 1, *Nautobot Architecture*, dives into the internal components of Nautobot, reviewing its use of Django, Celery, Beat, and databases such as Postgres and MySQL for those who want to understand Nautobot at a deeper level.

Appendix 2, *Integrating Distributed Data Sources of Truth with Nautobot*, introduces the problem of managing distributed data sources and explains how Nautobot can be used as part of the solution to integrate and aggregate data by using the Nautobot Single Source of Truth framework. Solving network data problems in large enterprises is not a trivial task.

Appendix 3, *Performing Config Compliance and Remediation with Nautobot*, explains how Nautobot Golden Config can be used to conquer the most common use cases in networking, including backups, generating intended configurations, and ultimately performing compliance and remediation.

To get the most out of this book

You should have basic network knowledge (CCNA or greater), along with at least 6-12 months' experience of using Python and Ansible for network automation, and you should be comfortable with Netmiko, NAPALM, or Nornir. You should understand how to read and use Jinja templates, YAML, and JSON.

Software/hardware covered in the book
Ubuntu 22.04
Python 3+
Ansible 2.16+
Nautobot 2.1

Many of the demos can be followed on the public Nautobot instance hosted by Network to Code at `https://demo.nautobot.com`. *This is mentioned throughout the book.*

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Network-Automation-with-Nautobot`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "In the device management section, search for the `WayneEnt_FW1` firewall."

A block of code is set as follows:

```
devices_url = "https://demo.nautobot.com/api/dcim/devices/"
# adds ams01-leaf-11 to the location AMS01
r = session.post(devices_url, data=json.dumps(payload))
# the UUID of the device will be saved for the next API call
device_id = r.json()["id"]
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
payload = {
       "name": "ams01-leaf-11",
       "device_type": "74cf95a8-4233-46b9-a740-fba4f5dc88d3",
       "status": "9f38bab4-4b47-4e77-b50c-fda62817b2db",
       "role": "869267d8-7d75-4bd3-8a9e-5e6adcf200f6",
       "tenant": "1f7fbd07-111a-4091-81d0-f34db26d961d",
       "platform": "f48fd9e2-45c5-4c2f-aa54-28964edb3e1e",
      "location": "9e39051b-e968-4016-b0cf-63a5607375de"
}
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Click on the **Interfaces** tab; if one does not exist already, you can click **Add Components | Add Interface**."

Tips or important notes
Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message. You can also talk to the authors directly if you join the `#nautobot` channel in the Network to Code slack. Self sign-up is at `slack.networktocode.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share your thoughts

Once you've read *Network Automation with Nautobot*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/978-1-83763-786-7`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Introduction to Source of Truth and Nautobot

This part covers the what and why of network automation, Source of Truth, and Nautobot. It provides you with a general overview of the problems in network automation and how understanding the relationship between data and network automation changes the way you think about and approach network automation. From there, you will learn about Nautobot and how it is used to power enterprise network automation solutions, understanding key use cases and the Nautobot core data models.

This part consists of the following chapters:

- *Chapter 1*, *Introduction to Nautobot*
- *Chapter 2*, *Nautobot Data Models*

1
Introduction to Nautobot

Data-driven network automation powered by Nautobot is gaining momentum across the industry. This chapter provides the foundation required to understand the *what* and *why* of network automation and gives an overview of Nautobot and the role it can play in the greater network automation ecosystem. This chapter will start by uncovering the relationship between data and network automation and how **Source of Truth** (**SoT**), when used with Nautobot, is an integral part of the network automation journey. You'll learn what network automation is, key use cases for network automation, and why you should consider network automation, dive into SoT, and be introduced to Nautobot and the power it can provide on the journey with Nautobot as a SoT and a network automation platform.

This chapter covers the following main topics:

- Introduction to network automation
- Understanding SoT
- Nautobot overview
- Nautobot use cases
- Nautobot ecosystem

Introduction to network automation

If you're reading this book, you've realized you need to think differently about managing your network. And you are not alone. If you ask any network engineer, there is still not a day that goes by when they are not logging into a device via SSH *and doing work manually*. Over the last few decades, the most common approach to managing networks of any size, ranging from tens to thousands of devices, was connecting to the device and using the network **command-line interface** (**CLI**). The network CLI is used to gather data, troubleshoot, and make configuration changes. This remains the most common way of managing networks. However, this is changing.

Over the last 10 years, we've seen significant growth and improvements around the operational models for networks. The **software-defined networking** (**SDN**) era brought us controllers and APIs. Controllers provide APIs and fewer points of management. Rather than manage thousands of devices, it is possible to manage tens of controllers (or fewer in some cases). Independent of the number, the point is that the number of directly managed nodes continues to decrease. The SDN era also shined a light on the programmatic interfaces, or lack thereof, of network devices. We have evolved from SSH and SNMP to APIs – REST APIs, GraphQL, gRPC, and event-driven webhooks from controllers and devices. While SSH and SNMP are still the de facto standards across the industry – even for automation, progress is being made. For that, we need to recognize the progress and celebrate, but continue to demand more.

The progress around network automation has been driven by open source. Before network automation, there wasn't much use of open source in the network industry. The industry is learning from its history – that is, if you solely purchase and use vertically integrated tools, there is less flexibility and you could lose control of your network. With current trends, the belief is that those that adopt even just some open source remain in control and can extend libraries and tools as needed to ensure maximum adoption of network automation in their environment. Don't worry – we'll cover some of the most common open source tools and technologies for network automation in the *Industry trends* section of this chapter.

We'll start by exploring what network automation is, its key use cases, and the value it can provide an organization. From there, we'll dive into SoT and Nautobot.

What is network automation?

Any advanced and hot technology *always* gets flak when there are formal definitions because there are always varying opinions, and that's okay. For this book, our approach is to keep it simple. So, what is network automation? Network automation *is* next-generation network management. Period. We can talk about Python, Ansible, Nautobot, YAML, JSON, REST APIs, NETCONF, RESTCONF, YANG – the list can go on for pages. Here is the bottom line – all of these tools and technologies are being used to improve how networks are managed and consumed daily, which is, simply put, *better* network management. Network automation involves transforming operational models that can radically transform careers and technical and business operations.

One major point you should think about on your network automation journey is that it isn't just about doing *your* tasks better and more efficiently. That is only the starting point. You need to be thinking about how to expose your automation to other engineers, teams, and even non-technical people, thus enabling all parties with the self-service they need to do their job functions.

Let's assume you are automating tasks such as **operating system** (**OS**) upgrades, which involves gracefully moving traffic from one device (and circuit) to another. This is a complex workflow. Sure, this can help you when you need to upgrade a device or perform maintenance on a device, but what about exposing that automation to individual site leads? If this workflow is made more accessible, can this expand who can perform the task using your trusted automation? Does it allow you or your

team to delegate a little more? How often are upgrades happening today contrasted with how often you'd like them to happen?

What about if you had automated diagnostics? What if your **Network Operations Center** (**NOC**), **Security Operations Center** (**SOC**), or service desk could go to a portal, click a button, and diagnose their most common issues? In a manual process, one person opens a ticket, and that ticket remains open and an engineer picks it up. The engineer reviews the request and sees it is a semi-common problem. Maybe they need to check with another engineer or two along the way. After a few discussions, they know where to go, which devices to log in to, and which tools to log in to. They correlate the data gathered between the devices and tools. They ensure things look good and update the ticket. Common workflows like this should be automated.

Would your leadership be astounded to learn that the countless hours needed to gather data, let alone the hours spent formatting to make it look good, can be eliminated with automation? Compliance and reporting tasks often take a lot of engineering time and effort because they involve manually gathering and processing information. Now, imagine being able to automatically create any compliance document or report you need. Documents that include pre/post change tests. Documents required for change control. Reports you need to run monthly, quarterly, or annually for compliance. Reports that verify your devices are operating as expected.

This is network automation.

Network automation use cases

We just discussed some examples of network automation to bring it to life. Now, let's look at some of the most common use cases, including the ones that were already mentioned:

- **Common config changes**: Is your team performing the same types of changes day to day, week to week, or month to month? These are changes such as adding VIPs, turning up a port, adding a VLAN to a switch port, managing firewall policies (also discussed later in this chapter), turning up a new BGP peer, updating routing preferences, adding static routes, and updating zones and ACLs. These changes are ripe for automation because they happen so frequently.
- **Common operational tasks**: These are similar to the previous use case, but they involve performing operations tasks that do not require a configuration change. Some examples include updating SSH keys and certificates on devices, performing a config save or backing up a configuration, copying files to devices, rebooting devices, checking logs, and even performing non-network device tasks such as checking and updating tickets.
- **Mass changes**: While *common config changes* are scoped to a set of devices (this could be just a few devices), *mass changes* are meant to be site, campus, regional, or global. Mass changes include changes such as updating AAA, NTP, or SNMP but could also include changing the format and structure of all interface descriptions on every device. These types of changes don't happen as frequently, but when they do, they are impactful and usually a large project.

- **Data gathering and reporting**: How often is someone you know logging into numerous devices or tools to perform health checks, troubleshooting, or simply to execute a request that comes in for application or network performance degradation? Automated data gathering, reporting, and documentation is not only one of the best use cases for network automation – it is a great area to start with since it is less impactful in the event there is bad automation (because it'd be read-only automation). It could also be added to nearly any other use case producing reports before and after changes or generating compliance reports specific to your team or organization.
- **Configuration and operational state compliance**: Compliance comes in two major flavors and can be best understood by asking the following two questions: *Is the network configured as expected?* and *Is the network operating as expected?* Configuration is easy to understand, but it does mean you'll need to understand the intended state of the network. This is where SoT and data-driven network automation comes into play. We'll cover this in more detail later in this chapter in the *Understanding SoT* section, as well as *Chapter 11*.
- **Pre/post-change state validation**: Similar to the previous compliance use case, pre/post-change state validation is more focused on a defined scope of devices. There may be automation when performing global compliance that only runs daily, but changes are happening continuously. Pre/post change state validation ensures that the network is healthy and operating as expected before and after the change.
- **Firewall policy automation**: How many firewall rules are you adding per day, week, or month? How do you know which firewalls need a new policy? How do you know where in the list of rules the new one should go? Do you know? Could you document this for a fellow engineer? Try. This is the start of firewall policy automation. While the last mile is configuring the actual firewall, the questions prior illustrate that a company's firewall rule change workflow often involves many steps before the actual configuration change.
- **OS upgrades**: While already mentioned briefly, how often are upgrades happening today contrasted with how often you'd like them to happen? How many of your devices adhere to your software standards? How many upgrades can you currently do in a single change window? Do you find yourself watching the console of devices as you upgrade them? Do you run any automation to see if devices have the required disk space before copying the new image to the device? Do you run any automation to verify the md5 checksum of the image after it is copied to ensure it isn't corrupt? Is your network at risk due to vulnerabilities left unpatched? Upgrading devices often happens when needed, versus having a defined cadence. It is never a priority. Automation changes that.
- **Greenfield sites and devices**: If you are repeating deployments, there is room for automation. It may mean adding new top-of-rack switches in the data center, it may mean adding a closet or IDF closet in a growing campus, adding a new retail location, or even a new colocation facility or **point of presence** (**PoP**). Much of the automation discussed here is around the configuration of these devices, but that is the easy part. Site planning and deployment is about data curation and management not to mention each organization's business logic required for deployments.

How do you and your team know which IP addresses, VLANs, ASNs, and overall configuration should be entered on those devices? Is it from spreadsheets or a SoT? Again, more on SoT later.

- **Vendor migrations**: Have you ever not moved forward with changing vendors due to the work effort of migrating configurations? With a properly defined SoT and data strategy, this becomes trivial. Your focus becomes storing the intended state of the network using data, decoupled from any vendor-specific syntax. Syntax for a given vendor is generated by running the data through a set of vendor-specific configuration templates. In a migration, you can generate the desired state configuration for a given vendor by running the data through a different set of templates and then deploying those new configurations. Beyond configuration management, you'll also want to ensure multi-vendor operational state compliance to ensure there are no gaps in visibility during the end-to-end migration.
- **Self-service**: It is critical to think through how a given workflow will be triggered along with who the target user is. *Self-service* does not mean that it needs to be a click-button UI. It may mean an IT tool, CLI tool, pull/merge request, ChatOps, or yes, it may mean a full self-service user-friendly form. The point is that you do not need one way to expose network automation or even one way per workflow. Using an architectural and a platform approach to network automation allows you to expose the same workflow through multiple self-service interfaces. You should cater to your culture and your users. This will drive more adoption of network automation.

It is recommended to use a holistic multi-domain network automation architecture to serve as a platform to meet today's requirements. This architecture will also serve as the foundation for tomorrow's requirements. As you embark on the journey, be cautious about using different network automation architectures for different types of networks and domains. If so, it'll create more issues and give your team even more tools to manage while making it harder to unify standards and processes. In *Chapter 10*, we will talk much more about network automation architecture to ensure a consistent approach to managing networks independent of size, domain, and location.

Why automate your network?

After covering the *what*, let's take a look at the *why*. While many use cases are *horizontal* and can be used by any organization type (or verticals), the actual *why*, impact, and justification will differ per organization. Just to clarify, by *vertical*, we're referring to companies with different business types. A few examples of different verticals include financial services, pharmaceutical, retail, telco/the cloud, manufacturing, accounting/legal professional services firms, state and federal government, K-12 education, and universities.

For some verticals, the network may be the business. It may either be a business enabler or have serious consequences if the network is down. For other verticals, other factors may be a bigger concern. For this reason, the *why* is going to vary widely, and we'll cover general reasons to automate the network. Here are some common examples:

- **Lower costs**: Every leader in every business is always asked by their leaders or directly by finance if there is a way to lower costs. In reality, automation helps lower longer-term costs. The more a company can show how automation lowers costs, the greater the chances are that the automation projects get initial buy-in and long-term support. With some of the use cases already mentioned, costs can be significantly lowered. If a company truly documents each of the tasks required and the time to do each for a workflow (such as OS upgrades or troubleshooting) and verifies the most common incidents, they are going to see drastic savings in time and effort when using automation. Time equates to money. It doesn't mean anyone is getting replaced. However, it does mean that there is more time for more projects, each of which adds more value to the business. Increasing velocity without needing to hire new people is a tremendous cost savings.
- **Enhance security and reduce risk**: In today's world, security is top of mind for everyone; it's integrated into all that we do. No company wants to be the headline in the local, national, or global news. Security-focused automation ranges from automated scans, firewall provisioning, VPN connects and disconnects, compliance and remediation, governance adherence and monitoring, and patch management just to name a few. Even if you are not directly on a security team, you should ask yourself if security can be improved in your domain. Can you rotate passwords more frequently? Maybe change those SNMP community strings? The list can easily go on.
- **Provide greater insight and control**: Data is king and that includes greater visibility into your network and automation infrastructure. Automation can be used to gather data, document data, understand patterns, and compare against known baselines. Sure, there may be tools that provide this in the **user interface** (**UI**). That's a great start, but what about seamless workflows that open tickets, update tickets, send emails, and send chat messages in response to network data that is outside the expected range? With automation, you have the opportunity to get the insights you need to answer the questions you have and know that the answers are contained within the network. Think about that. If you are logging into a few portals, copying data into a spreadsheet, creating Excel formulas, or creating a new document to then turn into a PDF and email, there is a better way. There is an automated way.
- **Increase business agility**: Each business and team is always trying to go faster and also perform activities that are not possible without automation. Organizations need to work smarter and more efficiently. In some cases, it may also make sense to hire more people. However, hiring more people often slows things down because, at a certain point, people can start to get in each other's way. In contrast, automation can reduce cost, improve performance/velocity, increase reliability, and do things that humans just cannot do. One example is automation-enabled self-service, which helps business stakeholders obtain the outcome they need sooner. Automation can also improve business-to-business connectivity, allowing organizations to either recognize revenue sooner (for those that are doing business over those connections, tunnels, or circuits) or start

consuming a new service. Think about deploying a new application in a lab or test environment. If it takes weeks to get a new application and its network and security configurations deployed for each environment (dev, test, UAT, and so on), it may be an aggregate delay of months. This is either delaying employee or customer satisfaction or revenue. Using automation improves this and increases business agility.

In all that you do, keep automation top of mind, and try to understand the business and organization-level benefits for various leaders in your organization.

Persona-driven network automation

While we already looked at network automation use cases and the rationale for automation, let's take a different spin on use cases. There is *usually* never one network team. There are usually teams focused on day 0 or architecture and/or engineering; day 1 or implementation; day 2 or operations. These teams may even span network domains such as LAN, WAN, WLAN, or Security, depending on the size of the network. Recognizing the work of the various teams will help structure automation projects for what's possible *within your team*.

Here is a list of example projects and tasks broken down by the three types of teams often found in network organizations:

- Day 0 or architecture and/or engineering:
  - Ensure configuration standards are documented in a structured and modeled manner that is programmatically accessible
  - Ensure hardware standards are documented in a structured and modeled manner that is programmatically accessible
  - Ensure software standards are documented in a structured and modeled manner that is programmatically accessible
  - Ensure architectural and engineering tests exist within every CI pipeline – for example is there redundancy?
  - Develop automation architecture and framework used by other teams
- Day 1 or implementation:
  - Use automation to generate configurations
  - Use automation to perform configuration changes
  - Use automation for pre- and post-deployment verification
  - Use automation for continuous verification of deployment standards

- Day 2 or operations:
  - Execute network device automation for common troubleshooting tasks
  - Continuously update automation that is used for common troubleshooting tasks
  - Execute network device automation for common changes
  - Ensure automation for dynamically reading emails from ISP/NSPs for circuit notifications
  - Execute automation for gathering and collecting information from various tools and devices to aid in troubleshooting
  - Execute automation for dynamically creating, updating, and closing change management tickets

Industry trends

As we've already discussed, the CLI still dominates the industry. However, each year, month, week, and day brings us closer to transformative and better network management through the use of network automation. In this section, we'll look at several of the trends that are collectively driving the industry forward to do more with less and allow for more efficient network operations.

This list is not meant to be exhaustive, but illustrative of the trends that are driving operational efficiencies and automation:

- **SDN**: SDN took the industry by storm in the 2010s. Most modern network architectures include controllers that simplify management and visibility and provide programmatic access with APIs. Simplified management is made possible because it allows users to manage systems versus managing devices and nodes, which allows more abstract policies to be created and applied. Because they allow for fewer points of management, SDN controllers simplify workflows and integrations using the controller (versus individual device) APIs. With SDN, you may have different controllers and solutions for campuses, WAN, data centers, and the cloud. So, if you are looking for a unified network automation strategy, there will be a bit of integration that needs to happen when it comes to data and orchestration. More on this later.
- **NetDevOps**: We've learned a lot about the DevOps industry over the last 10 years. When we talk about NetDevOps, we're referring to doing DevOps but applied to network infrastructure, engineering, and operations. Here are a few examples that highlight trends:
  - Using Git-based **version control systems** (**VCSs**) such as GitHub, GitLab, or BitBucket. Using VCS enables collaboration while providing traceability and audibility on all software or file-based artifacts (templates, data files, scripts). VCS allows users to create owners of particular projects or sections of a project providing accountability to the respective teams.
  - Using **continuous integration** (**CI**). Organizations that use VCS will require basic CI. CI allows users to create tests that must pass before accepting or approving any changes. These tests focus on ensuring nothing is going to break in the automation or the application. CI can also be applied more directly to the network, enabling **network CI**.

    - Implementing **network CI**. If the initial CI tests pass on code and static files, users can do tests such as pre-change analysis based on models of the network (mock devices or real equipment, if you have a larger budget), running active tests on the network (does the network need to be a certain state before making the change?), perform the actual change, and then finally ensure the network is operating as expected after the change.

  While DevOps and NetDevOps can be talked about for days, the actual industry facts show that nearly every network automation project in the world includes version control, automated tests, and some level of CI. If your organization is one of the few that aren't using these three key items, be sure to explore them as soon as you can.

- **Open source**: Many open source tools are used in the DevOps ecosystem. The same holds for NetDevOps. We'll mention some of the most common tools in the *Tools and technology* point covered in this section. Regardless of the tools deployed, it is more important to understand the real *value* of open source. In the context of open source, the real value lies in its *extensibility*, *ecosystems*, and *community*. Extensibility and ecosystems can drastically change and improve what's possible on your network automation journey. Keep in mind that each of these is predicated on the fact that there is a strong community at the foundation. Extensibility is what should give you confidence that no matter what decision is made for your network, you can adapt and change to account for that decision. A *change* may be as simple as upgrading to the latest version of software, migrating from vendor A to vendor B, or migrating from a traditional network to a controller-based network. In any of these scenarios, an organization needs to be confident that its automation can be tailored, updated, or augmented for their needs. While certain commercial tools offer extensibility, it is usually limited and extensibility features tend to be in a perpetual state of *coming soon*. Ecosystems built around community also play a critical role in open source software, further enhancing what is possible with particular open source projects. Ecosystems are usually fostered around extensions, adapters, apps, or add-ons that are outside of the core open source project but are powered by it. It is these ecosystems that usually incorporate the solutions required for true multi-vendor management and automation. The point is not that everything needs to be open source, but that open source software and solutions should either lead or complement any network automation strategy. If they do not, there may be a great risk to the success of the automation journey three to five years out.
- **SoT**: Since you're reading this book, you've likely heard about SoT. In fact, the main topic of this book is Nautobot! At its core, Nautobot is a network SoT that is actively being developed specifically for network automation environments. A SoT is a growing industry trend and probably why you're reading this book, but the short overview of a SoT is that it is the location where you can define the intended state of the network. This is the truth; it is what should be. The SoT is not what is on the device or network. That is referred to as the actual or observed state. The intended state, or SoT, can be extrapolated and used to document the intended configured state and intended operational state, or even used as the place to define the intended state for monitoring thresholds and events. Overall, it allows for greater governance of network data with a focus on what should be in a manner that is often vendor-neutral. We'll spend much more time on SoT in the next section and throughout every other chapter in this book.

- **Self-service**: We covered self-service in the *Network automation use cases* section, but to restate it one more time, the notion of self-service is not one-sided. Those organizations that are successful on their network automation journey understand that it is about having the right mapping of workflows to people (consumers) and from those people to the right user interaction, or the right tool to execute and request that automation. If you get this wrong, there is a great chance to end up with network management systems that aren't used, which will take us back a few decades.
- **Streaming telemetry**: SNMP has been around for decades, and network visibility as we know it is largely based on SNMP. Streaming telemetry is what you may expect when you think about modern network visibility. In this modern era of streaming telemetry, network devices can continuously "push" or "stream" network data to a centralized location. This allows for greater visibility, querying, and trending based on data that would have normally been lost. Wouldn't it be great if the network device could send you the information you need when you need it? Wouldn't it be great if you could turn on a stream of data (collection of data points) from a series of devices on particular interfaces versus getting a response from an interface poll that may kill the device if your poll frequency is too high? Wouldn't it be great if you could build a closed-loop system that can operate in near real time? This is made possible by streaming telemetry.
- **Intent-based networking** (**IBN**): When you look at the key use cases and trends, you can start to see common components of an architecture, such as orchestration, automation, SoT, and telemetry. When these components are fully integrated, the result is an IBN. An IBN is just a comprehensive network automation architecture. It allows organizations to define intent, continuously collect network data (streaming telemetry, SNMP, show commands, and configuration data), analyze that data, ensure intent is deployed, and then react based on intent violations. The reaction to the data may be to remediate or make a change for managing capacity or minimizing the blast radius for a known issue. IBN becomes a natural progression as you start to deploy a holistic architecture for network automation.
- **Artificial intelligence** (**AI**): Our general belief is that a significant amount of automation must be implemented without AI/ML, meaning don't let flashy new tech derail projects and outcomes that are solving today's problems. That said, at the time of writing, we've seen the launch of OpenAI's ChatGPT (`https://openai.com/blog/chatgpt/`), Google's Gemini, and many more services like these. It should be obvious that AI/**machine learning** (**ML**) coupled with **natural language processing** (**NLP**) creating more digital assistants is going to have a transformative impact on where we are as an industry in 5 to 10-plus years as it gets mainstream adoption. Until then, it'll be explored and implemented by pioneers and manufacturers who can make it consumable in a turnkey and meaningful way.
- **Tools and technology**: This is always one of my favorite topics since we live in a product- and tool-centric industry, but let's look at existing tools trends for network automation. From an open source perspective, the dominant tools are Ansible, Nautobot, Batfish, and Terraform. We also see a sprinkling of Salt, but its presence is still largely seen in application and systems automation. Looking at open source from a lower-level library perspective, there is continued

growth with Netmiko, NAPALM, Nornir, pyntc, ntc-templates, and scrapli. If you are using open source or building your solutions, you want to check out these projects. For example, if you need a custom Ansible module or custom Nautobot App, you're more than likely going to consume those libraries to perform your automation. From a telemetry perspective, there is also growth in various stacks that include Prometheus, Influx, Telegraf, and Grafana. Teams that have the skills or are further on their journey can use these stacks to provide greater visibility through data aggregation, data enrichment, extremely powerful queries, and a holistic view of their networks and their IT infrastructure. From a commercial tool perspective, and exclusive of SDN products, we're seeing the most adoption of Itential, IP Fabric, and Forward Networks.

Information

Interested in seeing a comprehensive list of all network automation projects, tools, and products? Check out *Awesome Network Automation* (`https://github.com/networktocode/awesome-network-automation`).

From a trends perspective, we thought it may be worth calling out a few things that get attention at industry events and in social circles, but aren't gaining traction. The first is the direct use of YANG data models within automation tools. They are still mostly used by vendors to define their schema. Of course, there are outliers such as hyperscalers or a select few enterprises, but generally speaking, the actual use of YANG by network teams is not a trend. If you're using an API that is based on a YANG schema, we do not consider that a trend for end users, but it is a trend for certain manufacturers. We'll also call out REST APIs on network devices. While they are becoming more commonplace because the dominant majority of devices in production still don't have APIs, and instead have two or more (different APIs per vendor and OS) ways of performing automation, the majority of device-specific automation still happens via SSH.

Understanding SoT

We've already mentioned SoT a few times. It's finally time to dive in. Let's start by talking about *data*. We'll do that through the lens of making a change on the network.

Let's assume that you want to turn up a new port that's going to terminate a connection to a new building. If you look at other similar configurations on the same device, you're going to find a configuration similar to this:

```
interface vlan100
 description Routed Interface for connection to off campus house
 ip address 10.1.100.1/24

interface GigabitEthernet4/1
 description connects to och-sw-01 GigabitEthernet1/1 (off campus
house)
```

```
 switchport
 switchport access vlan 100

vlan 100
 name off_campus_house
```

Is there any other way to configure the same interface? Could we have used a routed port? Could we have configured a trunk instead? A different prefix? Sure, these are all valid possibilities. The point is that you are going to have your own standards, and they will drive your *new* configuration. When adopting a SoT approach, we need to decouple *data* from configuration syntax.

For example, the standard configuration you copy and paste becomes your template while you extract the data. That data becomes any input that changes to derive a configuration. In this example, the data is as follows:

- **SVI interface**: 100
- **SVI description**: Routed interface for connection to off-campus house
- **SVI IP address**: 10.1.100.1/24
- **Physical interface**: GigabitEthernet4/1
- **Physical interface description**: Connects to och-sw-01 GigabitEthernet1/1
- **VLAN ID**: 100

In reality, both descriptions – that is, the SVI interface and the IP address, could be removed from data inputs since they can be auto-generated from the VLAN ID. We'll see that soon. For descriptions, they can be auto-generated by having a use case or description of the project defined. Let's look at a few examples of showing this data as YAML structured data:

> **Note**
> Teaching YAML and Jinja2 is outside the scope of this book.

```
svi_interface: 100
svi_description: Routed Interface for connection to off campus house
svi_ip_address: 10.1.100.1/24
physical_interface: GigabitEthernet4/1
physical_interface_description: connects to och-sw-01
GigabitEthernet1/1 (off campus house)
vlan_id: 100
```

You may opt to nest some data, like this:

```
svi:
  interface: 100
  description: Routed Interface for connection to off campus house
  ip_address: 10.1.100.1/24
physical_interface:
  name: GigabitEthernet4/1
  description: connects to och-sw-01 GigabitEthernet1/1 (off campus
house)
vlan_id: 100
```

Going one step further, a few values could be eliminated if there is more logic in your Jinja2 template. This one also adds data for the remote peer:

```
physical_interface: GigabitEthernet4/1
vlan_id: 100
connection:
  description: Routed Interface for connection to off campus house
  remote_peer: och-sw-01
  remote_interface: GigabitEthernet1/1
```

Finally, a Jinja template that could consume this data and render a configuration snippet would look like this (focused on one of the devices):

```
interface vlan{{ vlan_id }}
 description {{ connection['description'] }}
 ip address 10.1.{{ vlan_id }}.1/24

interface {{ physical_interface }}
 description connects to {{ connection['remote_peer'] }} {{
connection['remote_interface'] }}
 switchport
 switchport access vlan {{ vlan_id }}

vlan {{ vlan_id }}
 name {{ connection['description'] }}
```

Defining SoT

After looking at a few different ways to represent data, the main point is that we have successfully decoupled data, which is shown as YAML, and syntax, which is shown as a Jinja template. The templates are built or defined by those who own the *standards*. However, data is what needs to be created or updated for any given change. Focusing on the data focuses on a change, without getting pulled into syntactical details that vary per vendor.

This data is now the SoT (*technically, the SoT would be the file that contains the data*).

With our focus on the data, now comes the real questions to ask:

- Why did we pick GigabitEthernet4/1?
- Why was VLAN 100 chosen?
- Why was 10.1.100.1 chosen?
- How did we construct the interface descriptions?

It would be fairly common if you were checking one or more spreadsheets to get this data, but it's more likely that *you just knew* because you're good at what you do and you checked the devices and connections that you most recently deployed.

The idea of a SoT is that it allows you to plan and focus on what should be. A SoT defines the desired state. With a SoT, users manage the data that's used for upcoming changes, which is then programmatically accessed by automation tools during a change. The automation tools access the data, render a network configuration, and then ensure that configuration exists on the network. On your SoT journey, you should be able to build a document that defines one tool as the authoritative source per type of data – for example, ASNs, VLANs, and so on.

Due to the breadth of network data required to manage a production network, often, one or more systems are used as an authoritative source of information to build a configuration. For example, a database might be used for inventory and IP addresses, and another that has policies used for ACLs. The authoritative source of data is the location where updates are made. This is also often referred to as a **system of record** (**SoR**). *It's worth calling out that SoT and SoR are often used interchangeably*:

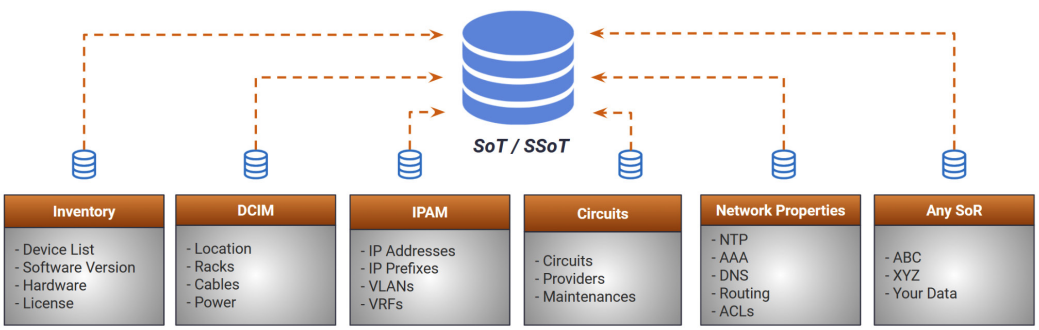

Figure 1.1 – Visualizing SoR, SoT, and SSOT

Generally speaking, the term SoT is a system that stores data from one or more SoRs. However, how often SoR and SoT are used interchangeably, the term **Single Source of Truth** (**SSoT**) is often used to reflect a system that is aggregating data from multiple SOR. This type of system allows relationships to be formed between these datasets and also provides one unified API that can be used to access all network data. Having this data accessed from a single API significantly lowers the amount of work

required by your automation tooling. In *Appendix 2*, we review working with multiple SoTs, doing a deep dive on the Nautobot SSoT application, and discussing other designs used for managing network data.

Approaches to SoT

The previous section described the purist view and the most correct approach to understanding a SoT. It is based on the premise that *the SoT always contains the intended state*. This means that as a user, you change the data and then perform your change using that data. Of course, using automation to fetch the data is the ideal state, but even if you were using it as a documentation store, it's a step in the right direction. The gap in this approach is that the SoT does not always reflect the *actual* state of the network (maybe a user makes a manual change because they don't like automation or they are just fixing something quickly). There should be tooling built around the SoT in this approach that compares the SoT and the actual network. This provides assurance and compliance that the network is operating as expected.

> **Note**
>
> Based on the network technology deployed or your preference, another approach is also possible when implementing a SoT. The alternative is to ensure the SoT reflects what exists on the network. This approach may be used as a one-time event to turn the initial data population into a SoT. This may seem a little confusing because it goes against the purist view of SoT, but we thought it is worth calling out because it is reality.

With the growth of NetDevOps over the past few years, one common place to start with a SoT is to define data in a YAML file and version it in a Git repository. The YAML data is the intended state. That data gets rendered with one or more templates to generate the intended configuration, which is later deployed to the network. This approach provides peer review (through pull and merge requests) on the data before being merged and later deployed and also enables users to run automated tests with CI on the data providing even more assurances the data is good. This approach of defining the data first and having that drive automation is what data-driven network automation is all about.

Due to the plethora of technologies that exist today from SDN and cloud-native networking, networks are not always planned – they may be dynamic. There may be auto-scaling or dynamic policies. In these types of environments, you may prefer to see the actual state in one place. This is also possible by using a SoT. *With this approach, it is more analogous to a discovery engine, but for configuration data.*

It is also possible to employ a hybrid approach. This would mean certain data in the SoT is authoritative and drives the intent of the network, and other data shows what exists in certain domain managers, controllers, or clouds. The general assumption here would be that the data added via controllers or the cloud is authoritative and what is intended to be configured.

Overall, it's always worth remembering that not all purist points of view and ideals can be implemented in a network that has been evolving for 25 years. We need to take a pragmatic approach, but it is important to recognize proper definitions and terminology to ensure everyone embarking on their SoT journey is on the same page.

Keeping the purist view in mind allows us to see the relationship between network data and network automation, given the data is ultimately at the center and driving network automation. The beautiful thing about data-driven network automation is that it allows us to start thinking about abstractions and the level of intent that we want to describe the network.

Even in this book, we're talking about lower-level data, which leads to lower-level intent. However, once you've embraced data, it is possible to build abstractions around design. Consider the earlier example at a higher level of intent:

```
connection:
    source:
        device: nyc-sw-01
        interface: GigabitEthernet4/1
    destination:
        device: och-sw-01
        interface: GigabitEthernet1/1
    type: off_campus
```

In this example YAML data file, you'll notice `off_campus` defined as a *type*. This was not used in the prior example. With logic in your templates and automation, the right data will be generated and then populated in the SoT based on the standard `off_campus` designs for both required devices. You could go one step further and not even choose the devices and let the automation tell you the ports to use on particular devices that have capacity. This will take time, but it starts with repeatable standards (few to no snowflakes) and data, meaning **it starts with SoT**.

SoT tools and products

After learning more about SoT and the role of network data in network automation, we're ready to look at SoT tools and products. The fact is that there are not many tools that focus on network data specifically for network automation. Let's look at some tools that may be used in building out an overarching SoT strategy. Some are more common than others:

- **Nautobot**: It should be obvious and is likely the reason you're reading this book, but we believe Nautobot is *the* SoT for networking. With native models, extensibility, and a framework in place for aggregating data to and from other data sources, it is becoming the de facto standard for enterprises adopting a SoT for network automation. Nautobot is an open source project sponsored by Network to Code. Network to Code's mission is to continue to drive network automation around the world, one network at a time.

- **YAML files**: Usually playing a part in almost every network automation journey, they provide a solid path to getting started and understanding data-driven network automation. In *Chapter 6*, we'll look at integrating YAML files stored in a Git repository directly into Nautobot – showing that with the click of a button, those files and data can be pulled directly into Nautobot.
- **NetBox**: The motivation for Nautobot, NetBox is a solution that models and documents modern networks. NetBox is an open source project sponsored by NetBox Labs. Nautobot forked NetBox when NetBox was at v2.10 and has continued to diverge (as a hard fork (`https://producingoss.com/en/forks.html#:~:text=Hard%20forks%20(also%20sometimes%20called,line%20with%20their%20own%20vision`)) since February 2021.
- **Configuration management databases** (**CMDBs**): More often than not, CMDBs are part of a greater ITSM strategy, including ServiceNow and BMC Remedy. These tools may be used as the SoT for inventory or general asset management but are usually not used to model network configuration data due to a lack of data models, lack of skills, and how these teams are often disconnected from the network teams. These tools are often built off auto-discovery engines with a general trend toward showing what is versus the intended state.
- **Device42**: This is usually seen and adopted for **data center infrastructure management** (**DCIM**) with a focus on inventory, data center design, rack layouts, and IPAM with automated discovery. Similar to CMDBs, there is a focus on auto-discovery with a general trend toward showing what is versus the intended state, but usually not used to model actual network configurations such as routing, interfaces, and more and powering network automation solutions.
- **Infoblox and BlueCat**: Arguably the most widely deployed IPAM solutions, their focus is on IPAM. They also have discovery capabilities. They have some SoT branding and marketing, but usually, it's on discovering IPs versus defining the intent of IP schemes and having that drive automation.

These are just a select few tools that exist on the market and are being used by network teams. What we believe, and the premise for creating this book, is that Nautobot has grown immensely over the past 2 years and fills a gap in the market as an enterprise network SoT catered specifically for network automation. Through the remainder of this book, we hope you'll see what Nautobot has to offer and how it can act as the SoT and nucleus to power your data-driven network automation stack on your network automation journey.

Finally, let's dive into Nautobot.

Nautobot overview

Nautobot is an open source network SoT and automation platform that launched in February 2021. Being an open source company-sponsored project, its maintainers are from the official sponsor – Network to Code. Network to Code is a network automation solutions provider that helps clients around the world build and deploy network automation technology.

It's now been over 2 years since the launch of Nautobot and there has been significant growth, traction, and development by the Nautobot core team, as well as the community. There have been nine minor releases since inception with the second major release, 2.0, that just launched in September 2023. Nautobot 2.0 is a major milestone for the project bringing many new features and improved usability to Nautobot.

Nautobot forked NetBox in February 2021. This was due to the industry's need for a network SoT that had an immense focus on network automation with great flexibility and extensibility capabilities. Nautobot was also created to foster an ecosystem around an open source network automation platform. The details of the fork can be found at `https://blog.networktocode.com/post/why-did-network-to-code-fork-netbox/`.

Some statistics, as of March 2024, regarding the project and community are as follows:

- Over 120 releases, including two major releases, nine minor releases, and 100+ patch releases (on a defined biweekly cadence)
- Over 1,600 members in the #nautobot channel in NTC Slack (self-signup at `https://slack.networktocode.com`)
- Over 110 Nautobot blog posts on the NTC blog (blog.networktocode.com)
- Over 60 Nautobot YouTube videos in the *All Things Nautobot* playlist on the Network to Code YouTube channel

We'll highlight several key Nautobot features in this chapter but will spend a lot more time on them throughout this book.

Nautobot use cases

Before we get deep into Nautobot, let's level set on what Nautobot is as a **network SoT** and **network automation platform**. These are the **two** primary use cases for Nautobot.

These are not mutually exclusive and can be used in conjunction with other solutions. We'll review all of that and more, but let's start with the basics.

Network SoT

We already introduced the concept of a SoT and how it is the foundation for data-driven network automation. Adopting a SoT shifts the paradigm to focus on *intended* state data. At its core, Nautobot is a network SoT. What does this mean?

First off, it probably means a migration away from spreadsheets, which is a big win in itself:

Figure 1.2 – Evolution of implementing a network SoT

The usual next step is YAML and then deciding which data should be in Nautobot. However, these are not mutually exclusive as Nautobot has native Git integration, which allows users to sync YAML files directly into Nautobot. Much more on that later. The following are the power of Nautobot, where you can effortlessly manage your network inventory, define locations, and organize your infrastructure according to your unique needs:

- Nautobot allows you to store network inventory-defining locations, location types, floor plans, racks, and more alongside *custom* location types. In the real world, network devices are everywhere. They are in campuses, buildings, closets, racks, ceilings, locations on a manufacturing plant floor, cars, and spaceships... the list goes on. The goal of Nautobot is to provide an opinionated way to get started but allow users to define an inventory and organization structure that makes sense to them. The Nautobot data model will be discussed in great detail in *Chapter 2*.
- Nautobot allows you to store and model your devices based on vendors (manufacturers), device models, platforms, and roles. All of these are extensible and customizable for your environment. For example, common roles are leaf and spine for the data center, but if you use different roles or naming conventions, it is as simple as adding them.
- Nautobot allows you to store your IP Addresses and prefixes with support for namespaces that allow for overlapping IP space. This is an area where there may be existing solutions in place, such as Infoblox or BlueCat, as mentioned earlier in this chapter. However, IP addresses are required for assignment to interfaces and policies in Nautobot. With the Nautobot SSoT app, it's possible to synchronize data from third-party systems into Nautobot, giving you flexibility if you need it. Having this data aggregated in Nautobot streamlines your automation initiatives.
- Nautobot allows you to store and model circuit data ranging from circuit providers to individual circuits and then allows you to attach them to specific interfaces on a device. Going one step further, it is possible to use the Nautobot Circuit Maintenance app to dynamically parse and read circuit notification emails from providers and update Nautobot accordingly attaching that notification to a circuit and a device.

- Nautobot embraces extensibility by allowing users to add *any* model to Nautobot to store the data they need and how they need it. For example, there are already open source Nautobot applications for Nautobot that allow you to store security ACLs, BGP routing protocol configuration, and device life cycle information such as End-of-Sale/End-of-Life data in Nautobot. This means that as the Nautobot core project continues to evolve, the community and users around the world can add data models they need to continue to store the intent needed to drive their network.
- Nautobot allows users to define the *relationships* that make sense for them. Nautobot has a defined data model, but *relationships* allow users to associate unrelated object types. For example, you can map a VLAN to a rack; you can map an IP address to a device (remember, IPs are assigned to interfaces); you can map a circuit to an IP address; when using Nautobot apps such as Device Lifecycle Management, you can map contracts to devices, and more. The list goes on.
- With flexibility in mind, Nautobot supports a Data Validation API that allows users to write any logic required to accept and add data to Nautobot. While many users use the Data Validation app, which allows for RegEx and ranges in the UI, the Data Validation API allows you to write any Python logic to ensure *your* standards and governance are enforced – for example, naming conventions, preventing certain data from being deleted, and more. All of your data standards can be codified and enforced so that bad data never finds its way into Nautobot.

This is just a glimpse into how Nautobot is a network SoT. The following visual also shows firsthand how Nautobot can power data-driven network automation:

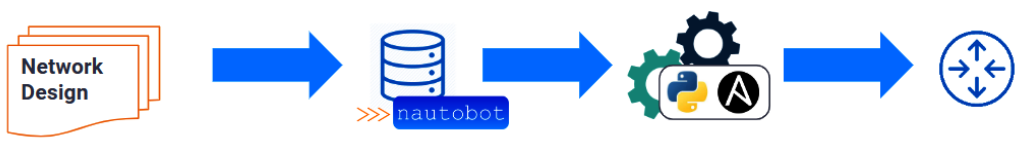

Figure 1.3 – Codifying network designs through data enables network automation

As a network SoT focused on network automation, Nautobot has many features that showcase how it can seamlessly integrate into **NetDevOps environments**. Let's look at a few of those features as a precursor of what will be covered throughout this book:

- **APIs**: From REST APIs to GraphQL to webhooks, data in Nautobot is very accessible. The REST APIs provide your traditional **Create, Read, Update, and Delete** (**CRUD**) operations. GraphQL provides an extremely efficient and user-friendly way to query the exact data you want. Rather than parse through large data sets from a REST API, GraphQL allows users to query for the exact element or elements needed. We'll cover APIs in much more detail in *Chapter 8*.
- **Native Git integration**: Nautobot supports the ability to use NetDevOps workflows, allowing you to store files in a Git repository; then, in the UI, you can configure Nautobot to clone those specific repositories. You can store YAML data, Nautobot jobs, and export templates in a repository and easily clone into Nautobot all from the UI. This ensures you can run CI on your repositories, perform peer reviews, and then, once merged, *sync* those updates into Nautobot.

- **Job automation**: Nautobot Jobs are arbitrary Python code that can be used to perform any task you would script, including analysis of the data in Nautobot and simplifying data management and population, though they can be used to perform actual network automation tasks. Jobs also simplify creating self-service forms to streamline the adoption of network automation. Jobs also supports Job Hooks, which are similar to webhooks, in that when there is a change to data in Nautobot, a job can be triggered. *Chapter 10* is fully dedicated to jobs, so there's much more to come on this topic.
- **Secrets integration**: To perform network automation, there need to be integrations with secrets, credentials, SSH keys, and API tokens. There needs to be intent on which secrets are needed for a location or device. Nautobot has native secrets integration to map secrets to environment variables or files on the system, while also providing more advanced features with the Nautobot Secrets Providers app, which includes dynamic integration with HashiCorp Vault, AWS Secrets Manager, and many more Enterprise Secrets Management tools. This allows users to rotate and change secrets in secrets management or vault platforms with Nautobot fetching them as automation is performed.
- **Flexible location models and dynamic groups**: Nautobot supports flexible location models and allows you to filter on many different attributes. However, Nautobot also supports *dynamic groups*, which are based on the metadata of a given object. With automation, you likely need to automate based on predefined criteria. For example, you may need to automate all devices that are in a given region, are a given device type, and have a given status. So, the next time a device enters *that* status, it's automatically part of that group, so targeting that *dynamic group* simplifies the automation required. Rather than checking the devices, device types, and statuses, you're simply querying for devices in that logical group.

These are merely five ways Nautobot embraces network automation as a first-class citizen. All of these and many more will continue to be covered throughout this book.

Network automation platform

Nautobot is also a network automation platform, thus going beyond a SoT. Let's take a look at this in more detail to understand what this means.

Nautobot jobs

The first major feature to be aware of for Nautobot being a network automation platform is the support of Nautobot jobs.

Nautobot jobs offer users the ability to create self-service forms in a matter of minutes. Self-service is needed to drive the adoption of network automation; Nautobot jobs are the foundation of Nautobot's platform strategy. Imagine having data stored in Nautobot and you want to verify that it is on the device:

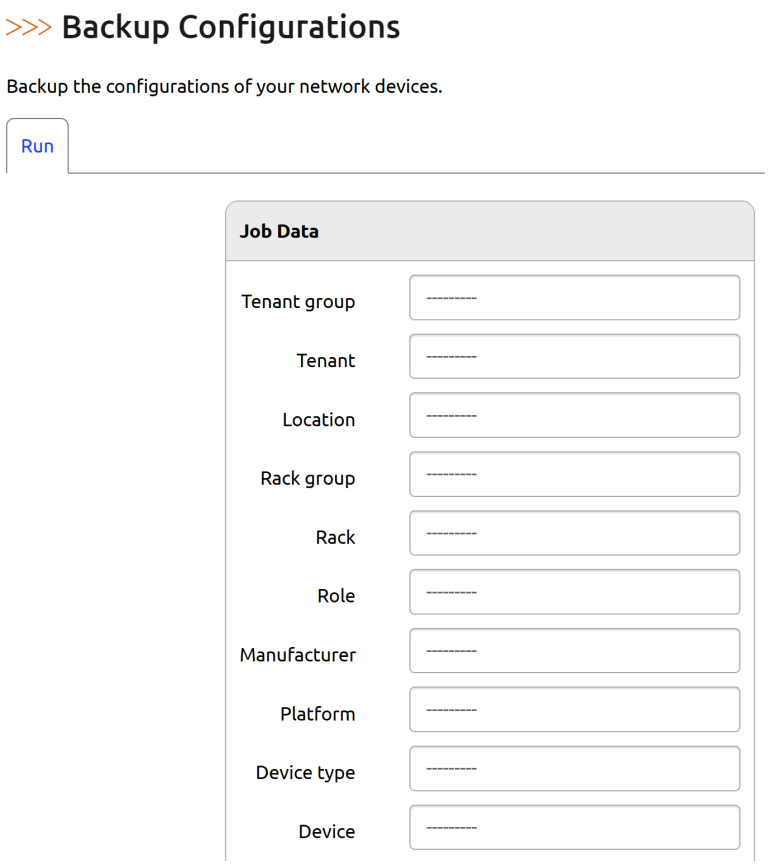

Figure 1.4 – Example of a self-service job form

Usually, there is a need to create some code or automation somewhere, often in another tool. Based on size or scale, that may be needed; but for many environments, tying it into Nautobot as a job makes sense because the data is already there. Keeping in mind that jobs are Python code, that *code* can be stored as a job in a Git repository and easily integrated into Nautobot, thus providing self-service to any user that needs to execute it. This is just a basic example, but any automation task that can be built as a script can be deployed as a Nautobot job. There are already Nautobot integrations to Nornir, which is one of the most common Python-based network automation frameworks in the open source community.

Nautobot apps

Beyond Nautobot jobs, Nautobot as a Platform has a powerful developer API that allows users to create Nautobot apps. Nautobot apps enable users to create APIs, create new views and pages, and create any data model required in Nautobot. Nautobot Apps are what encapsulate specific functionality

and are the entities that are created for specific use cases. Thus, apps can be as lightweight as only modeling and storing new data – maybe you want to model and store SNMP data, maybe you want to model load balancers, and so on. Apps can be heavier-weight Python applications that perform actual network automation tasks:

> **Note**
>
> Nautobot Apps is the new name for Nautobot Plugins. You may see older commentary online and in the code base that says the word plugin, but that is referring to what is now called Nautobot Apps.

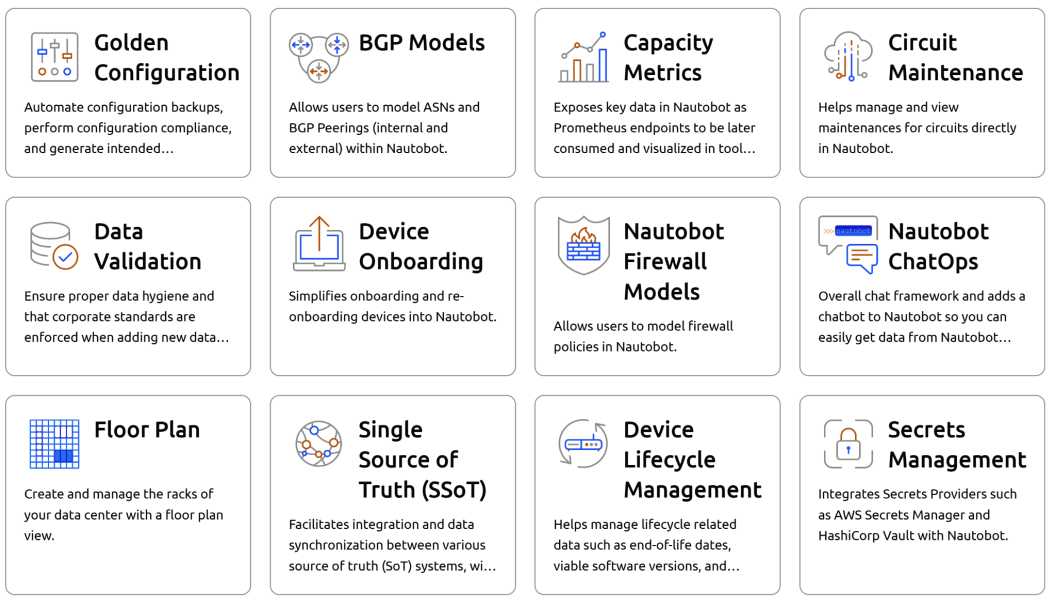

Figure 1.5 – Overview (subset) of Nautobot apps

Nautobot apps leverage the power of Nautobot as a Platform. Using Nautobot as a Platform to construct a network automation application allows users to focus on the actual development without doing the heavy lifting of creating an application from scratch. You get to take advantage of Nautobot APIs, RBAC, logging, GraphQL, relationships, Git as a data source, SSO, and the list goes on. What this means is you can add your own items in the navbar, insert menu items in existing dropdowns, insert new pages, and insert new tables and widgets on detailed object pages. This flexibility allows you to tailor Nautobot to your liking by building Nautobot apps driven by your requirements. Nautobot apps are built at a more accelerated rate than building custom stand-alone applications.

There are already numerous Nautobot apps in the open source community and this number continues to grow. Let's explore some of them.

Nautobot ecosystem

The Nautobot ecosystem is comprised of Nautobot apps that solve a variety of use cases. There are already 15+ open source Nautobot apps written by Network to Code and numerous others written by individuals in the community. Keep in mind that a Nautobot app can be as lightweight or as robust as needed to address the requirements at hand. Here are some examples of different types of applications that could be built using the Nautobot App developer API:

- Lightweight examples:
  - Create a database table, UI views, and API to manage NTP
  - Create a new page (UI view) to aggregate data from devices and VLANs the way you want to see it based on relationships
  - Create Nautobot jobs that are distributed through a Nautobot app
  - Create a command runner that fires off commands to selected devices that are already in Nautobot
- Robust examples:
  - Create an application to store, manage, and deploy firewall policies (inclusive of database tables, views, and APIs).
  - Create an application to discover and crawl the network (inclusive of database tables, views, and APIs).
  - Create an application that performs network configuration backups, generates intended configurations, and performs compliance (which, by the way, exists already in the Golden Config app!). You'll get a deep dive into Golden Config (`https://github.com/nautobot/nautobot-app-golden-config`) with Nautobot in *Appendix 3*.

> **Note**
> There is also a Nautobot app template in the form of a cookie-cutter GitHub repository (`https://github.com/nautobot/cookiecutter-nautobot-app`) that helps anyone create a new app.

If you can't see it already, the opportunities are endless with Nautobot apps.

As mentioned previously, the Nautobot ecosystem already consists of many Nautobot Apps. We'll take a look at a summary of a few of them while diving into a few of these in *Chapter 13*:

- **Golden configuration**: Automates configuration backups, performs configuration compliance, and generates intended configurations (`https://github.com/nautobot/nautobot-app-golden-config`).

- **Floor plan**: Allows users to create a floor plan of their data center or other locations of the racks and devices that exist within Nautobot (`https://github.com/nautobot/nautobot-app-floor-plan`).
- **Version control**: Allows users to have change (workflow) management with approvals when managing data within Nautobot powered by a Dolt database. This is in an alpha state, but watch out for the announcement of the official release (`https://github.com/nautobot/nautobot-app-version-control`).
- **Design builder**: Allows users to create data-driven designs (such as small, medium, and large sites) that then allow you to deploy a new device/site/location with that design, automatically generating the desired data for that design based on your data standards.
- **BGP models**: Allows users to model ASNs and BGP peerings (internal and external) within Nautobot (`https://github.com/nautobot/nautobot-app-bgp-models`).
- **Capacity metrics**: Exposes key data in Nautobot as Prometheus endpoints to be later consumed and visualized in tools such as Grafana (`https://github.com/nautobot/nautobot-app-capacity-metrics`).
- **Circuit maintenance**: Helps manage and view circuit maintenance directly in Nautobot (`https://github.com/nautobot/nautobot-app-circuit-maintenance`).
- **Data validation**: Ensures proper data hygiene and that corporate standards are enforced when adding new data to Nautobot (`https://github.com/nautobot/nautobot-app-validation-engine`).
- **Device life cycle management**: Helps manage life cycle-related data such as end-of-life dates, viable software versions, and maintenance contract information (`https://github.com/nautobot/nautobot-app-device-lifecycle-mgmt`).
- **Device onboarding**: Simplifies onboarding and re-onboarding devices into Nautobot (`https://github.com/nautobot/nautobot-app-device-onboarding`).
- **Firewall models**: Allows users to model firewall policies in Nautobot (`https://github.com/nautobot/nautobot-app-firewall-models`).
- **Secrets providers**: Integrates secrets providers, such as AWS Secrets Manager and HashiCorp Vault, with Nautobot (`https://github.com/nautobot/nautobot-app-secrets-providers`).
- **SSoT**: Facilitates integration and data synchronization between various SoT systems, with Nautobot acting as a central clearinghouse for data. Open source integrations exist for ServiceNow, Cisco ACI, Infoblox, IP Fabric, and Arista CloudVision, but integrations can be written for any remote system. Note that these integrations used to exist as their own dedicated GitHub projects, but were recently consolidated into the main SSoT project. SSoT will be covered in greater detail in *Appendix 2* (`https://github.com/nautobot/nautobot-app-ssot`).

- **Nautobot ChatOps**: Provides an overall chat framework and adds a chatbot to Nautobot so that you can easily get data from Nautobot directly from chat, including Slack, Microsoft Teams, Webex Teams, and Mattermost. This also has out-of-the-box chat integrations for Grafana, IP Fabric, Cisco Meraki, Cisco ACI, Ansible AWX, Arista CloudVision, and Palo Alto Panorama. Note that these integrations used to exist as their own dedicated GitHub projects but were recently consolidated into the main ChatOps project (`https://github.com/nautobot/nautobot-app-chatops`).

Summary

This chapter provided a general overview of data-driven network automation with Nautobot. It started by reviewing key use cases for network automation before highlighting the important relationship between data and network automation. It should be evident that getting an understanding of the data that drives network automation should not be understated and that having good, clean data will simplify the overall network automation journey. Finally, this chapter provided an overview of Nautobot and its two key use cases – SoT and network automation platform, and how both are further enhanced through its developer API and the Nautobot ecosystem that continues to grow with open source apps such as Firewall Models and Golden Config.

In the next chapter, we'll explore and start to understand the data models at the core of Nautobot.

2
Nautobot Data Models

At the core of Nautobot are two main use cases that were covered in *Chapter 1*: a network automation platform and a network **Source of Truth** (**SoT**). While it is true that certain classes of automation can be built void of specific structured data, it is generally true that the SoT powers much of the automation that provides the most business value.

The mantra that holds in Nautobot is that the automation that you rely on is only as good as the worst data you feed into that machine. It is this fundamental truth that drives the SoT aspect of Nautobot. In this chapter, we will focus on the critical data models and relationships of the data in Nautobot that are used to power your network automation stacks.

The following are the main topics that will be covered in this chapter:

- Nautobot data models overview
- Network device inventory data models
- IPAM data models
- Circuits data models
- Data model extensibility
- Custom data models

While this book is focused primarily on enabling you to make effective use of Nautobot in your network automation journey through deeply technical hands-on topics, this chapter will first lay the foundation for the network data model that Nautobot provides. This understanding is key to how you will later use the data model to build automation capabilities through its consumption, and even by extending it to meet your specific needs

Nautobot data models overview

Before we begin, let's touch base on a few terms and concepts. First, *data modeling* refers to defining business requirements for the expression and relationships of data in Nautobot. Thus, the output of our efforts is the *network data model*, which we will dive into now. Such a comprehensive data model is naturally broken down into several high-level data domains, such as the inventory, circuits, and IP addresses. While we logically compartmentalize our network data model to make it easier to manage, it still comprises many cross-model relationships that span the boundaries of the domains. For example, a router has interfaces and those interfaces have Layer-3 IP addresses. These relationships are the key aspect of a comprehensive network SoT that positions it as a better way to manage data than numerous siloed spreadsheets or even disconnected systems.

Data model summary

We will begin our journey through the Nautobot data model with a high-level review of the landscape. Here you will note some of the data domains we spoke about earlier, and hopefully will appreciate the need to logically break the data model up in this way.

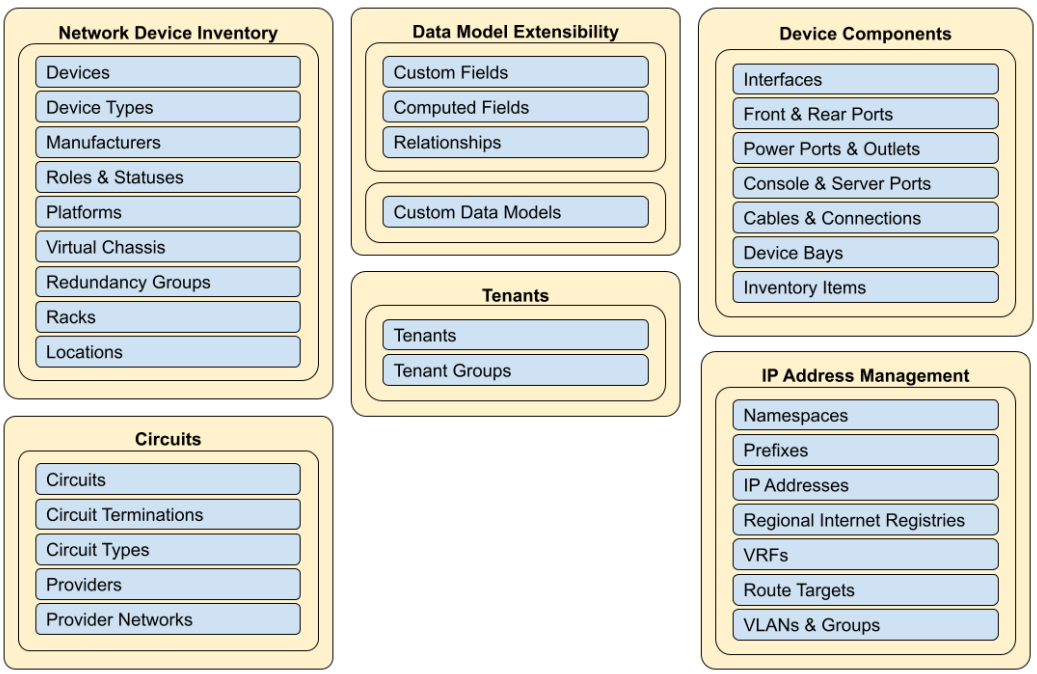

Figure 2.1 – Nautobot data model overview

As you can see, Nautobot affords the ability to track a tremendous amount of specific networking data out of the box. In this chapter, we will dig into each of these domains in more detail. Still, though, you are likely wondering about other parts of the networking world not covered in the preceding list. This is natural, and rest assured that Nautobot has you covered through its extensibility features (covered in *Chapter 6*) and the ability to build custom Nautobot data models (covered in *Chapter 15*), which allow for additional models to be provided through a variety of means. You'll see more on that at the end of the chapter, and even more in *Part 4*, where you'll get to see firsthand how to extend Nautobot.

For now, we will take a closer look at each of these models, their common and important attributes, and what you can do with them.

Network device inventory data models

The foundation of the *network data model* is rooted in the primary entities that a network organization cares about, namely devices. You will find the Device model in Nautobot to be the proverbial heart of the operation, with many ancillary models such as Interfaces associated with it, and tie-ins to other important areas of the data model. You will soon see that there are many aspects of metadata surrounding the Device model that go into constructing a logical and robust view of the world.

Devices

Tracking network devices is arguably the most popular feature of Nautobot and serves as the basis for many other data modeling and automation activities. In Nautobot, a device can represent many different types of network assets, from the obvious rack-mounted router or switch, to firewalls, and even servers. Going further, it is perfectly valid to track virtual networking appliances using the Device model. This also means that a device need not be constrained to a physical rack at all (though tracking racks affords other possibilities that we will discuss later).

Devices in Nautobot track common attributes such as the hostname, serial number, asset tag, and primary IP address (with options for both IPv4 and IPv6). Here is an example of a **Devices** table from Nautobot:

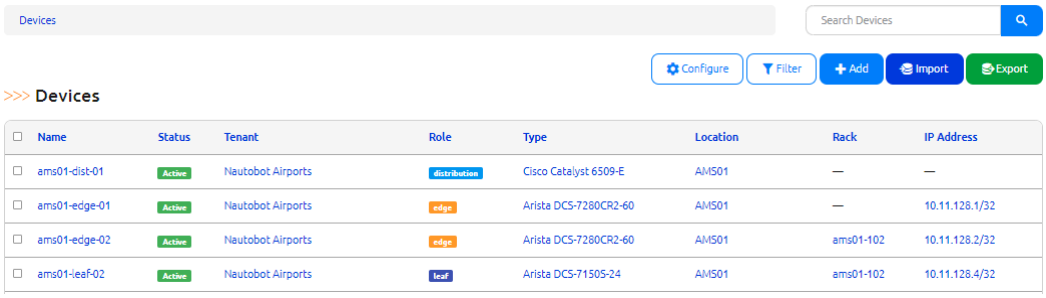

Figure 2.2 – Nautobot Devices table (list) view

Beyond the obvious, we also associate a device with a physical location (more on that in a moment). We can also attribute tenant ownership, which is its own area of the data model. As network engineers, we usually want to know the purpose of a device, and for that, the device model has a linkage to the role model, which is user-definable based on your network. Likewise, the **Status** field allows your organization to define lifecycle values based on how you operate the network. The Platform relationship is commonly employed to designate the software family the device is running, such as Cisco IOS or Juniper JunOS. As in the real world, devices can be located inside a rack, where we track which rack, which position in the rack, and on which side or face the device is installed. Finally, a device has a linkage to a Device Type, which represents the hardware model. In terms of the data model, we could say that devices are instances of a Device Type, and this works in much the same way as if we took a piece of hardware off a shelf or pallet and deployed it on a rack.

You can start to get the sense that the relationships in the data model are what makes Nautobot interesting as a network SoT. Still, though, at this point, we have described only the ability to create Device records with some specific attributes, and while that is certainly important in asset inventory and even automation contexts, we could do that with a spreadsheet! So let's explore some of the more intricate data models and their features in Nautobot.

Device components

With devices being an anchor point in the data model, device components primarily make up relationships to other parts of the model, for example, tracking interfaces and other port types. There are other general component types such as Inventory Items and additional use cases with Device Bays, such as chassis child devices.

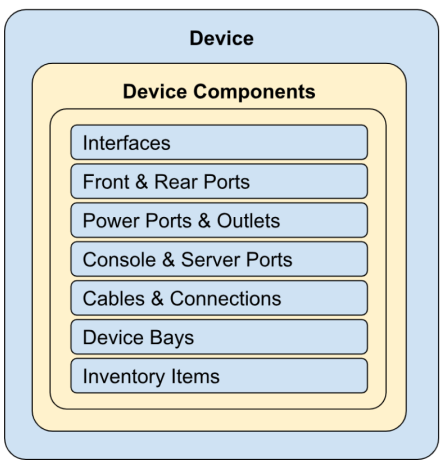

Figure 2.3 – Visual of a device model in Nautobot

We'll now take a look at these other components that can be mapped back to a given device.

Interfaces

Probably the most commonly used device component in Nautobot is the Interface model. Network interfaces play several vital roles in the real world of networking. They provide physical connectivity between devices, logical addressing, and Layer 2 management. So too, Nautobot supports all of these use cases and more. Nautobot can model both the physical and logical interfaces with support for most common form factors and configurations, including LAGs, bridges, and parent/child virtual relationships. Here is a snapshot into an **Interfaces** table inside Nautobot:

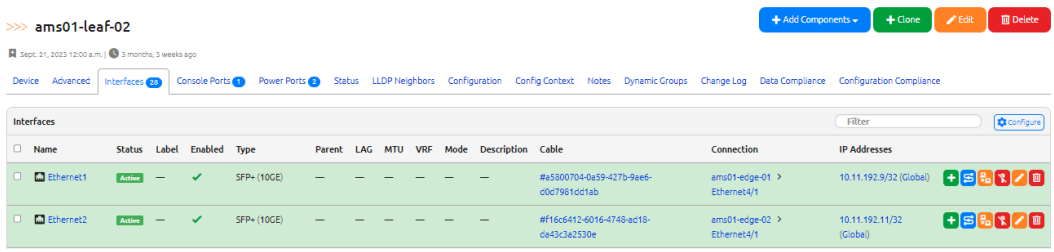

Figure 2.4 – Glimpse into the interfaces of a device

We'll dive much deeper into what's possible in later chapters in the book, but take note of the icons on the right-hand side of each row. You can perform a trace when there are cables connecting two interfaces in Nautobot, which proves to be a valuable function.

Layer-3 addressing is covered with support for primary and multiple secondary IP addresses per interface. You can also specify the 802.1Q mode with lists of tagged and untagged VLANs. You can enable or disable an interface, and make use of customizable statuses that allow for use cases such as tracking the provisioning state of an interface in some business process—it's up to you. Perhaps the most interesting usage of interfaces in Nautobot, though, is the ability to connect them to other interfaces or components, thereby creating a cable, which means you can track your entire physical cable plant if you wish, or simply indicate that two interfaces are connected "in some way." You can also flag such connections as reserved, allowing you to plan capacity (you might also do this by having a "reserved" status on the interface, depending on your use case).

Front and rear ports

The ability to model a cable plant is achieved through the usage of front and rear ports. In practice, these are combined to create patch panels and fiber cassettes in which the front port accepts the connection to an interface or other front port. A rear port is mapped to a front port and allows for multiplexing to model the bundles of cables or shrouded fiber runs that go between termination panels.

Power ports and outlets

Like the physical cable plant, Nautobot can also be used to track the power plant. This starts with power ports and outlets that model PSUs and the corresponding PDUs they plug into within a rack. Here is a view of **Power Ports** within Nautobot:

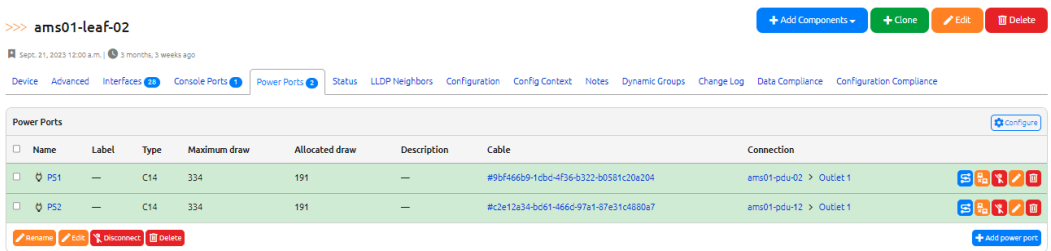

Figure 2.5 – Power Ports tab on a detailed device view

In this model, Nautobot tracks the power draw so you can budget the PDUs and connect them to power feeds and power panels to track the power type (phase, etc.) and distribution. The budget and utilization calculations are visible in a few different areas, such as viewing racks, allowing for effective capacity planning. It's worth noting that as with any of the device component feature sets, you are not required to use them if you simply want to track an inventory of devices.

Console and console server ports

Console and server ports follow the same basic principle as their power-related counterparts but allow you to model the actual console port(s) on a regular network device, but also the console server devices themselves, and the relationship between the two via a connection. You will note the ability to designate the port type, such as DB-25, RG-45, USB-A, and so on, as can be seen in the following screenshot showing the adding of a console port to an instance of a device:

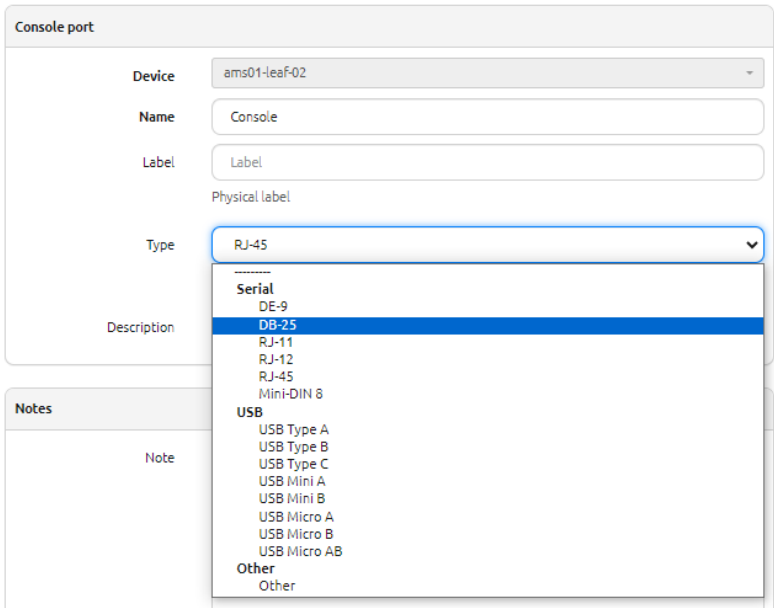

Figure 2.6 – Console port Type options

It's in many of these little details, which are usually never tracked anywhere, that Nautobot tends to shine.

Cables and connections

Having now covered many of the various port types, it is important to understand how they can be connected together. Every connection between device components is represented as a cable, embodying a direct physical link between two termination points. These points could range from console ports to patch panels, or between network interfaces. Each cable is defined by two endpoints, often referred to as A and B, but it's important to note that cables in Nautobot are inherently direction-agnostic, meaning the order of terminations doesn't impact their function. Cables can connect to a variety of objects, including instances of Circuit terminations, Console Ports, Interfaces, Pass-through Ports, Power Feeds, Outlets, and Ports. For each cable, details such as the type, label, length, and color can be assigned. Additionally, an operational *Status* is required for every cable, with default statuses including **Active**, **Planned**, and **Decommissioning**. This comprehensive approach allows for detailed tracking and management of the physical connections in a network.

While the power to track an entire physical cable plant is present, sometimes it is not warranted or necessary. It is perfectly acceptable and possible to treat cables as abstracted connections between Device Interfaces, ignoring the physical aspect, but retaining the context of the connected interfaces.

Nautobot also provides a tracing feature for cables. Users can trace a cable from either of its endpoints, either through the UI or using a REST API endpoint. Here is an example of a simple cable trace:

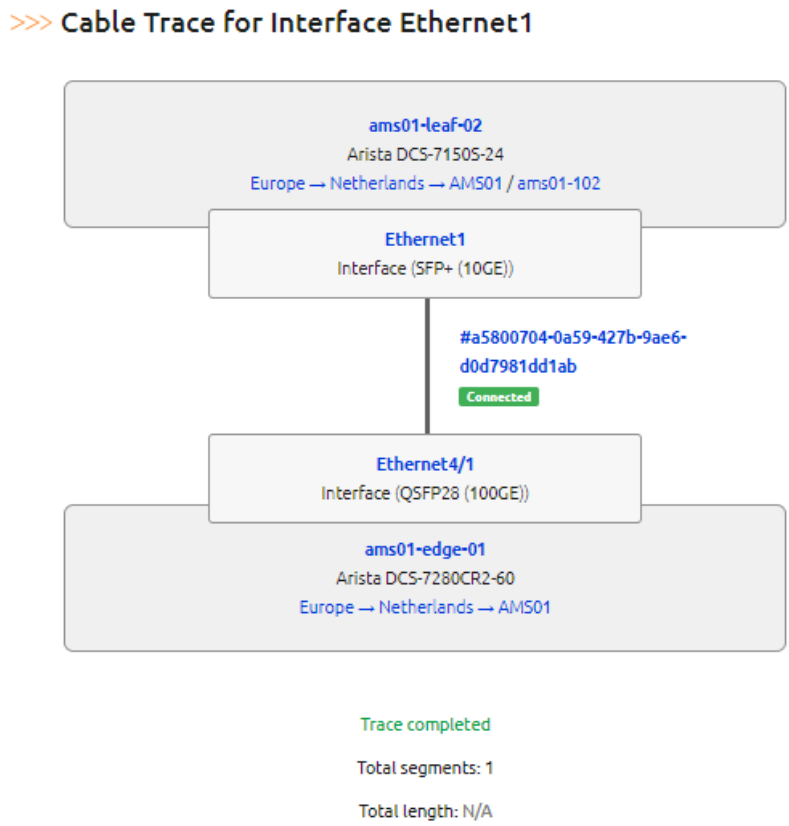

Figure 2.7 - Visual of a cable trace in Nautobot

This function follows the path of the connected cable from one termination point to another. If a cable connects to a pass-through port and there's another cable connected to the peer port, Nautobot continues tracing the path until it reaches a non-pass-through or unconnected termination point. This tracing capability is crucial for mapping out the physical path of connectivity across a network, aiding in troubleshooting and network documentation. An interesting aspect of cable tracing is its ability to trace through circuits. For instance, if a cable path includes a circuit, the tracing will show the connection from a device interface to the circuit's termination points, providing a clear view of how different network elements are interconnected through physical cabling. This feature enhances the understanding of network topology and the role of each physical connection within it.

The following diagram shows an example of modeling a cable plant that includes patch panels.

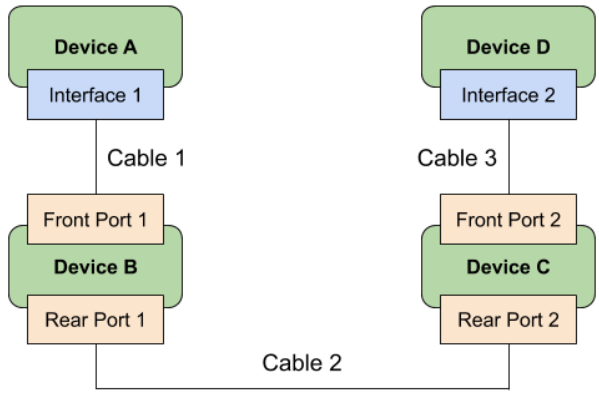

Figure 2.8 – Visualizing device connectivity with a patch panel (Device B and C) in use

Device A is connected to Device D through two patch panels, B and C. The rear ports of B and C represent the riser cable between the two panels.

This next example shows a cable path trace across a circuit, connecting two Device Interfaces.

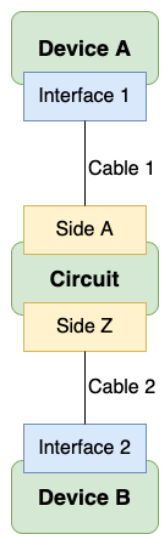

Figure 2.9 – Cable path trace across a circuit that connects two device interfaces

Device bays

Device bays diverge from the norm we have just discussed with other components. They are used as the basis to model hardware chassis-based devices. In this way, we create an instance of a device that represents the chassis itself and then create bays within that chassis that individually accept other child devices. It is very important to consider the specific set of use cases that are intended with this model.

Device bays adopt the ability to create parent/child relationships between child devices and the chassis that houses them, but the chassis is intended to be "dumb" in this model. That is, the intent is more to model blade servers in which the chassis provides housing, power, and connectivity, but has no other relation to the child blades installed in the bays, or vice versa.

This means in the networking world, device bays would not typically be the best way to model a chassis-based switch or router that contains several line cards. The litmus test for this distinction is to ask whether the chassis device has a single management IP from which you configure and control the entire device across all line cards. If you do have a case where you are managing line cards independently of one another (thereby logically managing each line card as its own device), device bays might be an acceptable means of modeling such devices. But we typically find that to be rare. Instead, it would be more appropriate to create the chassis network device as normal, but create all of the interfaces across the line cards as discrete interfaces on the chassis (named accordingly) and track the line cards as Inventory Items on the device. There are ways to bulk-rename interfaces if you need to move line cards around. Chassis devices are also not meant to model distinct network devices with a shared control plane, like a Cisco StackWise switch. The virtual chassis model is suited for that purpose and will be discussed later in this chapter.

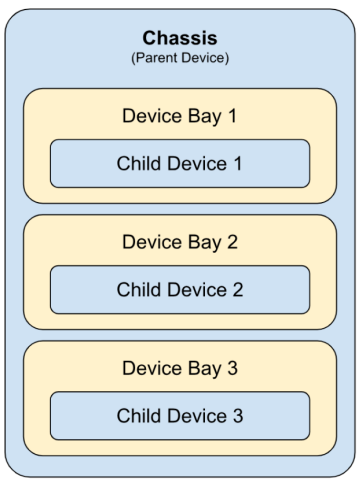

Figure 2.10 - Usage of device bays to model a chassis-based device and its relationship to child devices, such as blade compute servers

Inventory items

Speaking of inventory items models, they are a way to associate *any* other type of component to an individual instance of a device for tracking. Normally, we would think of things such as hard drives, CPUs, PCI cards, and so on—basically, anything ancillary to the device itself that you want to track for asset inventory purposes. As you might imagine, we have the option to attribute a manufacturer, part ID, serial number, and asset tag to Inventory Items. Inventory items can also have their own parent/child relationships, which helps in tracking things such as optical transceivers in line cards. In the near future, Nautobot will allow more direct modeling on device modules, such as chassis line cards and their direct relationships to device interfaces. In doing so, the Inventory Item data model will evolve to allow hierarchical relationships and more meaningful tracking of ancillary device components.

Using device components in Nautobot

In order to add or manage device components in the Nautobot UI, you add interfaces, device bays, and any other component within or under a device as shown here:

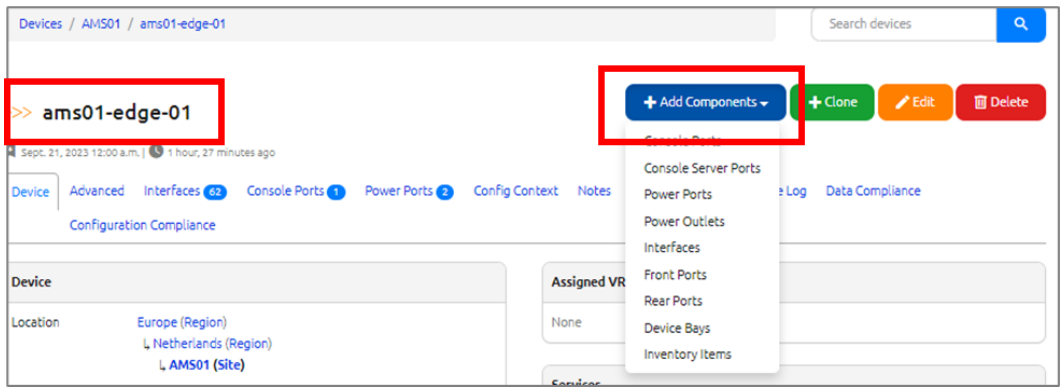

Figure 2.11 – Add device components for a single device

However, you may be thinking that you want to apply those components across all similar devices and device types. That is possible by also adding components at the device-type level as shown here:

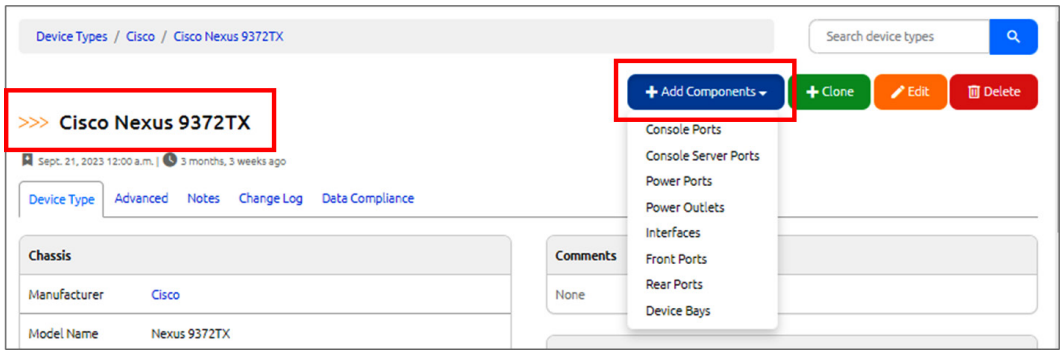

Figure 2.12 – Add device components for a device type

Now we'll dive deeper into device types.

Device types

Device types are closely related to devices, in that they represent the hardware model, or type, of the device. But in Nautobot this manifests in modeling several aspects of the hardware model. You have self-explanatory attributes such as the manufacturer and model number, but the real power of device types comes from the tracking of component templates. Basically, we take all of the port-based component models, plus the device bays previously discussed, and create simplified versions of them called **component templates**. You can see them at the bottom of device-type details pages as shown in the following screenshot.

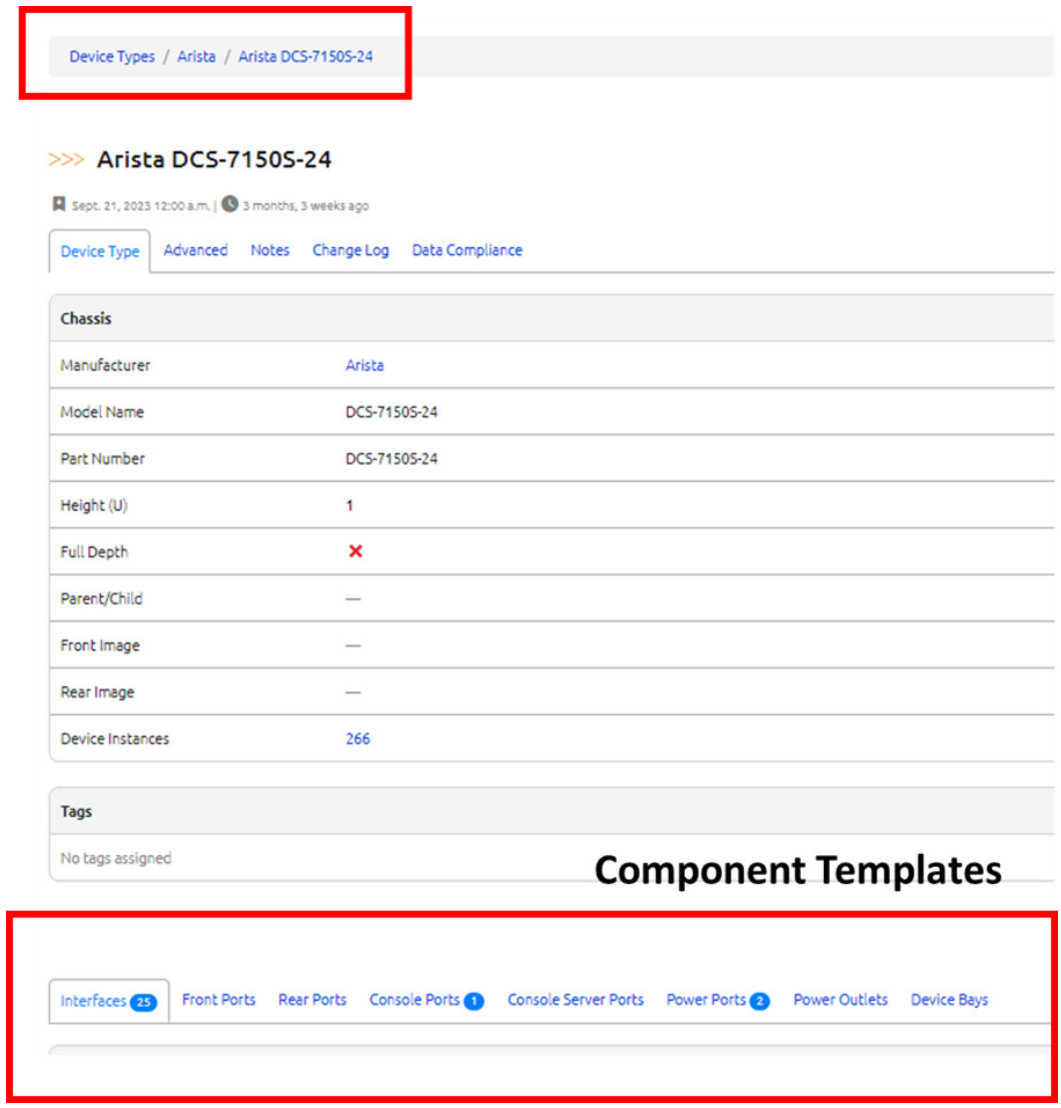

Figure 2.13 - Components templates for a device type

Don't worry, you'll have plenty of time to navigate the UI in the next few chapters.

The idea is that we create a representation of a device as that particular hardware is shipped to you from the manufacturer. It is important to note that a device type is specifically void of any deployment type logic or anything that would be used to distinguish two devices of the same type. In this way, they are true templates of the devices that we instantiate, or rather deploy, in our network. For instance, if we use Juniper EX3400-48P switches in our network, we would create a device type for the model number,

and then add 48 interfaces to that template. When we have several device types, which is certainly common in most networks, we end up with a library of types to choose from when creating devices; and yes, they can be imported from shareable definitions. Because we have defined templates, when you do create an actual instance of a device, you are asked which device type it uses. This causes all of the template components to be copied into the new device that gets created, effectively jump-starting the definition of that device in Nautobot.

Manufacturer

The manufacturer of a device is tracked as an attribute of the device type. The model itself is very simple, namely tracking just the name of the entity, such as Cisco or Palo Alto Networks. But having a separate entity in the data model allows for more complex use cases, as we will later discuss with elements such as custom fields and relationships.

You might ask why the *manufacturer* is only an attribute of a device type and not also, or exclusively, of the device. The answer lies in understanding data normalization, which is an advanced topic of data modeling and ultimately out of the scope of our discussion. But we point it out to say that great care has gone into the design of the core data model in Nautobot. In this case, the normalization is explained by pointing out that a device is always associated with a device type. And since a device type carries the understanding of the manufacturer for that type of hardware model, we can then infer the manufacturer for a given device without having to track it directly on the device model.

Roles and statuses

Now that we have described how to use the device model to create an inventory of devices, what can we do with it? One of the most basic questions we often need to answer in our network inventory is, *"What does this device do on the network?"* The role model is the primary mechanism to express that in the Nautobot data model. The role model itself is simple, but again, you will see the power of model extension later. Out of the box you get to specify the name of the role, a description, and a color to visually distinguish roles in the web UI. This means that the definition of roles is entirely up to you as a user. Your organization might create roles such as `switch`, `router`, or `firewall`, or more complex roles such as `dmz-edge-peer` – the point is that it is up to you. Once roles have been defined, you assign them to devices and begin consuming that added context in your SoT and automation endeavors.

Likewise, statuses reflect the administrative state of devices or interfaces. Just as with roles, you can create organizational-significant status values, or can use the ones provided out of the box, such as **Active**, **Offline**, or **Staged**. Both the role and status models in Nautobot are actually generic in nature and are used across several other use cases such as IPAM and circuit tracking. This gives platform administrators a central place to manage this type of metadata, and reduce duplication where the same values might be relevant across multiple uses. Similar to roles, statuses are covered in more detail in *Chapter 6*.

Platform

The platform model is an example in the core data model of something abstract that can be implemented to suit your needs, or simply ignored. This model is commonly used to specify the software family or even the specific version that a device is running. This is of course done by defining a platform instance and then optionally relating it to a device. The platform model also has special attributes related to Nautobot's built-in NAPALM integration feature set and these are used to tell NAPALM what driver and configuration options to use for a given device connection. More on that later.

Virtual chassis

Virtual chassis is a device deployment–specific model that allows you to track switch stacks or instances of multiple devices that share a common control plane, such as the Cisco StackWise or Juniper virtual chassis products. In Nautobot, you create a virtual chassis by grouping two or more member devices, specifying a master device from that group, and assigning membership priority values. Then, when dealing with components of the virtual chassis, we have an aggregate of all components (like interfaces) across all members. Virtual chassis are covered in more detail in *Chapter 5*.

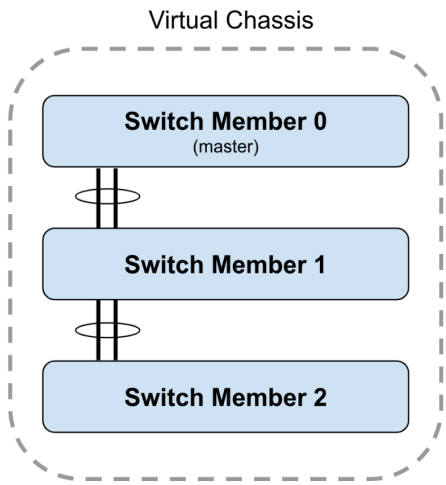

Figure 2.14 – A virtual chassis depicting a three-member switch stack (such as Cisco StackWise or Juniper VC, and their backplane connectivity)

Device redundancy groups

Related in nature but serving a different set of networking use cases is the device redundancy group model. The intent here is to model **High Availability** (**HA**) topologies involving separate control planes. Examples of this are two routers participating in an ECMP topology or perhaps a set of firewalls with some form of proprietary failover mechanism. We are tracking clusters of devices that have some form of HA, so an individual device may only be a member of one group at a time, and

carries a membership priority value for that group. The group itself designates an HA strategy, either active/active or active/passive.

The following diagram shows one example use of device redundancy groups wherein pairs of routers are participating in a redundant ECMP topology.

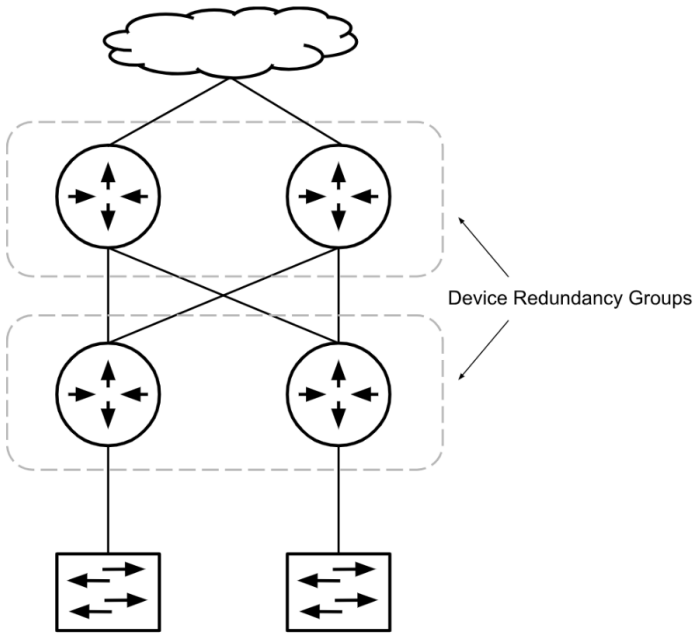

Figure 2.15 – Redundant ECMP topology using device redundancy groups

Note that device redundancy groups are covered in much more detail in *Chapter 5*.

Interface redundancy groups

Interface redundancy groups are similar in concept, but are designed to group interfaces that share a single virtual address, typically used in redundancy protocols such as HSRP or VRRP to provide fault-tolerant default gateways in networks. These groups must be created prior to assigning interfaces to them, ensuring proper setup and configuration. While their primary design is for first-hop redundancy protocols, they are versatile enough to represent any grouping of redundant interfaces, regardless of addressing or not. Adding interfaces to these groups requires setting a priority value for each interface, dependent on the redundancy protocol used, such as a range of 1 to 255 for HSRP. Optional features include associating an IP address with the group to act as the virtual address, specifying the redundancy protocol (such as HSRP, VRRP, GLBP, or CARP), using secrets groups to store sensitive information such as authentication keys, and including a protocol group ID, which can be either an integer or a text label up to 50 characters long. Like device redundancy groups, the flexibility in the

model means you can use the feature in a number of ways. For example, it is not uncommon to use interface redundancy groups to express circuit redundancy at a site by terminating circuits to interfaces participating in a group.

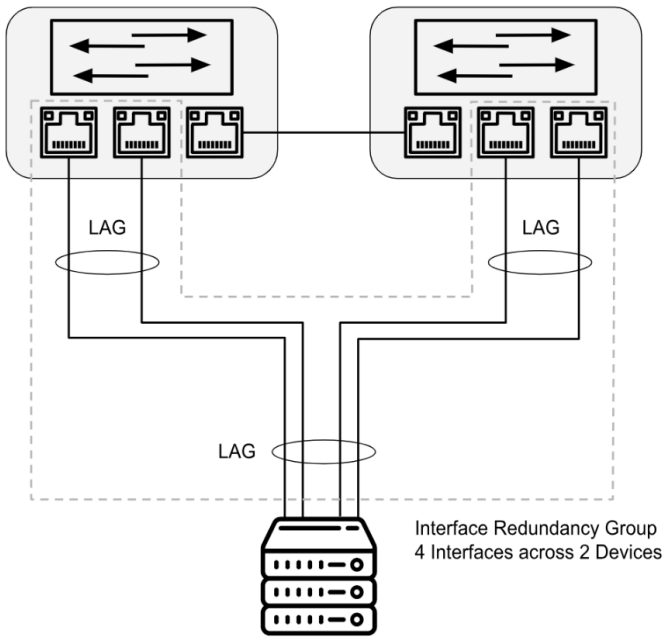

Figure 2.16 – Two-switch MC-LAG topology

The diagram shows a two-switch MC-LAG topology, where an interface redundancy group is used to track the aggregate of interface members across devices and is where config such as virtual Layer-3 addresses live.

Racks

Racks are another cornerstone of tracking a physical device plant. In Nautobot, a rack specifies its position within a location and its dimensions, in terms of width and total number of units that can be populated. It is not required to make use of racks at all in Nautobot, as evidenced by the data model relationship between devices and racks being options. Here is an example of viewing an example of a rack in Nautobot:

Figure 2.17 – Front and rear rack views shown on a detailed rack view

If you do wish to "rack" a device, you specify the relevant attributes on the Device instance. Those are the rack to assign the device to, the base (bottommost) used to install the device inside the rack, and the face (front or back) of the rack. The height of the device (measured in whole rack units) is specified in the device type. You have the option of reserving space in a rack, which will block devices from occupying such space at the data model level.

Going beyond just viewing racks in Nautobot, you can also populate a floor plan with racks to view an end-to-end layout of a given room, as can be seen in the following screenshot:

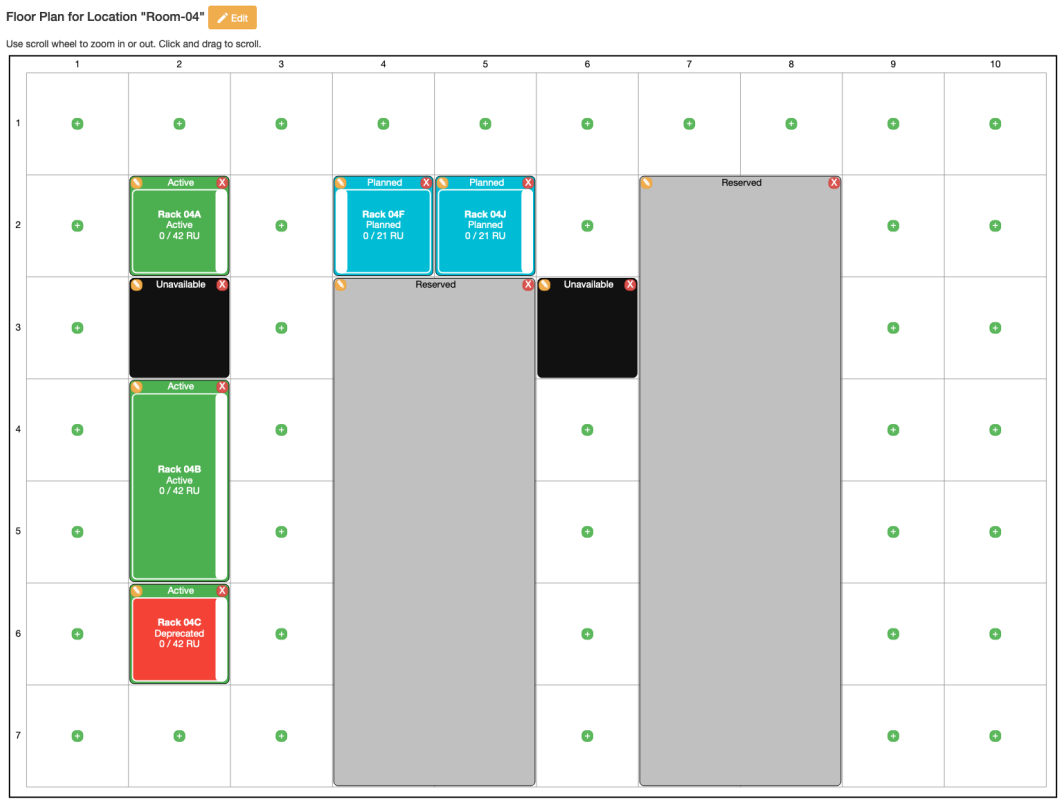

Figure 2.18 – Data center floor plan view with the Nautobot floor plan app

In order to place and view racks on a floor plan, you need to use the Nautobot floor plan app.

Rack groups

Rack groups allow you to build a nestable hierarchy of rack groups that in practice are used to represent entities such as network closets, floors in a building, rows and cages in a data center, and so on. Their use is optional, but a rack may be assigned to only one group.

Locations

Locations are an example of an organizational model that is ancillary to the network data model but still very useful in aggregating assets and other aspects of our network data. Broadly speaking, locations are any entity in which you might place network assets or associate some context, such as a prefix or VLAN. The location model is very flexible, allowing you to define a hierarchy that suits your needs. An example might be countries divided by business regions, or you might create states in the USA. The Location model is also great for modeling more fine-grained physical entities such as the

buildings on a campus, floors in those buildings, or even network closets on those floors. Locations are not limited to just the physical world, though; you can also use them to designate logical areas in your network. Locations have a name, a linkage to their parent (within the user-defined hierarchy), and a set of metadata fields, such as contact name, email, phone number, address, and so on. A few models in Nautobot require assignment to a location, including devices and racks. Several others allow the optional association to a location, including VLANs, and in some cases (like VLANs) certain feature sets are augmented based on the assignment of a location.

Location type

The location type model is the underpinning of the user-definable hierarchy. You define a set of types and how they relate to one another in terms of the parent/child tiering. So, from our earlier example, a hierarchy of **Country | City | Building** would be represented as three location types with the relevant parent relationships set. Location types also designate what other types of objects may relate to a given location. For example, if you wanted to ensure that all devices are assigned to a building, and not simply a country, the location type is the place to define that constraint. The following screenshot shows an example that nests three location types for a higher-education use case in a university. It includes a **Campus | Building | IDF** hierarchy, and here it is shown in the Nautobot UI:

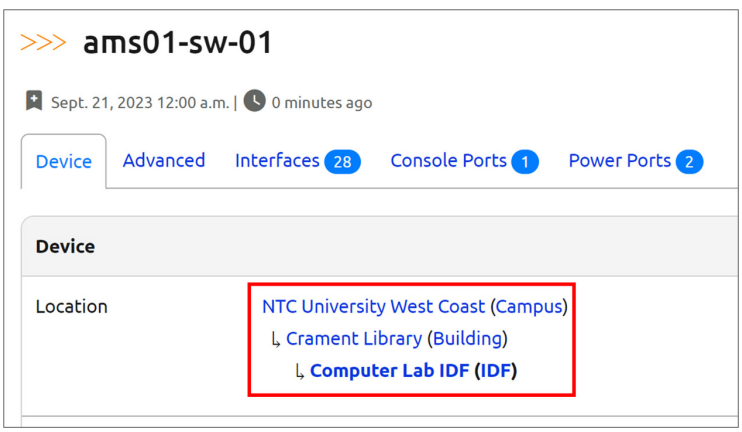

Figure 2.19 – Viewing user-defined location types in use for a single device

These location types are completely configurable to make sure you and your user base understand the data in your vocabulary.

Tenants

In Nautobot, tenants represent distinct groupings of resources, commonly used for administrative segmentation within an organization. They are typically utilized to symbolize individual customers or internal departments. A wide range of objects within Nautobot can be assigned to tenants, including locations, racks, rack reservations, devices, VRFs, prefixes, IP addresses, VLANs, circuits, clusters, and virtual machines. This assignment is crucial for indicating the ownership or association of a specific object with a particular tenant, thereby organizing the network resources efficiently.

For instance, if a rack is exclusively serving a specific customer, it would be assigned to the tenant instance representing that customer. In a service provider scenario, a container network may be split into many child subnets, each of which is assigned to a particular tenant that relates to a customer environment.

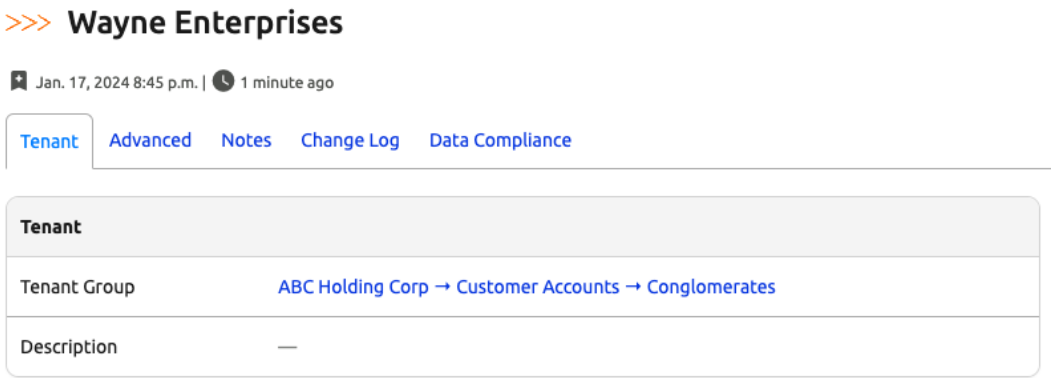

Figure 2.20 – Viewing a tenant definition inside of a user-defined tenant group hierarchy

Additionally, tenants can be grouped together for better organization. Custom groups such as "Customers" and "Departments" can be created, with the assignment of tenants to these groups being optional. This grouping allows for more structured and understandable organization within Nautobot, especially in scenarios where a clear distinction between different operational or customer segments is necessary.

Furthermore, Nautobot supports recursive nesting within tenant groups, enabling the creation of a multi-level hierarchy. For example, under a broader "Customers" group, there could be subgroups for individual tenants categorized by specific products or account teams. This hierarchical approach to organizing tenants provides a flexible and scalable way to manage and represent various entities and their relationships within an organization's network infrastructure.

IPAM data models

The next major area of the network data model is for tracking **IP Address Management** (**IPAM**) data. This includes IP addresses and subnets (called *prefixes* in Nautobot), but also other related models such as VRFs, route targets, and VLANs.

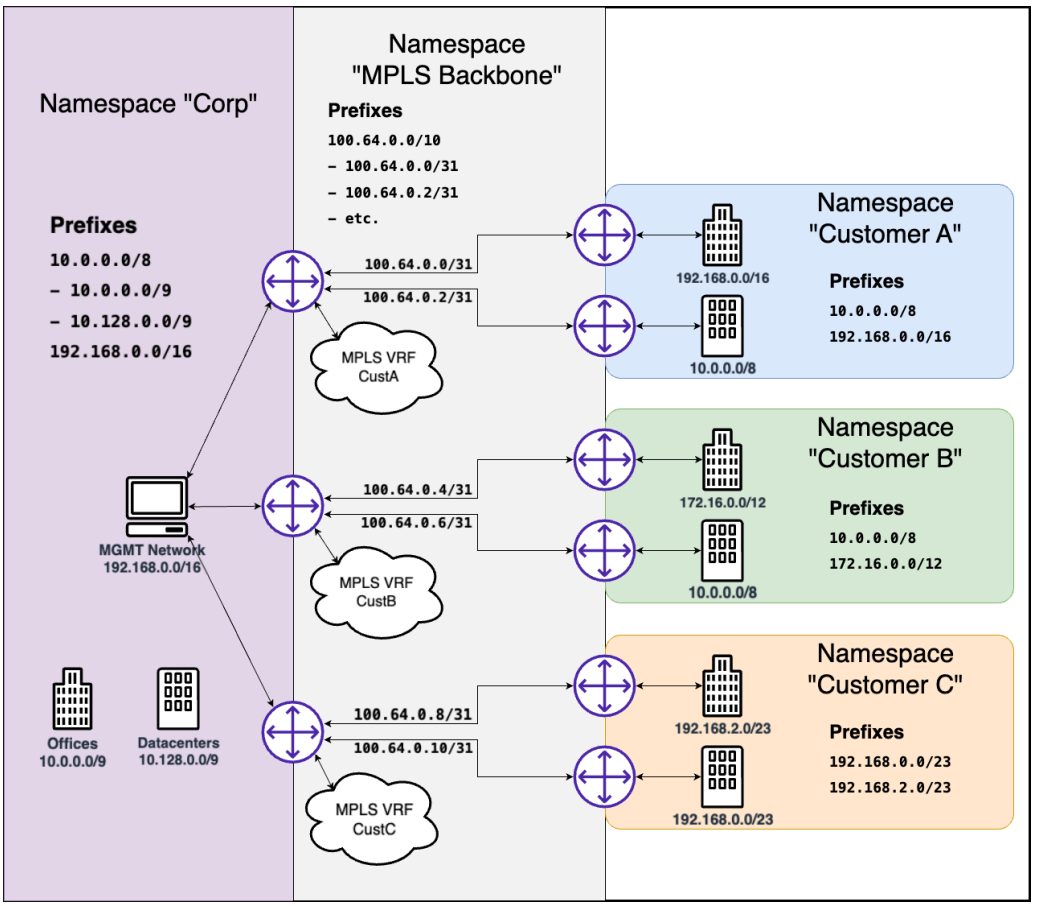

Figure 2.21 – Various aspects of the Nautobot IPAM data model

The preceding diagram describes the various aspects of the Nautobot IPAM data model. Take note of the manner in which namespaces, devices, prefixes, and VRFs all play a part. In particular, this diagram shows the uses of overlapping address space and duplicate network deployment scenarios.

Namespaces

In the IPAM domain in Nautobot, namespaces are integral for grouping and managing VRFs, prefixes, and IP addresses. They act as boundaries or constraints for ensuring uniqueness and avoiding duplication within IPAM data, where use of overlapping address space occurs. The single, default namespace might suffice for simpler networks without overlapping prefixes or duplicate IP addresses, even with thousands of IPAM records. However, in more complex scenarios, such as those of managed service providers or large enterprises, multiple namespaces become essential to accurately model the network and differentiate between records that might appear duplicate.

Each namespace in Nautobot is identified by a name and a description, and optionally, it can be linked to a specific location for informational purposes. Within any given namespace, there must be only one record for each distinct VRF, prefix, or IP address. While a single record, such as a VRF or a virtual IP address, may be utilized in various parts of the network—such as being configured on multiple devices or assigned to multiple interfaces—it is treated as a singular network entity in these instances, aligning with Nautobot's data-modeling approach.

Nautobot's implementation of namespaces particularly addresses scenarios where what appears to be the same VRF, prefix, or IP address might actually represent distinct entities within a network. This is especially relevant in situations such as corporate mergers, where overlapping network spaces from different entities might need to coexist in parallel namespaces rather than being merged into a single namespace. Another example could be an MSP that deploys network segments in an identical fashion for each of its customers. This functionality ensures accurate and distinct representations of network components in complex or evolving network environments.

Prefixes

A prefix represents an IPv4 or IPv6 network defined by a network address and mask in CIDR notation, such as `192.0.2.0/24`. It includes only the network portion of an IP address, meaning all bits outside the mask must be zero, except for `/32` IPv4 and `/128` IPv6 prefixes, which can represent specific IP addresses. Each prefix is unique within a specified namespace and can optionally be linked to a location or associated with one or more VRF instances. Those not assigned to a VRF are deemed part of the global VRF within their namespace.

Prefixes have assigned statuses to indicate their operational state. Default statuses include **Active** for in-use prefixes, **Reserved** for future use, and **Deprecated** for those no longer in use. Additionally, prefixes can have an optional role, which is customizable and represents their function, such as distinguishing between production and development environments. A prefix may also be linked to a VLAN, aiding in correlating address spaces with Layer-2 domains. VLANs can have multiple prefixes associated with them. Prefixes can be assigned to an **Regional Internet Registry** (**RIR**) for tracking authorization to use certain public IP spaces. The `date_allocated` field is available to mark the allocation date of a prefix, whether by an RIR or for internal assignment.

The prefix model includes a `type` field with three options: *Container*, *Network* (default), and *Pool*. A *Pool* type indicates that every IP within the range is assignable, while *Network* type assumes the first and last IP addresses in an IPv4 prefix are unusable. Nautobot organizes prefixes and IP addresses into a hierarchy using the `parent` field. A *Container* type prefix should only have a *Container* parent; a *Network* type should have a *Container* parent; and a *Pool* type should have a *Network* parent. Any prefix can be a root prefix, meaning it has no parent. This hierarchical structure aids in managing and understanding the relationships between different network segments.

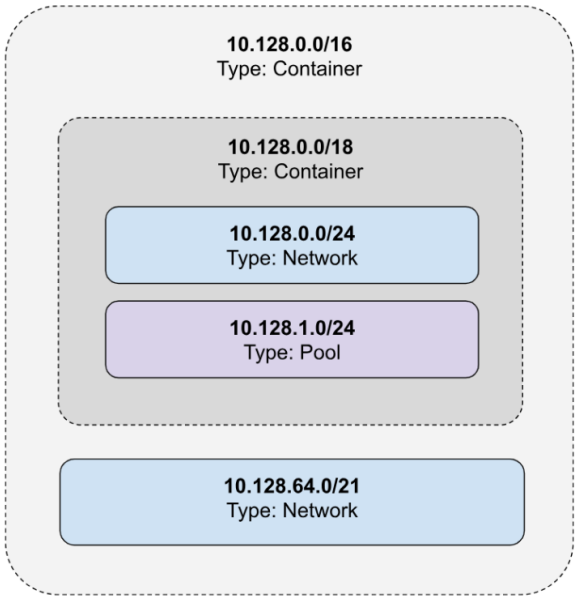

Figure 2.22 – IPAM prefix hierarchy showing how /16 could be carved up using Container, Network, and Pool prefix types

IP addresses

An IP address is its own model and is defined as a single host address, either IPv4 or IPv6, along with its subnet mask, mirroring its real-world configuration on an interface. These IP addresses are automatically organized under parent prefixes in accordance with the IP hierarchy. They do not have direct assignments to namespaces or VRFs; instead, they inherit these attributes from their parent prefix.

Each IP address in Nautobot can be assigned an operational status and a functional role. The default statuses are the same as those provided for prefixes. Functional roles, which are conceptual and not customizable, include options such as 'Loopback', 'Secondary', 'Anycast', 'VIP', and various roles for redundancy protocols such as VRRP, HSRP, and GLBP. These roles help to indicate special attributes or uses of an IP address. IP addresses can also be classified by types indicating specific functions, such as 'Host', 'DHCP', or 'SLAAC'. The default type is "host". This classification aids in identifying the specific function or behavior of an IP address within the network.

An IP address in Nautobot can be linked to interfaces of devices or virtual machines, and an interface may have multiple IP addresses assigned to it. Furthermore, each device or virtual machine can designate one of its interface IPs as its primary IP for each address family (IPv4 and IPv6). Nautobot also supports the designation of IP addresses as inside addresses for **Network Address Translation** (**NAT**). This feature is useful for denoting translations between public and private IP spaces, and the relationship is maintained bidirectionally.

With regard to data modeling, the IP address model has a direct relationship to its parent prefix, and this relationship aids in many areas, including the performance of working with deeply nested address space.

RIRs

RIRs are recognized as authorities responsible for allocating globally routable IP address space. The primary RIRs are ARIN (North America), RIPE (Europe, the Middle East, and parts of Central Asia), APNIC (Asia-Pacific region), LACNIC (Latin America and the Caribbean), and AFRINIC (Africa). In addition to these, Nautobot also treats certain RFCs, such as RFCs 1918 and 6598 that define address spaces for internal use, as equivalent to RIRs. This categorization acknowledges these RFCs as authorities owning specific address ranges. Nautobot provides flexibility in managing RIRs. Users can create custom RIRs and assign prefixes to them as needed. The RIR model in Nautobot includes a Boolean flag to indicate whether the RIR is designated for private IP space allocation only.

For practical application, consider an organization that has been allocated a specific IP range, such as `7.128.0.0/16`, by ARIN. This organization also uses internal addressing as defined by RFC 1918. In Nautobot, the organization would create two RIRs, one named "ARIN" for the publicly routable space and another named "RFC 1918" for internal addressing. Subsequently, prefixes corresponding to these address spaces would be created and assigned to their respective RIRs. This approach allows for organized management of IP spaces, both public and private, ensuring clear documentation and tracking of address allocations within the network.

VRFs

In Nautobot, a **Virtual Routing and Forwarding** (**VRF**) object is used to represent a VRF domain, which functions as an isolated routing table. VRFs are instrumental in segmenting networks, commonly used for isolating different customers or organizations within a network, or for managing overlapping IP address spaces, such as multiple instances of the `10.0.0.0/8` space.

Each VRF is given a name and a route distinguisher, which must be unique within the namespace to which the VRF is assigned. VRFs in Nautobot can be associated with specific tenants, aiding in organizing and managing the IP space according to customer or internal user groups. This association is particularly useful for service providers or large organizations managing multiple customer networks. Additionally, VRFs can have import and export route targets. These are used in Layer-3 VPNs (L3VPNs) to control the exchange of routes (or prefixes) among different VRFs, facilitating the selective sharing of routing information across different segments of the network.

In terms of IP address management, prefixes and their contained IP addresses can be assigned to one or more VRFs within their namespace. This flexibility allows for the alignment of IP address usage with the specific requirements of different parts of the network. Any prefix or IP address not assigned to a specific VRF is considered part of the implied "global" VRF within its namespace. It is important to note that this "global" VRF is distinct from any other "global" namespace, which might contain several different VRFs.

Route targets

A route target is used as an extended BGP community for controlling route redistribution among VRF tables, especially in L3VPNs. Each route target is assigned a unique name following the format prescribed by RFC 4364, similar to VRF route distinguishers. In Nautobot, route targets are linked to individual VRFs as either import or export targets to accurately model route exchange in an L3VPN. Additionally, route targets can be optionally associated with a tenant and *tagged*, providing further organizational capabilities within the network.

> **Note**
> For more advanced BGP modeling capabilities, see the Nautobot BGP app.

VLANs and VLAN groups

VLANs are used to represent isolated Layer-2 domains, as defined by IEEE 802.1Q. Each VLAN is identified by a unique name and a numeric ID that ranges from 1 to 4094. VLANs in Nautobot can be assigned to a location, a tenant, or a VLAN group. Each VLAN is required to have an assigned status and, like prefixes, can also be assigned a functional role.

VLANs have relationships to prefixes and device interfaces. Prefixes, of course, track which Layer-3 networks reside on the VLAN. Nautobot also allows proper modeling of 802.1Q by way of tracking tagged and untagged VLANs on individual interfaces. While one might expect a relationship directly between VLAN and device, the interface tagging is more meaningful to the device configuration and through understanding of the overall network data model, we are able to derive all VLANs relevant to a device, through its interfaces.

VLAN groups in Nautobot serve as a means to organize VLANs. Each VLAN group can be optionally linked to a specific location. VLAN groups are particularly useful for enforcing uniqueness among VLANs: within a group, each VLAN must have a unique ID and name. In terms of network management, this is helpful in distinguishing discrete VLANs when repeated in various network segments/topologies that are designed to look the same way, such as branch offices.

Please also note that VLANs not assigned to any group can have overlapping names and IDs, even if they belong to the same location. For instance, it's possible to have two VLANs with the ID 123, but they cannot be part of the same VLAN group.

Circuits data models

The circuit domain is the other large area of the network model. Here you can track your providers' details and certainly your inventory of circuits, but also how those circuits connect to the network. There are also abstractions that allow you to model your provider networks, which you do not control but that play a role in connectivity.

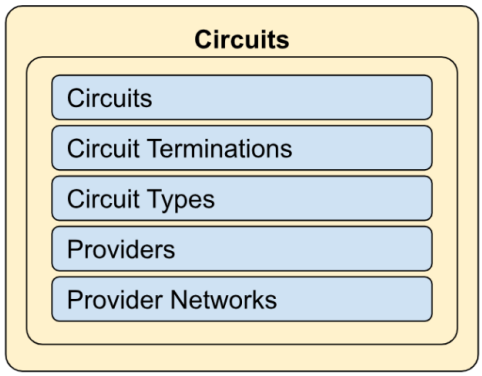

Figure 2.23 – Overview of the circuits data model provided in Nautobot

Circuits

Circuits in Nautobot play a crucial role in representing the physical connectivity within a network, much like the device types and devices do for hardware. At its core, a circuit in Nautobot is defined as a communications link that connects exactly two endpoints, known as A and Z terminations. This model allows for a flexible representation of a circuit's connection points; it's not uncommon to define only one termination, especially in cases where the details of the provider side, such as in internet access circuits, are not a primary concern. However, for more complex setups, such as private network connections linking customer locations, both terminations are typically modeled to accurately reflect the network's physical layout.

Every instance of a circuit in Nautobot is linked to a provider. This relationship is akin to how devices are linked to device types. Additionally, each circuit is categorized by a user-defined type, allowing for a detailed and customized classification of the network's various connections. For instance, a network might utilize internet access circuits from one provider and private MPLS circuits from another, each distinctly identified by their type. Circuits are primarily identified by the combination of their provider and unique circuit ID. Nautobot also introduces a robust status system for circuits, with default statuses encompassing the entire lifecycle of a circuit—from **Planned** and **Provisioning** to **Active**, **Offline**, **Deprovisioning**, and eventually **Decommissioned**. These statuses are also fully customizable, with more detail later in the chapter. Circuits in Nautobot can be enriched with several optional fields. These fields include the *installation date* and *commit rate*, adding layers of detail similar to how device types track attributes such as manufacturer and model number. Additionally, just as devices can be attributed to a location or tenant, circuits may also be assigned to Nautobot tenants, providing a clear demarcation of responsibility and ownership within the network's infrastructure.

Circuit terminations

A circuit termination in Nautobot is essentially the point where a circuit connects to a specific location or device, capturing the physical reality of network connectivity. A circuit in Nautobot can have up to two terminations, labeled as A and Z, echoing the common practice in networking of identifying the two ends of a link. The flexibility of having either one or two terminations allows for various use cases: a single-termination circuit is apt for scenarios where the far end of the circuit is unknown or irrelevant, such as an internet access circuit connecting to a transit provider. On the other hand, a dual-termination circuit is instrumental in tracking circuits that link two specific locations, mirroring the physical connection between them.

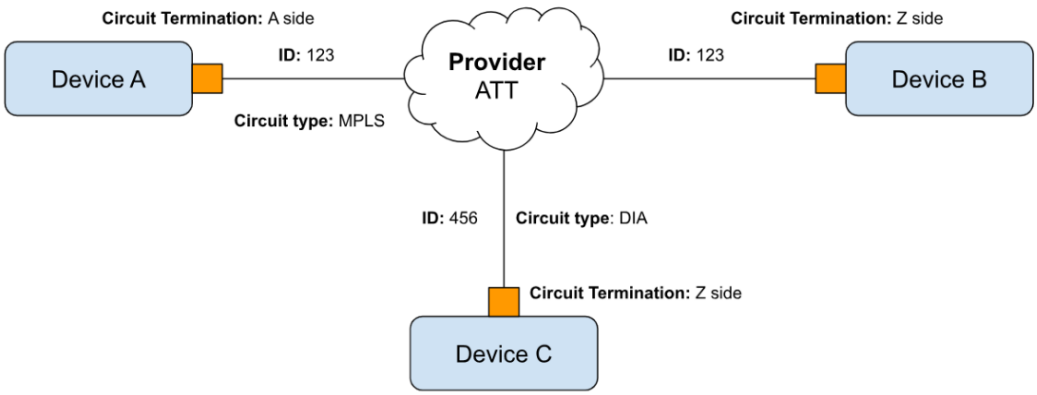

Figure 2.24 – A diagram showing various circuits terminated to devices

Each termination of a circuit is associated with either a specific location or a provider network. When linked to a location, a termination may be further detailed by connecting it via a cable to a particular device interface or port within that location, much like how devices are connected to the network. This level of granularity allows for precise tracking and management of network connections. Furthermore, each circuit termination is required to have an assigned port speed, mirroring the operational parameters of the network. Additionally, it can optionally have an upstream speed defined, especially relevant in scenarios where downstream and upstream speeds differ, such as with DOCSIS cable modems. This mirrors the attention to detail seen in other aspects of Nautobot's data model, such as the specific attributes of devices and device types.

In line with Nautobot's philosophy of closely modeling the real-world configurations, a circuit is restricted to connecting only to physical interfaces. This means circuits cannot terminate at virtual interfaces, such as LAG interfaces. In scenarios where a circuit connects to a LAG, each physical member of the LAG must have its separate circuit, and each one must be modeled discretely. This approach ensures that the Nautobot model stays true to the actual physical layout and functioning of the network, ensuring accuracy and clarity in network documentation and management.

Circuit types

Circuit types in Nautobot are entirely customizable, offering a high degree of flexibility to adapt to the specific needs and terminologies of different network environments. The primary purpose of defining circuit types is to convey the nature of the service being delivered over a particular circuit. By categorizing circuits based on their function, network administrators can easily understand and manage the various types of connectivity within their infrastructure. This is especially important in complex networks where different types of circuits serve distinct roles.

Examples of commonly defined circuit types in Nautobot include the following:

- Internet Transit
- Out-of-Band Connectivity
- Peering
- Private Backhaul

Circuit providers

A circuit provider is defined as any entity that facilitates connectivity, whether between different locations or within a single location in Nautobot. This broad definition encompasses a variety of entities, not limited to traditional carriers, but also including internet exchange points, and even organizations that are direct peering partners. The role of a circuit provider is fundamental in the configuration of circuits in Nautobot.

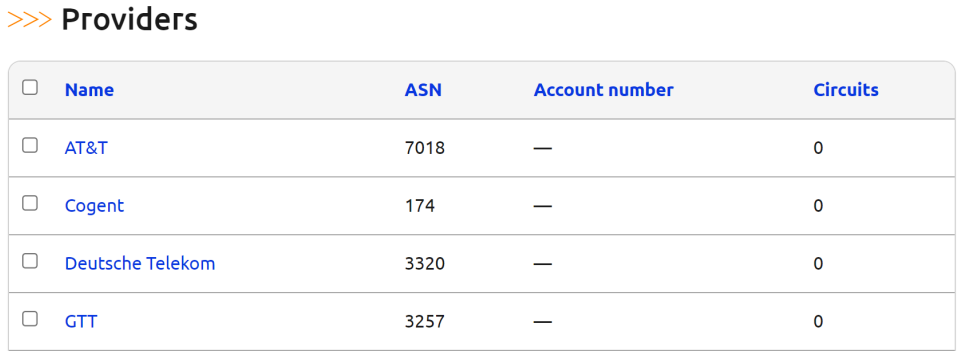

Figure 2.25 – Circuit Providers table (list) view

Each circuit must be linked to a specific provider, ensuring that there's clear documentation of the entity responsible for the connectivity service. Providers can be detailed with additional attributes, enhancing the depth of information available for network management, such as ASNs, customer account numbers, or contact information.