O'REILLY®
Technical Guide

Practical Guide to Modern Networking Telemetry

How Telemetry Can Be Used to See Into Your Network's Performance and Usage Patterns

Early Release
RAW & UNEDITED

Compliments of kentik®

Avi Freedman & Leon Adato

kentik®

The Network Intelligence Platform

Take the hard work out of running your network.

UNDERSTAND EVERY:

- ✓ Cloud
- ✓ Network
- ✓ Container

BOOK A DEMO TODAY

The world's best infrastructure teams trust Kentik:

zoom · M · box · Canva · servicenow · Dropbox

www.kentik.com

Practical Guide To Modern Networking Telemetry

How Telemetry Can Be Used to See Into Your Network's Performance and Usage Patterns

Avi Freedman and Leon Adato

O'REILLY®

Practical Guide To Modern Networking Telemetry
by Avi Freedman and Leon Adato

Copyright © 2025 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Megan Laddusaw
Development Editor: Gary O'Brien
Production Editor: Katherine Tozer
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

August 2025: First Edition

Revision History for the Early Release

2025-03-26:	First Release
2025-07-31:	Second Release

See *http://oreilly.com/catalog/errata.csp?isbn=9798341608900* for release details.

979-8-341-60888-7

[FILL IN]

Table of Contents

Brief Table of Contents (*Not Yet Final*)

Introduction

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the Introduction of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

About Modern Network Telemetry

Right at the outset, you may be asking yourself, "What's so important about Network Telemetry? What can it really do to make my work or network measurably better? This is not only an understandable question, it's a pretty common one that we hear. A lot. After all, the most important work done by IT practitioners is designing, implementing, and maintaining systems and architectures. Much of monitoring and observability needs to be set up in advance, with the benefit often coming into play only after something has gone wrong (and sometimes only after it's gone catastrophically wrong!).

So before diving into the tools, techniques, and technologies, we'll take a minute to talk about "the network" and step back and discuss the benefits and return on investment that building robust observability options into your infrastructure can bring.

What Is “the network,” exactly?

Back in 2023, in his book “The Ultimate Guide to Network Observability”, Avi wrote,

> “People often refer to ‘the network’ in their organizations, but in most cases the network isn’t one entity. It’s a complex, diverse, fragmented, and loosely interconnected set of physical and virtual links and equipment, and it’s housed in a variety of places, including data centers, corporate wide area networks (WANs), private and public clouds, the internet, container environments, and even inside hosts. Organizations own and control some of those resources but simply pay to use others.”

Since that time, things remain largely the same. Sure, the cloud has gotten more cloudy (meaning opaque to users) even as it has become more ubiquitous within organizations large and small. And at the same time corporate LAN and WAN environments have not, by and large, become less complex (or sprawling, or expensive). In fact, the opposite has been true. Even as company’s investment in cloud has increased, so too has their investment in on-premises networking gone up.

What Is “network telemetry,” exactly?

Network telemetry refers to the data *FROM* and *ABOUT* your network, both from your network elements and from watching data moving through your network.

That includes simple things like device types, basic hardware metrics, and so on. But it also extends to status and performance information about the data moving around the network: source, destination, protocol, and more.

All of this makes network telemetry important (and valuable) for several reasons:

Infrastructure Performance and Health - Uptime, Health, Planning
: In order for user and application traffic to flow well across the network, the devices themselves need to be functioning well,

with enough CPU and RAM capacity, healthy hardware and optics, and links that work and aren't too full.

Watching logs, metrics, traffic flow, configurations, and other network device telemetry (plus a bit of active performance testing) is critical to running a great network.

Network and Application Performance
: All of the traffic for users and applications becomes packets flowing across the network, and the network is a great source of truth for shining light on both the net "composed" performance of the system, and even into applications that are having performance problems.

Cost
: At current infrastructure scale, cost can be enormous for many companies, and optimizing the network infrastructure often is a full time job - or more. Combining network telemetry with business data about cost can drive huge savings that often fund the entire network observability stack.

Life Cycle Automation Support
: Everyone's eager to move their application to the cloud, but is it really performing better there? Network telemetry helps you see the before, during, and after changes from more minor to whole migrations so you can be certain you're getting the improvement you think you're paying for - and didn't break anything!

Security
: Network telemetry remains a fast and great way to identify most cybersecurity issues, such as DDoS attempts, compromise and lateral movement, and the impact of botnets.

About You ("Is this book for me?")

This book is for you if any of the following things are (or might be) true:

- You'd describe yourself as a "learn and do" kind of person.
- You are comfortable with application monitoring and observability, but not networking, and you'd like to find out how

network monitoring and observability are different (and beneficial!)

- You are comfortable with networking, but not monitoring and observability, and you'd like to find out how network *monitoring and observability are different* (and beneficial!)
- You build, maintain, support, or are simply curious about "the network" and the ways in which network performance impacts everything that rides on top of it, from the data to the application to the overall user experience.
- You know how to look at a dashboard and interpret data presented in charts and graphs, but you want to understand how monitoring and observability data are represented in those forms.

On the flip side, what does this book presume you already know? To be honest, there's not a lot of requirements. Throughout this guide, we'll not only provide detailed information on terms and technologies, we'll point you to external content when we think some readers might appreciate a deeper dive than we have pages to cover.

That said, you will be most comfortable with the information we're sharing if the following things are generally true about you:

- You're familiar with the basic network devices - routers, switches, and firewalls - and what they do.
- You have a general understanding of cloud infrastructure concepts like virtual machines and cloud providers.
- You're aware of typical network security issues and threats, like DDoS attacks

We'll If you aren't rock-solid on those topics, DO NOT PANIC (also, don't put this book back on the shelf. We're not done paying off our kids' orthodontist yet.). Throughout this guide we'll offer information, instruction, and examples. And if you need more, we'll also provide links to background and deeper dives on these and other topics as we cover them.

About This Book: What Will I Learn?

We know that a book of this nature is an investment of time and attention. As such we wanted to suggest the return you may enjoy for spending some of your precious time here. By the end of this book the reader will gain a better understanding about:

Types of network telemetry
: Including traffic data, device metrics, events, synthetic measurements, routing information, configuration data, and business/operational data.

Network telemetry sources
: Including physical and virtual network equipment, servers, clients, cloud environments, and more.

The different planes
: (Management, control, data) that serve as points of contact where network telemetry can be gathered..

Ways to wrangle telemetry data
: Such as collecting monitoring information from devices, replicating data to analytics systems, feeding broader observability systems, and understanding data system requirements and trade-offs.

Ways to use network telemetry in situ
: From guided exploration (e.g., starting from a network map and zooming in), to unbounded exploration (e.g., drilling down on specific aspects of network traffic), through using telemetry as part of workflows, issue responses, and automation. Basically how you navigate and display and use network telemetry within the tool(s) you use to collect it.

Using network telemetry as part of your larger observability ecosystem
: Network data extends, enhances, and informs the telemetry you get from application and infrastructure monitoring. In this section we show you how to integrate and correlate the information so you have a better sense of what is happening from the top to the very bottom of the application stack and the OSI model.

But this guide isn't just geared to increasing your awareness. We'd also like to believe that we'll provide you with skills you can actively apply. Therefore, after reading this book we also hope the reader will be able to:

- Justify (both to colleagues and decision makers) the business case for implementing and using a network observability solution in the workplace.
- Apply the knowledge about the different types of network telemetry, along with each type's strengths and deficits, in order to select the best mix of options when displaying data about a particular network, application, issue, or architecture.
- Design better monitoring and observability solutions by combining data and telemetry from various tools (be honest, we know you've got more than a couple) in ways that provide clarity and uncover issues.
- Build, adapt, and improve monitoring and observability outputs - everything from dashboards and reports to alerts to automated workflows - based on your (perhaps newfound) understanding of how network telemetry works.
- Lead the charge for better network observability by educating team-mates, departments, and even business leaders.

NOT About This Book: What WON'T I Learn?

Well-organized technologists don't just focus on the list of things they want to do, they also maintain healthy boundaries by keeping in mind the things that are NOT part of the roadmap. We'd like to show the same discipline and organizational rigor here by listing some of the things this book is NOT going to teach you about. We hope this will both set your mind at ease and also let you know whether the book you're holding has the answers you're looking for or not.

This book isn't going to teach you about things like:

- Basic networking. We won't explain in detail the OSI model, networking protocols, or how to configure a routing protocol on a layer 3 device (but will provide links to background on routers)
- How to evaluate, select, install, configure, or use a specific observability solution.

- How to evaluate, select, install, configure, or use a specific type of networking gear; or networking protocols; or standard networking architectures.
- How to evaluate, select, install, configure, or use a specific cloud provider; or how to migrate a particular application to (or from, or between) the cloud.

CHAPTER 1

Network and Telemetry Introduction

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

What IS Network Telemetry, Redux

We offered a high-level description earlier in the book, but now it's time to really dig in and offer a detailed explanation: *network telemetry* refers to data ABOUT your network, rather than the data that's moving AROUND your network. Network telemetry includes everything from the relatively simple (and relatively unimportant) device information (think: what are the make, model, and sub-components in the gear that make up my network); to the more relevant (and important) performance and state information about the devices themselves (e.g. metrics related to CPU, RAM, disk, bandwidth, number of connections, and so on).

But where the concept of network telemetry really hits hard is when it tells you detailed information about the network traffic itself. Where are those packets coming from or going to - which might include everything from IP addresses to URLs to countries? How much of the data is one protocol vs. another? How many of the intermediate hops from source to destination are remaining static, and how many are changing? And those intermediate hops - are they the most efficient hops, or is some of the traffic getting hung up in sub-optimal routes?

Those questions are (or should be) deeply interesting to IT practitioners because they speak to aspects of application performance that can't be sussed out using higher level tracing options found in non-network-centric observability solutions.

This makes network telemetry important (and valuable) for several reasons:

Security First

Saying "security first" is about 12 astronomical units (1.1 billion miles, or over 70 trillion average-lengthed bananas) away from actually *doing* something about security. Even saying "security is everyone's responsibility" is pointless (not to mention insulting) because sure, it is, but how many of us have ever earned a quarterly bonus because we "did good security"? Yeah, that's what I thought.

Nevertheless, if a technique allowed you to not only become more aware of security issues, but identify their root causes and even address them, you probably wouldn't turn your nose up at it, either.

Network telemetry allows you to do exactly that with specific types of security issues. First and foremost there are DDoS and botnet attacks. Not only does network telemetry tell you it's happening to (or on) your network, the right tools can show you when those events are happening "near" your network - meaning they're happening to someone else who is also using your ISP's infrastructure, using same routes as your data is traversing or hitting intermediate network gear your traffic is also using.

But it goes further than that. Because it can tell you both source and destination, as well as the volume, protocol breakdown, and applications involved, network telemetry can quickly identify when data is being exfiltrated. Even if no data is involved, having a network observability solution will allow you to put alerts in place that tell you when unexpected or unwanted protocols, ports, or destinations appear in the mix, or appear above a certain baseline.

Understand your network top to bottom and end to end

Network telemetry gives you a view of your internal network (LAN, whether in the cloud or on premises) but it also tells you how your network traffic is performing once it exits the edge router and hits the WAN. That means you can make informed decisions about traffic and capacity management at the edge of your network. It's also the best way to understand how your precious (and expensive) investment in everything from your Internet provider(s) to your cloud architecture to your SD-WAN is performing.

Used either as-is, or in combination with those higher-level application observability tools I mentioned above, network telemetry provides a unique view of your infrastructure, allowing you to pinpoint the area that's actually having a problem (versus just knowing that "the application is slow" or "we can't get to the database"), and thus speed resolution.

All of this allows you to confidently understand your current use (including new and un-expected services and traffic patterns), plan for future growth, and thus control costs.

Make your cloud environment less… foggy?

Smooth the transition of applications from on-premises to cloud by allowing you to first baseline the current state in terms of performance, load, traffic direction, etc. And then, once it's been transitioned to the cloud, you'll understand when performance differs, and where the breakdown is occurring, and why.

In essence, network telemetry empowers you to gain a deeper understanding of your network's health, performance, and usage patterns. This knowledge allows you to proactively manage your network, optimize its performance, and ensure a smooth digital experience for users.

Anatomy of a Network

Let's be honest - networks are composed of simple base components but at scale *anything* but simple. Sure, most of the network diagrams you see in class are 3 routers (inevitably named "Spring", "Summer", and "Fall"), connected to a switch, which is connected to "the cloud".

And yet, in the real world, networks are composed of many (MANY!) different device types in multiple configurations and use cases.

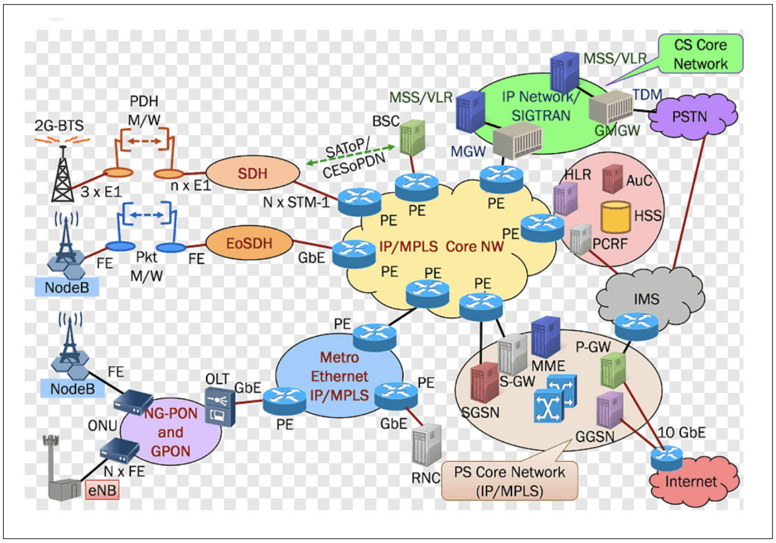

Figure 1-1. Title placeholder

So we wanted to take a moment to identify and define the most common devices and functions you might see, as these are going to be the sources of network telemetry that we discuss later in the book.

Broadly categorized, these include:

Routers, switches, and access points
: Physical and virtual equipment responsible for moving your traffic in your data centers and clouds; across your campuses and wide-area networks; and over the internet and broadband and mobile networks.

Servers, clients, and IoT endpoints
: Physical and virtual equipment that connects to your network, such as servers, clients like PCs, laptops, and mobile devices, and even internet-of-things (IoT) endpoints.

Cloud VPCs
: Virtual private clouds in your public cloud infrastructure. Includes subnets and the container environments where you deploy your microservices-enabled applications.

Controllers, service meshes, load balancers, and firewalls
: Physical and virtual devices that control, orchestrate, load balance, program network configuration, and filter inbound and outbound network and application traffic based on policies and threat intelligence across data centers, cloud, WANs, and the internet.

Transport devices
: Many modern transport devices (layers 1 and 2 in the OSI Model[1] in fiber, broadband, and mobile networks now support active and passive telemetry.

TAP/SPAN/NPB devices
: Physical and virtual test access points (TAP), switch port analyzers (SPAN), and network packet brokers (NPB) that provide port mirroring, testing, and monitoring.

But a few of those deserve a more detailed description. Once again, the purpose of this guide is not to teach you every aspect of network design, architecture, implementation or management. Instead, we want to describe the devices below in terms of the telemetry they emit and the insights that telemetry provides.

It's also important to note at the outset that each of these devices has their own hardware metrics that shed light on the network - everything from

- up/down status of individual components
- to CPU and RAM to fan, temperature, and power supply data

1 For those who need a reminder, the layers are: Physical, Data Link Layer, Network, Transport, Session, Presentation, Application. A popular mnemonic for this is: "Please Do Not Throw Sausage Pizza Away". For those who need more than a reminder, there's always wikipedia (LINK TO: *https://en.wikipedia.org/wiki/OSI_model*)

- to disk activity and configuration changes

To a greater or lesser extent, combining that insight with the other details can tell you where problems are occurring or, conversely, when a problem is actually downstream of a device which seems to be complaining but is in actuality simply unable to communicate with the next hop in the chain.

The final caveat before diving into the specifics of each device type is that even something as seemingly innocuous as inventory (especially when visualized as a map) can have a profound impact on your ability to understand how a network is performing and where the root cause of a problem may lie.

Network Routing/Switching Primitives

There are four foundational primitives of networking that most networking elements use in various combinations for IP networking:

Link-Layer Forwarding (usually Ethernet)
: Most of this guide will focus on IP networking - watching IPv4 and IPv6 traffic. But underneath the IP layer there are link layers that think in "frames" - not "packets". Those frames are usually forwarded by learning dynamically what addresses (called MAC addresses for Ethernet) are at various places. There used to be more link-layer protocols for wired networking, but Ethernet has dominated for decades, perhaps with a smattering of Infiniband for supercomping and AI data centers.

IP Forwarding (also called "routing")
: Taking packets from one interface (physical or logical) to another. Routing tables are populated that describe how packets get from one place to another. Usually it's done mostly by looking at the destination IP address of the packet, but it can get more complex and look at the source IP address or other parts of the packet.

Access Control Lists / Firewall Rules
: Another core routing primitive allows filtering and rate-limiting traffic according to specified policies. These policies are often called Access Control Lists, or ACLs, in routers and switches; Firewall Rules in security elements; and sometimes just "policy" in more cloud-y networking layers.

Tunnels / VPNs

Originally used for more "exotic" configurations, tunnels are now commonplace and are protocol-based wormholes that connect different parts of a network together. Common tunneling protocols include GRE, IP (in) IP, and Wireguard. When these are exposed to users often they're just called VPNs, but people raised in networking often think of them as tunnels.

Network Device "Layers"

(Caveate: we're being really broad in our use of the word "device". We could say "element" but that's even more vague and prone to overlap with other technical elements… see what we mean? In any case, for our purposes the word "device" might mean a physical or virtual object which receives, processes, and/or emits data.)

Network devices forward packets or frames but how do you configure those devices, and how do the rules get loaded and the forwarding executed?

In the old world packets ran through the CPU but at current scales there just aren't general-purpose computers fast enough for that for many core network devices, so accelerated forwarding hardware is used.

Vendors and practitioners generally talk about multiple "planes" we interact with in network devices. With regard to monitoring and observability, a *plane* is really an abstract concept that broadly refers to a distinct *layer* of the network architecture where a specific process takes place.

Most commonly (in networking) you'll read about the management plane, the control plane and the data plane.

The *management plane* is concerned with device-specific information and tasks, such as the configuration of the device and its subcomponents. This is also the part of the architecture responsible for updates to firmware and operating system, security features, and (most importantly to us), monitoring.

The *control plane* is the part of the architecture responsible for how data (packets) are moved from one place to another - forwarding, refusing access, etc. Thus, the act of building a routing table is part of the control pane's functionality.

The *data plane* is the area of the network architecture that actually *does* the forwarding. Thus, the data plane takes a packet and sends it out a specific interface, based on the routing map assembled by the control plane. It's also the data plane that performs tasks like modifying a packet with additional header information, or applying a Quality of Service (QoS) rule.

Each of these planes has different kinds of telemetry we can consume or pull.

For example, CPU utilization from the management plane; table utilization from the control plane; and traffic speeds, summaries, and even detailed records of traffic sent/received from the data plane.

We'll map these in more detail in this module.

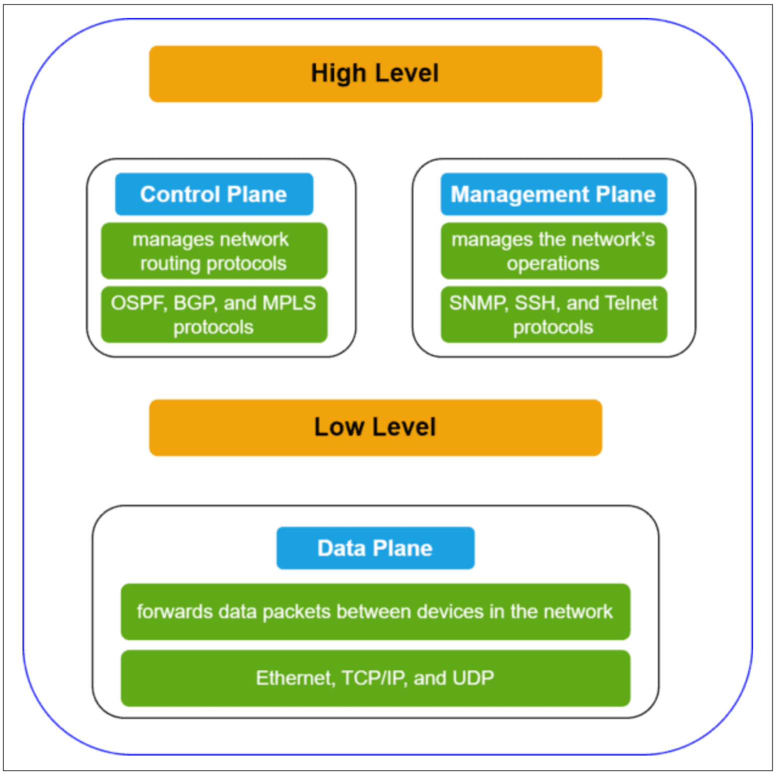
High Level
Control Plane
manages network routing protocols
OSPF, BGP, and MPLS protocols
Management Plane
manages the network's operations
SNMP, SSH, and Telnet protocols
Low Level
Data Plane
forwards data packets between devices in the network
Ethernet, TCP/IP, and UDP

Figure 1-2. Title placeholder

Device Types

Routers

Routers are hardware that generally run a Unix-based OS that interacts with users and other networking elements, and instructs specialized hardware (if present) how to forward, filter, and report on the packets going through.

Routers have interfaces - physical or logical - and the physical interfaces usually have optics or wired ports that can be monitored.

Many modern routers can do switch-like Layer 2 forwarding themselves, but generally, (unlike a switch), a router segregates Layer 2 forwarding unless told to do otherwise via configuration.

Layer 2 Switches

Layer 2 switches move (typically) Ethernet frames around, but also have OSes, CLI, protocols, tables, and telemetry they generate.

Layer 3 switches

Most switches today are very close to being routers, and do IP forwarding as well at line rate.

Virtual Private Clouds

In cloud computing and cloud networking, Virtual Private Clouds, or VPCs, are primitives that behave like router interfaces.

Web Logs

Web servers can emit log lines per transaction that describe various actions and transactions, both success and failure. These logs can shed a great deal of light on the network since they show source and destination IP addresses, application context, and performance information about both application and TCP-layer performance.

Load balancers

With regard to the devices we're exploring in this section, load balancers may be the most fundamentally different as well as functionally narrow-focused. They're really Layer 4+ routers in some sense, and the metrics and telemetry you should look for from them bears at least a passing resemblance to that of routers and switches.

Unlike routers, since they are part of the Layer 4+ transactions, they actually see and can report on response time, latency, throughput, error information, and even the traffic patterns of the data being handled. They can be great observation points but are often left out of network telemetry sources by teams.

Service Meshes

Broadly speaking, *service meshes* are load balancers designed to talk to other backend software elements, not browsers/users. They can do health checks, load balancing, content rewriting, policy enforcement, and telemetry just like load balancers, and are almost always delivered as a software layer or service, unlike load balancers which are sometimes still physical appliances.

Firewalls

When discussing routers, we mentioned that there are security controls that might be involved. Firewalls take that behavior and make it their entire raison d'etre. At the same time, there are still routing elements involved.

Hosts, Containers, and Kubernetes

No, these are not (usually, but see below) network devices. Yes, they still matter - even in the context of "network observability".

First and foremost, hosts, containers, etc are usually the ultimate and final end-points between which all network traffic flows. But second, they often have networking elements - real, honest-to-goodness, route-and-switch-type-data - contained within them, and thus they are (yet another) amazing and often-overlooked source of insight and telemetry.syslog

Other host types - from servers to containers and beyond may also be (openly or secretly) acting as routers. All it takes is two network adapters (physical or virtual) connected to two separate network subnets, and - voila! - network routing is happening.

In a modern way hosts get combined to run workloads, Kubernetes (K8s) is a hot topic!

K8s has *namespaces* that are just collections of containers that perform various tasks - some in service of the K8s orchestration functions itself, and more (most) related to the application which forms the raison d'etre for the K8s pod in the first place.

But inevitably, the K8s namespace will include a system running system primitives like forwarding tables and traffic filters, and most interestingly nowadays, Container Native Interfaces, or CNIs, that are the "routing" controller for the underlying systems.

Getting network telemetry on hosts can be done broadly the "traditional" way - running an SNMP daemon to expose interface, CPU, and other metrics, as well as exporting traffic summaries via a flow-exporting daemon.

There is also a newer way of observing traffic and metrics on hosts via eBPF, which was developed as a modern kernel instrumentation infrastructure that can observe and control what the Linux kernel is doing. Generally, eBPF agents that look at the network are TCP and application-decode focused and can provide enriched flow-like summaries with performance and application context - but are often architected to ignore lower layers, so may miss errors or attacks at lower protocol layers.

Tap/SPAN/Network Packet Brokers (NPB) devices

When network devices themselves can't process or emit traffic (flow-like) telemetry of their own, one solution is to use network taps (optical splitters), SPAN (Switch Port Analyzer) functionality on switches, or Network Packet Brokers that are "super SPAN" devices with more complex filtering and policy. Many of these devices just copy packets to the things that will generate telemetry, but some of them can generate flow summaries themselves.

Common Telemetry Types: The Four Pillars

If you've been anywhere near a conversation on social media about "observability", you will inevitably hear someone mention the three four pillars of observability (really, telemetry). These have been the subject of great debate, passionate argument, internecine flame wars, and (most of all) confusion.

At the heart, the goal was (and still is) an attempt to both list and then group the essential techniques that make up observability, such that customers could have a single reference by which they could evaluate the vendors and their products in this space.

Of course, it could be argued that the reason for some of the disagreements boils down to vendors not accepting a definition which would put their own products into a less-than-glowing category.

Arguments aside, the pillars seem to have settled themselves into some mix of "M.E.L.T.":

Metrics
: Individual (usually numerical) data points which can be gathered, graphed, averaged, etc to show a trend.

Events
: Strings of text and numbers stamped with a time and a source to show that such-and-such occurred to so-and-so system and thus-and-this time.

Logs
: Sources of messages and other output which might be aggregated across many systems and comprise multiple layers of the architecture, from low-level hardware to high level application.

Traces
: A coherent collection of information that show how a "transaction" within an application traverses multiple systems, and the ways the transaction is performed at every step along that path - usually augmented with a transaction or trade ID to allow correlation of all of the steps in that transaction.

These categorizations are, by and large, fine. However they are (and always have been) biased toward application telemetry. And that's fine, but it doesn't work completely when discussing network telemetry.

Therefore, Avi and I are presenting a different framework for understanding the different types of data that you'll commonly encounter when monitoring a network infrastructure.

Types of NETWORK Telemetry

Admittedly, monitoring and observability data comprise a really long list of a wide range of data types. A quick glance at the next section will show that more than a few aren't even relevant to network devices (looking at you, PerfMon counters). So let's take a moment to describe the essential telemetry types that will make up the bulk of our focus in this book:

Traffic
: This describes any data element (metric, trace, etc) that shows how your traffic is flowing across networks. Sample formats include NetFlow, sFlow, IPFIX, VPC flow logs, traffic data in JSON, and packet data via PCAP and eBPF for connections, process, and container context in cloud-native environments.

Device metrics
: These tell you the state or health of your physical and logical network equipment. Sample formats include SNMP, syslog, and streaming telemetry.

Events
: This indicates events like an attempted login, a threshold has been met, or a configuration has been changed. Sample formats include SNMP trap and syslog.

Tables
: Snapshots/state of the various tables in a router, mostly for forwarding/routing.

Synthetic
: These "synthetic" measurements reveal performance metrics such as latency, packet loss, and jitter, and can be triggered or collected via device telemetry interfaces. They span client and server endpoints, network equipment, and internet-wide locations at both the network and application layers.

Configuration
: This (typically static) data represents the operating intent for all configurable network elements such as topology information, IP addresses, access control lists, location data, and even device details such as hardware and software versions. Sample formats include XML, YAML, and JSON files.

Business or operational
: Often called "layer 8," this data provides business, application, and operational context about what the network is being used, and can be added to telemetry to help network pros measure

impact, understand value of certain traffic, and prioritize their work.

DNS
: DNS telemetry helps put other network data into context by indicating from or to where traffic is coming or going. Most DNS information comes in text-based files.

Drill-Down: Telemetry Types

The previous list of types of telemetry is concise and focused rather than comprehensive. But now that we understand the definition and the value of network telemetry, as well as the devices that make up a typical network, we need to take a moment to list out all of the various data types available for monitoring and observability.

This section will go into both the protocols themselves and, in some cases, touch on the Device Health and Status: Syslog.

Syslog is a protocol which allows one machine to send a message ("log") to a server listening on TCP or UDP port 514. This is more often used and at higher volume when monitoring network and *nix (Unix, Linux) devices, but network and security devices such as firewalls and IDS/IPS systems send system and component logs - and can be configured to send even more detailed logs, though care is needed not to overwhelm the CPU (control plane).

Syslog messages are similar to SNMP traps, but differ in that syslog messages are relatively freeform and don't depend on the MIB-OID structure required by SNMP.

In addition to being more freeform, syslog tends to be "chattier" than SNMP traps. However, it's more flexible because many applications can be set up to send syslog messages, whereas SNMP traps are generally used much more sparingly, and most companies have much more broad and robust log collection ability and scale than they do for SNMP traps.

Traffic: Flow Monitoring (sFlow, NetFlow, VPC Flow Logs, eBPF, and others)

Standard device metrics can tell you the WAN interface on your router is passing 1.4Mbps of traffic. But who's using the traffic? What kind of data is being passed? Is it all HTTP, SSH, or something else?

You probably recognize the term “NetFlow”. It’s been around for a while. But what we’re referring to is really the broader category of traffic data, or “flow”, in general. Examples of these protocols that report on network traffic details include NetFlow, sFlow, JFlow, IPFIX, and VPC Flow Logs, among others. Despite the differences in strengths or weaknesses, implementation specifics, and more, they all have similar aims.

Flow monitoring answers those questions. It exports in terms of “conversations”—loosely defined as one period of data transfer between two computers using the same protocol. If DesktopComputer_123 is sending a file to Server_ABC via FTP, you may see a snippet or snippets of it via flow monitoring. Flow monitoring usually at least includes protocol, source and destination IP addresses and ports, and the number of bytes and packets observed. Note that if the flow records are sampled, the bytes and packets need to be multiplied back by the sample rate to form the approximate total traffic actually observed…

While people usually refer to both sFlow and the more connection-oriented NetFlow/IPFIX/VPC Flow Log protocols as flow monitoring, and once parsed, the data their records contain look pretty similar, the way they observe and report on traffic actually is quite different. sFlow exports headers from a sample of packets and is much easier to implement and takes less resources. It is also usually much more real-time. The traditional connection-oriented protocols need to build software or hardware tables of the connections (called flows), accumulate data like bytes and packets seen, and every so often pick records to export, usually according to a flow expiry timer.

eBPF-observed traffic can be a bit of a mix of those, and is done on hosts that run native processes, containers, and VMs. Most eBPF traffic exporters are concerned primarily with TCP and UDP connections that terminate on the kernel of the server they are running on. This enables adding a lot of context not possible from outside the compute layer - for example, process ID, command line arguments, process memory usage, sometimes application decodes, and if containerized, pod and namespace. On the other hand the traditional approaches often don’t watch all the packets and can miss ICMP, non-IP protocols, bad layer 2 framing, and especially, traffic in hosted VMs.

While some flow sources only support un-sampled export (typically, firewalls and many VPC Flow Log sources), generally flow protocols and their software and hardware implementations will require sampling. And even if they don't, you may want to enable sampling to reduce the amount of compute and storage you need to receive, process, store, and query the traffic data.

Flow data is often captured by a network device located somewhere in the middle of the conversation—usually one or more routers near a local edge or core of the network (or one router at each remote location if there are site-to-site communications which don't go through the core). In prior decades, many routers that looked like they supported flow export protocols actually had real problems with platform stability and/or flow accuracy, but most platforms now support it fairly well.

Note the two machines in the conversation (DesktopComputer_123 and Server_ABC, in my example) do NOT need to be monitored. Just the network devices that they are either attached to, or that might see their conversations..

Traffic and Policy: Packet Monitoring

For higher granularity monitoring than flow allows, packet monitoring is sometimes used, though much less than years ago. This is generally enabled with optical taps that split a fraction of light off to be observed; by asking switches or routers to copy packets (called now generically SPAN, or Switch Port Analyzer); or by using specialized switching devices called packet brokers to do this.

The packet copies generally go to servers that can generate flow, store the raw packets, and/or perform DPI (Deep Packet Inspection) to go even deeper than most flow protocols allow.

While instrumenting modern infrastructure with full packet monitoring can be incredibly expensive and is not usual in most greenfield builds, packet analysis can generate flow-like data with application decoding and performance data simply not available in most flow-based traffic monitoring implementations, and storing all of the packets allows for more detailed security inspection and investigation.

A Device-to-Telemetry Rosetta Stone

Having gone over the types of devices you might meet in your travels through the network; and the types of telemetry you might encounter in your observability solution, I thought we'd take a moment to identify which types of telemetry you will be able to coax out of various device types.

Table 1-1. Title placeholder

Device type	Telemetry Summary
Routers	Just as important is the information about the traffic itself - in this we're grouping the volume, errors, discards, packet counts, packet errors; as well as the sources, destinations, URLs, ports, and protocols involved. Along with the essential service of routing traffic, routers might also do a fair bit of security functions, especially if ACLs or other permission-based handling is involved. And all of those functions are going to carry with them their own sets of metrics, logs, and other telemetry which further informs your picture of the infrastructure.
Layer 2 Switches	Just like routers, switches provide a full complement of metrics under the broad umbrella called "bandwidth" - sources and destinations (this time in the form of MAC addresses rather than IPs or ports - along with bandwidth volume, errors, discards, packet counts, and packet errors.
Layer 3 Switches	From a network telemetry point of view, these devices combine everything you might find in a router and an L2 switch. Thus, the very thing that makes them more powerful in terms of network architecture, is what makes them more complex with regard to monitoring and observability.
Virtual Private Clouds	VPC flow logs are equivalent to the flow records (e.g., NetFlow, sFlow, etc.) in a traditional (on premises) network. Various cloud network components such as a VPC, a subnet, a network interface, internet gateway, or a transit gateway, can generate flow logs which are used in the same way as NetFlow data. These started with the core cloud network functionality (VPC, NSG, VNET) but have versions now available for many other cloud services like storage, firewalls, and other layers of the stack.
Load Balancers	On the other hand, there are some very specific data types, like connections (active, passive, etc), request count, information on healthy/unhealthy hosts, and even the health and performance of the load balancer infrastructure itself. If the load balancer is in a cloud environment, you might also be interested in request tracing, changes to the load balancer itself (due to elastic compute), and connectivity to other cloud-based elements like storage, content engines, and more. It's also important to note that "load balancer" is a general term for a device which might be specifically designed for application, network, or even gateways (virtual devices like firewalls, or intrusion detection systems) traffic.
Service Meshes	Like load balancers, service meshes see a rich set of data that can be incredibly useful to establish Mean Time to Innocence (MTI) and to debug full-stack issues ("is it the network?").

Device type	Telemetry Summary
Web Logs	Nowadays most web/application servers are behind load balancers or service meshes and those tend to be the observation points. But classically and still sometimes, getting a stream of web transaction logs can be a great way to see performance / latency of requests, whether they succeeded or failed, the source information of the browser that made the request (even if the packet was subsequently bounced off a VPN or other obfuscating device), and cookies and other data harvested by the web application itself. All of that, along with the telemetry from load balancers and routing information, can paint an incredibly clear picture of the "intent" and performance of the traffic on your network.
Firewalls	For the intrepid network telemetry spelunker, that means being prepared to gather all of the bandwidth information (volume, errors, etc) along with information about the device connections themselves - data about blocked connections, intrusion attempts, traffic patterns that have been pre-determined to be "suspicious", and network requests that violate built-in security policies. All of those will have a wide range of specific metrics and logs associated that shed light on who tried to do what, when, and to whom.
Hosts, Containers, and Kubernettes	In K8s-land (and let's be honest, there aren't many other serious contenders at the time of this writing), intra-container and inter-container network telemetry can be just as important to your understanding of performance, availability, fault, and reliability as traditional LAN and WAN insights. It can get a bit complicated because of tunnels and NAT, and from an observability perspective, using IPv6 un-routed space to disambiguate workloads and their IP addresses can be very helpful as a NAT alternative.
Tap/SPAN/ Network Packet Brokers (NPB) devices	In the early days of network devices, many couldn't generate their own traffic telemetry well or stably, but while today most can, they don't do much to observe traffic performance, and often really need to sample to keep up even now. Taps and NPBs can make it possible to send line-rate un-sampled traffic records, and/or send copies of the frames/packets to appliances that can do deeper inspection of traffic performance and contents than routers and switches are capable of doing.

Collecting data, network style

….and (broadly speaking) how to get that data out of your devices.

In the next module, we're going to show you how to "wrangle" your data - meaning take it from its initial form and make it usable in various observability tools. But before getting to that, we thought it would be helpful to offer an overview of how to get the data itself.

To be clear, this section will not offer comprehensive device- or vendor-specific instructions. It will mostly focus on how, generally speaking, to get the various telemetry types (SNMP, streaming telemetry, etc) out of devices, with a few illustrative vendor-specific examples. So more of "how to do an SNMP get" than "how to configure SNMP on a Cisco IOS switch versus doing it on a Juniper router."

Telemetry Deep Dive: SNMP and Streaming Telemetry

SNMP has been around for decades, and therefore folks interested in learning more about it will find a wealth of information, history, and tutorials at their disposal. For that reason, we're going to keep this section relatively brief, and trust the reader to find the specific instructions they might need..

SNMP can be broadly divided into two types of information, based on the delivery method - push-based or pull-based.

SNMP Traps

This is the pull-based option. They are triggered on the device being monitored, and sent to another device that is listening for those messages (a trap receiver or trap destination). Nothing is needed on the part of the monitoring solution except to receive and store those messages, and then correlate them with telemetry from other sources.

All the configuration for this is done on the device itself, meaning that every device on your network that needs to send traps has to be configured with the trap destination, along with the security elements (community string or username/password). We'll explain more about those options below.

SNMP Get

As hinted at earlier, SNMP Get requests are pull-based. A remote system sends a request to the machine being monitored, requesting one or more pieces of data. The most basic form of this command is:

```
snmpget -v <SNMP version> -c <community string> <machine IP or
name> <SNMP object>
```

What that looks like in practice:

```
~$ snmpget -v 2c -c public 192.168.1.10 1.3.6.1.2.1.1.5.0
iso.3.6.1.2.1.1.5.0  = STRING: "BRW9C305B289C1"
```

Rather than get into the weeds of the various methods of SNMP-Get (which will largely be handled under the hood by whatever monitoring and observability tool you use), the point is that SNMP works to bring data about a remote system (whether numeric or text) into a local repository for storage, tracking, and visualization

A Note about SNMP versions

Each version of SNMP brought enhancements aimed at addressing the needs and challenges of network management at the time.

SNMP version 1 (SNMPv1) uses community strings, which act as a rudimentary form of authentication.

SNMP version 2c enhances the performance aspects of SNMPv1, including GetBulk requests.

SNMP version 3 strongly emphasizes security and privacy with the inclusion of authentication, privacy (encryption), and access control. The drawback is that configuration of SNMP v3 is somewhat more complex both for each device to be monitored and for the monitoring solution that collects the data.

Consistency is the Key

Or it would be, if SNMP had more of it. It's not just that a particular data point (temperature, or CPU, or bandwidth) is inconsistently presented from one vendor to another.

It might not even be such a glaring issue if the difference existed between two different types of devices (routers and load balancers, for example).

Or between two models of the same device type from the same vendor.

But the fact that there are differences in the way the same data point is presented between two sub-elements (like interfaces) on the same device is simply egregious.

We aren't sharing this to downplay SNMP's continued importance in the landscape of network telemetry options, but rather to alert you that some mapping of different SNMP variables might be needed to get a unified idea of, for example, "interface traffic", even on devices from the same vendor.

Streaming Telemetry

As described in the "Types of Telemetry" section, streaming Telemetry (ST) is notable for having many of the same data points as SNMP, and more; and a far more consistent data structure; and a significantly higher granularity with a significantly lower impact on the device being monitored.

Originally developed as an alternative to SNMP, the goal was to move away from poll-based observation (and each monitoring system separately polling devices), and towards pushing defined data from network devices, to then be consumed by all of those systems.

So what's not to love? Well, getting it set up can be a bit of a challenge. This mostly arises from the newness of the technology and the lack of experience on the part of both engineers developing observability solutions; and network professionals who are implementing ST in their environments.

Oh, and the fact that it's not supported on a wide range of devices yet. That's another critical factor.

Telemetry Deep Dive: NetFlow, sFlow, and Other Traffic Sources

NetFlow (and its variations like sFlow, JFlow, IPFIX, and others, which we'll refer to from hereon out collectively as "NetFlow" unless we're discussing something particular about the other variations) was practically purpose-built with the goals of network telemetry and network observability in mind. While NetFlow isn't the only protocol a network engineer may need, it is almost certainly the primary one they will refer to for the richest level of insight.

NetFlow is push-based, meaning the device observing and generating the telemetry (kind of. More on that in a minute) sends data to a listening device.

As you can see, the single machine is able to report on conversations between devices on the internal network, the internet, and many points along the way such as peer routers.

Telemetry Deep Dive: API

Talking about APIs in the context of network telemetry is a slightly different discussion than the one you might have when describing

APIs for programming, automation, or retrieving cute cat pictures. Or Chuck Norris jokes. Or things that are slightly more useful like the time for sunrise/sunset.

An older way to gather metrics is still also sometimes required - CLI scraping via libraries that log into the device command line, issue commands, and retrieve and parse responses.

But why bother with APIs or CLI scraping?

APIs are attractive because they allow the process of telemetry collection to move into a slightly more modern paradigm, both for individuals and for monitoring solution engineers.

But both API and CLI scraping also allow access to some network device telemetry that is unavailable via SNMP or ST (Streaming Telemetry).

For example, many devices don't expose all the operating parameters of physical optics (temperatures and light levels) via SNMP and ST.

This adds yet another dimension to the unification required to consume all of these metrics, and as in our discussion of ST, is left as an exercise to the observability tools team and/or vendors.

Telemetry Deep Dive: Synthetic Transactions

If there is one drawback to all of the methods we've discussed so far, it's that it requires the active presence (and participation) of those pesky network elements called "users". Meaning: without people actively using an application, system, database, etc there's no (or at least significantly less) data moving through the network, and therefore fewer data points to collect, and therefore fewer opportunities to identify issues.

Also, most sources of network metrics and traffic telemetry don't have any concept of user or traffic performance (latency, loss, jitter, and throughput). So how best to see how well things are performing?

And this is where synthetic transactions come into play. As the name suggests, these are actions which generate traffic by means other than authentic user behavior.

Just for context, and maybe as the simplest example of this type of monitoring, both "ping" and "traceroute" could be categorized as synthetic transactions. However, synthetic transactions are, more broadly speaking, pre-set (you might think of them as "pre-recorded") actions which run at regular intervals. At its most simple, you can think of them as a macro running on your laptop every x minutes. But in reality the technique (and its benefits) go so much further than that.

First, the action can run from multiple locations, but report the results into a single repository. This provides insight as to whether an issue is happening with the target application or system (i.e. the problem is consistent from all locations at the same time) or if the app/system is fine, but certain routes to it are impacted.

Moreover, synthetic transactions can be multi-step. Rather than just checking if a web page "is up" (returns a 200 code), a single transaction can:

1. Go to a website
2. "Click" on the login page
3. Log in with pre-set credentials to a dummy account
4. "Click" on the account balance page
5. Verify that the balance is $2.75

Not only will you be able to determine whether the app/system/site is up or down, you will also gain insight as to the timing of the overall transaction and each of the steps along the way.

The range of applications, systems, and conditions that can be checked via synthetic transactions include network status and performance tests - from the simple checks (ping and traceroute); to more complex for things like BGP, ASNs, and CDNs; Internet protocol tests for DNS and http responsiveness; Web-centric tests for things like page load time or API responsiveness; and multi-step tests like the one described above.

Configuration management

Configurations sit at the border between monitoring and management. On the one hand, a configuration identifies how a thing (a system, application, or operation) works on a fundamental level. On the other hand, knowing that a configuration has changed - and especially when it's changed - can often make the difference between having no idea why a system is suddenly having an issue; and knowing when the problem REALLY started and where to look.

At a high level, configuration management as it relates to monitoring and observability is the process of first identifying which files or elements count as “configuration”, doing an initial scan of that object, and then repeating the process and noting changes.

Generally this is not done by watching files on a filesystem for the “big vendor” routers and switches, though it may be on your Linux or BSD-based services.

So in that case, the monitoring system would have to either use an API (hopefully!) or log into the system using a terminal protocol like ssh (because friends don’t let friends use telnet), running some variation of the “show config” command, capturing (“scraping”) the results, and saving that information to the monitoring system (whether as a file or a database entry). That process would be repeated, and the two results scanned for differences. But that’s not all. Those same configurations can be scanned for everything from syntax errors to security errors.

If issues are found, it could trigger an alert; or it might show up on a periodic report with the new, changed, deleted, or problematic elements highlighted.

To summarize: Configurations may seem at first glance to be outside the scope of what a monitoring and observability tool might care about. It certainly doesn’t seem to fit the description of what “network telemetry” includes.

But in truth configurations are so tightly bound up in system, application, and sub-component stability and performance that NOT monitoring this critical aspect of the environment seems foolhardy at the very least, and possibly negligent when you consider the worst-case (which are sadly becoming more common in these days of companies joining the security-breach-of-the-week club) scenarios.

CHAPTER 2

Wrangling Telemetry

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

Once you've identified the types of devices that make up your network and the types of data they emit; once you've inventoried your environment so you know how many of each type of device you have; and once you've instrumented (a fancy way of saying "set up a tool or tools that can collect telemetry") - once all that's happened and the data is streaming in…Now what?

If we were a vendor of a monitoring and observability solution (spoiler: we are, but let's all pretend we're not for a second), this would be the place we'd show you all the fancy dashboards, sophisticated alerts, and stunning reports you would get if you bought our complete line of products. But it's not really that simple.

Yes, it's true that vendor solutions do a lot of heavy lifting behind the scenes. But this book is about exposing and elaborating on what that behind-the-scenes activity is. Part of the reason is so you can be a more informed customer. Knowing what happens inside the black box of observability solutions, HOW you get from "data comes in the left side" to "pretty charts come out the right side" allows you to make educated decisions about everything from OpenTelemetry methodology to aggregation options to pipeline management.

Once the data is extracted and obtained, there's a whole set of decisions about how it will get normalized, tagged for identification, enriched with additional attributes, routed to (one or more) consuming systems, and finally analyzed and visualized. *To put it into a sound bite: this section describes the way you transform data into information that drives action.*

The other reason for laying things bare is so that you can make an educated choice about the build-vs-buy decision that many monitoring engineers[1] either have to make, or have to defend against from leadership. Saying "I don't have time for that" is a weak argument without business-based justifications. But being able to elaborate on the tasks involved and the level of effort to execute those tasks puts that same response - "I don't have time for that." - in an entirely different context.

To be honest, as Avi and I were sketching the outline for this chapter, we quickly realized that we could do an entire book just on this one subject. Whether we approached things from the functional side ("Who cares about theory, this is how streaming telemetry is aggregated today.") to the academic side ("If you don't extract metrics that are accurate out to with five decimal places, why even bother?") there would be hundreds of pages of analysis, methodology, and opinions (not to mention DRAMA!) to explore.

We might do that someday (Let us know if that's a book you'd like to read!). But, our goal here is to get you thinking about the high-level aspects of "wrangling" your data. That means considering the following questions:

- What is producing my data?

1 Yes, that *IS* a job title, and this is a hill I'm willing to die on. - Leon

- What data is being produced?
- Where do I want that data to end up?
- What do I need to do to the various data sets before I send them on?
- How will I get the data where I want it to go?
- How will I use that data once (and after) it arrives?

And that is actually the perfect intro to the meat of this chapter. The questions above are a pretty good summary of the sections you're about to read. So let's dive in.

Transport

At this point in the process, we can assume you have (or are prepared to get) your inventory of devices, as well as the type(s) of telemetry they produce, and which you want. This section is devoted to exploring the ways you will get that data from its initial collection point to wherever it needs to land - along with some of the intermediate stops along the way.

Inventory of Telemetry Producers and Consumers

I know we just said that you'd already done an inventory, but it's worth taking a second to clarify what that inventory needs to include. First, you need to know what systems you have, what telemetry they are capable of producing, what telemetry you want/need, and how you are collecting that telemetry - both on an individual, system-by-system basis, and also in terms of whether all that data from multiple sources will get aggregated together before being sent. Some of that (especially the aggregation part) will be covered in more depth further along in this chapter.

Once you have that information, you need to create an inventory for "the other side" - where it's going to go. While vendors will be quick to tell you "send all your data to us," that's neither realistic nor cost-effective. In most cases, you don't need each observability tool vendor you use (and trust me, you're using multiple tools even if you aren't personally aware of that fact) to have a complete copy of your data. So your task at this point is to take all those telemetry types and map out where you want them to end up.

If you are feeling resistant to the idea, Avi and I suggest you take a minute and get comfortable with this idea. There simply is no singular "best" monitoring tool out there any more than there's one singular "best" programming language, or car model, or pizza style. There isn't a single tool that will cover 100% of your needs in every single use case.

NOTE

Title: Leon's Singular Sensations

Ahem: Perl, 1967 Ford Mustang 390 GT/A, deep dish from Tel Aviv Kosher Pizza in Chicago

And so our point stands: your network telemetry is going to be consumed by multiple systems: some of which will be super network-specific, while other systems will display network telemetry alongside other layers of the application and/or observability stack.

You need to have (which means both "create" and also "maintain in perpetuity") an inventory of those destinations and use cases.

The good news is that you have a range of options for getting your information to those systems. The bad news is that you have a range of options as to… OK, you get the joke. Your options (detailed below) include:

- Sending directly to each system
- Using a single replicator to send the same data to multiple analytics systems
- Using separate replicators to send the same data to multiple analytics systems
- Using a telemetry bus
- Observing the telemetry flow

Sending direct to each system

This is, to some extent, the default option. It is also, to some extent, the worst of the options we present because it implies that (at least for some architectures) every system is sending all of their unfiltered, un-normalized data to a repository (either on-premises or across the internet) where it has to be collated, correlated, mashed, spindled, and mutilated before it is made presentable in whatever form(s) you need it.

OK, I'm probably exaggerating a little, but ONLY a little. In smaller systems, this design can not only work, but it actually has advantages. First, it's straightforward. It's very clear how you need to prepare every system you want to monitor, and what's going to happen to your data once you set things up. By the nature of the way it works, it is extremely specific in terms of data collected and the devices that are monitored. And it obviously works exactly the way the vendor needs.

But you have to balance that with the fact that you might find yourself managing "1000 points of light data" in the sense that every agent is another opportunity for failure or conflict. I have been at companies that used these types of tools, where a team of 14 people spent their entire day just managing the agents for a couple thousand devices.

Worse, if the ultimate destination of that data is outside the firewall, you might find yourself asking the network team to punch dozens (or hundreds) of holes for each combination of device-destination IP/port. And if you need the same data going to multiple destinations (think "syslog"), you might find yourself using a tool like samplicator to duplicate it and send it in both directions.

Our point is that the "send it direct" method is easy initially, good for a small proof of concept, and simple to manage. But it doesn't scale, and it doesn't even always do what you need. That said, there are circumstances when you don't have a choice - many SNMP systems just need to be polled directly in order to work reliably.

And that takes us back to a point we made earlier: There's no single golden monitoring tool that will serve all your needs. In combination with solutions that scale better, you might find some systems that also need a setup like this.

Our suggestion is to manage it like you would any exception: document both the reason and the details of how it's implemented, and move on with life.

Pattern: Replicating to multiple network analytics systems

For anything except small environments - testing a new tool, seeing how a new device or system responds to monitoring, performing unit or integration tests, etc - sending all your data directly is not really viable. So what is?

A simple, common, and well-established pattern is one that uses a 3-tier model to collect the data using inexpensive (from the standpoint of both cost and compute resources) middle-layer systems before sending it to its final destination.

This design pattern has been in place almost since the inception of monitoring. Systems send data - either through agents, built-in subcomponents like SNMP (which is effectively an agent), or a script that opens a specific IP/port combination (<cough>AGENT!<cough>)on-box - to a device that runs software purpose-built to collect the data, aggregate it with data coming from other systems, and send it all to the main monitoring and observability solution. Data might be sent as a continuous stream, in bursts, or a combination of the two.

Lest you are left with the false impression that this middle-tier "collector" system needs to be powerful, I want to point out that the nature of the task - receiving data (usually on the same local network), quickly processing it, and sending it along - isn't as compute-intensive as it seems. Therefore, collectors can often operate in extremely economical form factors, including VMs and containers.

There are multiple benefits to this design. Foremost among them is the obvious advantage over the previous pattern. The single collector means fewer outbound connections and firewall exceptions. It also provides the opportunity to implement security options for the data before it moves to the next stop along the way. And speaking of moving the data, because collectors are usually in the same network, it means you reduce egress costs (especially for cloud environments) because the data can be filtered before it exits the network.

And that takes us to the concept of pipelines and how they relate to data wrangling.

Let's (briefly) start with the fundamental concept: In Unix (and unix-like systems like Linux, which I'm going to abbreviate to *nix from hereon), pipeline was the term for taking data of some kind (whether it was a static value or the output of a command or the contents of a file) and applying one or more other commands to it, so that the result was transformed. It was also called pipeline because it used the pipe symbol (`|`) to push data from one command to another.

The simplest example of this was taking the output of a file list (using the `ls` command) and pausing after a single screen of information by piping it through the `more` command:

```
ls | more
```

While we could wax poetic about the most elegant *nix pipelines we've written in our day, both Leon and Avi will show uncharacteristic restraint. Our point wasn't that *nix pipelines are similar to modern observability pipelines, but rather that they are the conceptual ancestor.

Data -> pipeline () wrangling happens here () -> output sent to destination.

Pipelines, as they apply to modern monitoring and observability, are far more powerful (and nuanced), so I want to take a moment now to explain some of their capabilities.

Input normalization

Normalization refers to taking multiple forms of input and making them consistent. For example, if some data is in bits per second, and other data is in bytes per second, and still other data is in megabytes per second, the process of normalization means converting everything into a single consistent scale. This, the more commonly understood type of normalization, is actually *semantic normalization*.

Another form of normalization that might occur is *syntactic normalization*. This is the process of making the output match the same structure. For example, one set of messages uses this structure:

```
{ID=123456, os="windows", cpu=12, mem=32}
```

And another looks like this:

```
ID        CPU MEM OS
123456    12  32  windows
```

Syntax normalization would shift one or the other so that they both used the same order, method of assignment, etc.

One important thing to note about input normalization is that there's no one-normalization-fits-all. The goal of this step is to get all the incoming data into the same scale and format, but with the intention of making it easy to further massage in later steps (manipulation, replication, etc.).

Data Set Manipulation

This step, which actually includes multiple variations on the theme, is all about taking the full set of data and somehow paring it down.

Filtering. If you've ever run a database query, you already understand *filtering*. If you only want customers from Ohio, or just the customers from Cleveland, Ohio, or just the customers named "Leon" from Cleveland, Ohio… then you're filtering.

Specifically, you are performing row filtering. You want all of the fields, but only for certain records/rows that match criteria.

It's important to note that you can also (or alternatively) filter columns. If you want just the Last Name, Hair Color, and Dog's Name columns - even though your database has a host of other details like eye color, cat's name, first name, etc., then that's an example of *column filtering*.

Obviously (but we're going to say it anyway), you can filter both rows and columns at the same time, for good reason.

Aggregation. *Aggregation* is the act of summarizing multiple rows into a single one, with numbers combined or manipulated in some way to result in a single value. If you've ever used a spreadsheet, the standard aggregation options should be familiar:

SUM
: Totalling all the values

AVERAGE
: Add up all the values, divide by the number of values in the list

MODE
: Display the number that appears most frequently

MINIMUM
: The lowest value of the set

… and so on.

It's important to note that, while aggregation can occur in the pipeline phase, it may also be an operation that's done once the data is in the destination repository as well. For example, data may be collected and transmitted every 5 seconds, and stored that way initially. But after 7 days, the data is aggregated into 1-hour summaries (averages). And after 30 days, the data is further summarized to a single row representing the average for the day.

Rollup. *Rollup* is also sometimes referred to as "flattening". It's the act of taking multiple rows of data and turning those row values into columns. An example may make this easier to quickly understand. Let's say my data looks like this:

Interface ID	Port	Date	Bandwidth (Gb)
12345	8080	1/1/2025	30.7
12345	443	1/1/2025	100.5
12345	8080	1/2/2035	27.6
12345	443	1/2/2035	50.2
67890	8080	1/2/2035	78.9
12345	8080	1/3/2035	36.1
12345	443	1/3/2025	22.2
67890	8080	1/3/2025	85.4
67890	443	1/3/2025	99.7

If you look, you'll see that we're really just talking about two interfaces, and the data they sent on two ports over three consecutive days. So we can "flatten" this by turning the port-day combination into columns:

InterfaceID	8080-1	8080-2	8080-3	443-1	443-2	443-3
12345	30.7	27.6	36.1	100.5	50.2	22.2
67890		78.9	85.4			99.7

From the example, you can see that with rollup/flattening, we send fewer rows of data, even if we're sending more columns. Depending on your application and/or the ultimate destination of the data, this might be advantageous either from a presentation or cost perspective.

NOTE

You can combine rollup with aggregation to great effect, by taking multiple aggregation operations (sum, average, min, max) and combining them to a single row:

InterfaceID	Date	Sum	Min	Max	Mean	Median	Mode
12345	1/1/2025	300.767	36.1	100.5	44.55	33.4	nul

Sampling. Sampling also involves only taking part of the total data set, but it shouldn't be confused with filtering. Filtering is both specific and intentional - the query you write determines which records and columns are kept or not. Sampling, on the other hand, uses an (almost) random method of determining which records to keep.

Before diving into how sampling works, we need to address exactly why you'd want to sample in the first place. Once again, the heart of the issue is that your observability system simply has too much data to handle. It's either too expensive (to transport, process, or store) or it's too big to use effectively (queries on the entire data set are cumbersome or create their own compute issues). You need to reduce the overall volume, but there's nothing about the data set that lends itself to filtering. Any record might be useful, but keeping every record is untenable.

In addition, while any record MIGHT be useful, the reality is that a lot of telemetry data is repetitive. If a large number of your data points show the same thing, you don't need every single one to meaningfully observe your system. You just need to show a representative sample of the records you see.

This is where the concept of fidelity comes in. Back in the 1950's, audio manufacturers began using the term high fidelity to describe the way their equipment faithfully reproduced sound to be as close as possible to the original recording. The concept, if not the exact term, carried over to images, and especially to digital photography. But what was discovered was that fidelity mattered less with images. In fact, certain elements of an image could be removed (reducing fidelity) without any difference distinguishable to the human eye.

The point we're making here is that what's true for images is true for telemetry as well. You can lose some data but retain the integrity of the whole, while achieving a meaningful reduction in the overall size of the data set. That's the intent behind sampling.

There are two common sampling methods: Head and Tail.

Head-Based Sampling
: This is where sampling occurs as early in the pipeline process as possible - before the entire telemetry record is inspected. The upsides to head sampling are that it's easy to understand, easy to configure, efficient, and can be implemented at any point in the pipeline.

  But we'll be honest: there is a lot to dislike about head-based sampling. You're more likely to miss interesting/useful data. Worse, you might never know what you're missing or that you missed it at all. That said, it does have a place in sampling strategies. There are two primary head-based sampling techniques:

  Consistent Probability (aka Deterministic)
  : By far the most common, this technique uses a straight ratio - sampling every 10th record, for example. There's no further rhyme or reason to it. You're just grabbing a mishmash of data and hoping/believing it's representative of the whole.

    And, we have to admit, for gigantic data sets, it probably IS representative.

  Parent
  : If the records you're dealing with have relationships to other records (sometimes referred to as *context* or *parent-child*), then this sampling method makes an initial decision about the parent record and then propagates the decision to all related records. If the parent record is dropped, all subsequent records are, too.

Tail-Based Sampling
: As you can probably imagine, tail-based sampling happens in the place opposite from head-based sampling–meaning at (or near) the end of the process rather than the beginning. Head sampling doesn't wait for the complete data set to be recorded before deciding to keep or drop it. Tail-based sampling does.

Because of that, there are a few tail-based sampling methods we should discuss:

Dynamic sampling
: This adds a data element to a record that indicates the frequency of this type of record, and samples it more often for less-frequently produced results. Example: if sampling on HTTP status code, it will sample 404 results more often than it will for 200.

Rules-based sampling
: This combines the concept of filtering with dynamic sampling. For example, 100% of records with error conditions are sampled, and all other records use the dynamic sampling method.

Throughput-based sampling
: This technique looks at the overall volume of data records being processed, and drops any above a fixed threshold of records per second. Thus, on a second-by-second basis, records are treated as first-come-first-accepted, but overall, the ratio of the types of telemetry is (somewhat) maintained.

Deterministic probability sampling
: If you're thinking, "Hey! The editor made a copy-paste error!" you need to get off Gary's back because he's awesome, and we did this on purpose. The truth is that you can ALSO use a ratio at this point in the pipeline if you want. It works the same way; the only difference is that you might have already dropped (sampled out) a bunch of records by this point via other sampling methods.

  What's that you say? "You're totally sorry about thinking ill of Gary, but what do we mean "might have already dropped a bunch of records via other sampling methods?"

  We are happy to say you're not limited to a single sampling method within a pipeline.

  - You could create a rule that samples just 20% of the records with "OK" conditions (i.e., rules-based);
  - Then use dynamic sampling for each of the error conditions;

- Then have a throughput-based sampling rule in place if the volume of records still exceeds a threshold.

With all of that said, I will leave you with a thought from a different O'Reilly book, *Learning OpenTelemetry*, by Ted Young and Austin Parker:

> "Filtering is easy; sampling is dangerous."
>
> "How to filter your telemetry is usually obvious - simply throw out any data you do not plan on ever using. When and how to sample is a much more difficult question to answer. [...]there is always the possibility for a critical error to occur infrequently enough that it is missed entirely by a sample. [...] missing any errors at all seems like a bad quality for an observability system to have! [...but…] tail-based sampling could actually cost you more in machine resources than you save in network egress costs."

Our conclusion aligns with Ted and Austin on this: avoid sampling until your egress and storage costs become significant. Lean instead (and first) on filtering, compression (discussed later in this chapter), and simply being intentional about the telemetry you need and want in the first place.

Output normalization. You've filtered, you've rolled it up, you've sampled it, you've sliced it, diced it, and made it into julienne fries. Now it's time to send your data on its way. This step is actually pretty straightforward because the tool you're using - whichever one it is - will, if it's any kind of pipeline tool at all, do almost all the heavy lifting for you. All you have to do is make a few choices and indicate them in the right spot in the (probably) YAML file:

First: What syntax do you want?
: You've got a handful of choices, including JSON, CSV, and GNMI. The answer will likely be found in the system you're sending the data *to*.

Second: What semantics do you need?
: When it comes to data, "semantics" is just a fancy word for "what are the columns in your database called, and what order do they appear in?" Many target systems require data to be in a particular order. That's all this step is asking you to specify.

Third: What transport will you use?
: Like spice on that famous desert planet (no, not Tatooine, the other one), the data must flow. And to do that, you have to

pick your poiso… I mean protocol. Like syntax, you have a handful of options to pick from, including Kafka, TLS, GRPC, and OTLP. But (again, like syntax) the one you use is likely going to be determined for you by the destination system. Your job here is simply to indicate that protocol so that the pipeline can do its magic.

Replication. You've filtered, you've rolled up, you've sampled, you've sliced, diced, and made julienne fries. Now it's… wait a minute, we already said that!

OH… I get it! We duplicated the same data, and we're sending it out again! It turns out that your pipeline can send data to more than one place at a time. We'll cover this as its own pattern/stand-alone process in the next section, but there are good reasons why you might want to utilize this aspect of pipelines instead.

While the techniques we'll cover in the next section focus on duplicating the data exactly, with no (or minimal) changes. Replicating data within the pipeline allows you to take the same source data, transform it two (or more) different ways, after which you can send every version in its own direction.

But there are downsides to be aware of:

- Some telemetry bus (i.e., pipeline) systems don't have connectors to some external vendors. If you end up in one of those mismatches, you either need another telemetry bus (which is… not great) or you need to fall back to one of the other patterns (separate replication, vendor agent, etc.)
- Overall, the tools you need to manage a fleet of collectors/telemetry buses aren't as mature as they could be I mean they will be I mean, you might need.
- Depending on which telemetry bus/pipelining system you use, load balancing - moving a group of systems, a single discrete system, or a fraction of a system's metrics - can be complicated. Or it might be automated to the point where you have no control at all. Neither of which is optimal depending on your needs.

ORDER NOW! OPERATORS STANDING BY!

I've dropped the "depending on which tool you use" comment a few times now, which is about when, if this book were a thinly veiled bit of marketing copy, we'd say, "BUY CHOCOLATE FROSTED OBSERVABILITY BOMBS TODAY!" But this isn't, so we won't. There's a host of good options out there, both open-sourced and closed, for you to choose from. My only goal here in pointing out that some tools do/don't perform a certain type of function is to help you build your checklist of things you might need.

Strike that, Reverse it

After talking about telemetry busses as a one-way trip from the source system to its final destination, we want to take a moment to point out that some vendors are now allowing you to do the exact opposite: Send your data in its raw form and store it as-is in what's come to be called a *data lake*. (Fed, of course, by a crystal clear data stream, which babbles next to an eco-friendly data warehouse. Feel free to set up a data cabin and sit by the data shore doing some light data fishing with your faithful data dog. But I data-digress…)

The concept of a data lake is that the information stored in it is all in its natural/raw format. You can then specify a chunk of that telemetry and "play it back" - re-running the exact flow of data as it was stored. As you do THAT, you can run any transformations - normalization, filtering, sampling, and the rest - and then use (analyze, display, report on) it however you like.

The benefits of this technique include the fact that you'll lose none of the fidelity of the original data. The collectors or relay systems (agents) will have far more processing capability available for moving telemetry since no other functions (filtering, sampling, etc) will be done. You can use that available compute power to focus on compressing the data to reduce overall traffic volume (and thus cost). And since data lakes tend to be in the cloud or other locations with lots of compute resources, the act of replaying the data can be done faster because more processing power is available.

Options to replicate telemetry: Separate replicators

You've filtered, you've rolled up, you've sampled, you've… wait a minute, we already said that! OH SNAP! We already said that, too!

Because replication means that if it's funny once, it'll be funny over and over again, right?

Ahem.

If you are:

1. In a situation like the one we described in the "Sending direct to each system" section, where you have data that needs to go to multiple destinations, but
2. You can't employ a sophisticated telemetry bus type option like the ones we outlined in the "Pattern: Replicating to multiple network analytics systems" section, then your next best option is to use a tool or utility that takes a single raw data stream and splits it into two identical branches. Doing so allows whatever you're using - agents, home-grown scripts, etc - to handle things from there, sending each discrete copy of the original data to whatever destinations you're using.

If this sounds familiar, it's because we're essentially talking about a pumped-up version of the `tee` command line utility found on Unix and Linux systems - which is itself a computerized implementation of the humble t-splitter found in plumbing.

Once again, you might expect us to now dig into various tools that do this function, but our job in this book isn't to point you toward any particular tool, whether open-source or closed. We've described the issue and the potential solution, and we trust you to do the research from here.

Observing the telemetry flow

> *Quis custodiet ipsos custodes?*
>
> *Who will watch the watchmen?*
>
> —Satires by Juvenal

Yes, this is the part of the book where we talk about monitoring for your monitoring. If you already know, you know, and you should feel free to skip to the next chapter. But if none of what I just wrote is having you nod your head and/or roll your eyes, please keep reading.

One of the challenges with observability - network or otherwise - is that it's so challenging in its own right, that there are so many moving parts, that the journey from start to end is one of many miles and many days; that once you have telemetry flowing it's hard to remember that simply getting it working doesn't mean it's working well, or consistently.

Stripping away the poetry, what I mean is that sometimes we get so caught up in just getting things working that we forget to put checks in place to ensure that it - the monitoring itself - continues to work the way we expect, the way it should, the way we need.

What kinds of issues might crop up with monitoring that wouldn't be immediately obvious because they cause a complete failure of the datastream?

Sources have disappeared

This one *should* show up pretty quickly, even without monitoring, but depending on the scale and scope of your environment, "pretty quickly" still might mean hours or longer. So it's prudent to have a test in place to show if data stops coming from various sources. Now (hopefully) obviously, we're not suggesting you monitor every target system twice - once for the actual monitoring, and once to ensure that the data is flowing to the actual monitoring. First, you should do this only for those sources you consider critical. Second, the check we're describing is not on the system itself, but is essentially a heartbeat test in the receiving pipeline.

Extending this concept further, a more thorough way to express this test is to look for the "wrong quantity" of data - too little, but also too much. Because a spike in data might be just as concerning as none at all.

Consumers/destinations not accepting data

At the opposite end of the spectrum from disappearing sources is the problem of destinations no longer accepting data. The good news is that this type of check is both easier to synthesize and far narrower in scope.

Records sent in the wrong "shape"

If the data comes to an agent or telemetry bus in a format it's not expecting, it creates a problem for every step after that, from normalizing to filtering, sampling, and the rest. Worse is when records are inconsistent - some come in as expected while others are, for lack of a better term, malformed. Being able to identify when this happens is the first step to creating more robust rules for data intake.

Not dropping data
: Remember all those discussions about filtering, flattening, and sampling? What if the observability system just… doesn't? This causes more than just a problem of sending more data than you want. This would also cause you to send the wrong data. And it could mean that you have unnecessary data in such volumes that the real issues are lost in a needle-in-the-haystack situation. Plus, all of those issues happen on top of the first and most pernicious one: every dashboard, every alert, every report represents a query against some (or all) of the data. All those queries add up fast and impact performance, user experience, and the ability of the system to do its core functions.

Records (or sub-records) are disappearing
: This issue is more insidious and even more damaging to a healthy and reliable observability system: Data that is being sent by the originating sources, received into the agent or pipeline, but lost somewhere before reaching their destination. The effort to create a system that accounts for each record in this way is by no means trivial. But without it, the reliability of the data stream is difficult to ensure with 100% confidence.

Some of these issues only require you to validate and check for a short period, like during setup. Some are things you should do occasionally, as part of a routine validation. And some might need to be done persistently. Kind of like monitoring and observability itself.

One example of a persistent check is called "beacons" (sometimes referred to as a "deadman's switch"). This is a regular, but synthetic, data element that's created at (or as close as possible) to the same targets that are origins of data streams. It flows through the system all the way to the typical destination for that telemetry type. An alert is generated if the beacon DOESN'T arrive. The goal of this exercise is to validate your observability system end-to-end, and to have as advanced a notification of its failure as possible.

Enriching Telemetry

What Is Enrichment?

Most of us first encounter the word "enrichment" when we're in the kitchen with a beloved relative, learning to bake.

> **NOTE**
>
> (Avi speaking) Leon's perception might be JUST A TRIFLE skewed. His daughter runs a bakery out of the house, and his primary source of exercise comes from shuttling 50lb bags of flour to the basement. His primary reason for needing the exercise is, as I mentioned, the bakery. Nevertheless, we humor him when he insists that "baking 100 loaves of challah and three cakes every week is TOTALLY NORMAL BEHAVIOR".

Enriched flour has extra nutrients - iron, B vitamins, and the like, which aren't normally found in regular flour - added. Thus, "enriching" data means inserting additional information into a data stream. It's the exact same process, although not nearly as delicious.

By itself, telemetry data has information specific to the observation point and source. for example, flow data on routers contains information about source, destination, protocol, flags, and other things the router is aware of.

But if you need to find out "which users are having issues with which applications," somehow the telemetry needs to be enriched with additional information.

We think of these additional data sources as metadata - data about the data - and metadata needs to be gathered and put into some sort of lookup table, or in more complex scenarios, a set of rules for applying the lookups.

Adding metadata in this way is commonly referred to as "enrichment".

Types of Enrichment

Some of the most common types of enrichment for network telemetry are:

Routing data

For analyzing how traffic is flowing across networks, and especially across the internet, it's helpful to have BGP routing information, especially the AS_PATH (a list of "network numbers", called ASNs) and Community strings (lists of policies to be applied by a given network). Adding elements like this to the device metrics, traffic, and performance data you're collecting allows you to create more sophisticated and robust alerts, searches, analysis, and visualizations.

> If you hang around a certain type of network nerd you will hear them mention AS-es, or ASNs ("Autonomous Systems" and "Autonomous System Numbers", respectively). They may even refer to common applications by their ASN (Netflix is AS40027, for example. "Oh yeah, that'll kick you in the old AS40027! Amirite!?!"). But what ***is*** an AS, exactly? And more importantly, is there an ELI5 (Explain It Like I'm 5) version for people who don't live and breathe routing tables and network protocols?
>
> Yes there is, and here you go:
>
> At its most basic level, an AS can be thought of as your ISP's router network. It is a self-contained, self-maintained, and self-optimized (where "self" is the ISP) collection of routers that allows traffic to enter, traverse, and exit as quickly and efficiently as possible. It's also a full mesh style network, where every single device knows about (and all the different ways to get to) every other device.
>
> ***What are AS-es? The ELI5 version***: Let's imagine that "the internet" is like all the roads on the planet. Every road goes somewhere, and you can (more or less) get from any road to any other road (eventually). In that example, an AS is like all the roads in a city. The city maintains all the roads inside the city and publishes maps to help folks get around quickly inside the city. BUT it also publishes the fastest routes OUT of the city to get to other cities (this way to LA, that way to Cleveland).

Some network devices can add the source ASN to telemetry, but very few can add the full AS_PATH or other BGP attributes.

Since there can be 1 million or more BGP routes to combine, doing this kind of enrichment requires robust and scalable systems, which we'll talk about later.

Geography: Physical location

For devices in your network that you know the location of, enriching traffic, metrics, logs, and other data with the device location (usually city and country, and sometimes even lat/long) can enable many use cases from simple querying ("show me total traffic in the US") to enabling map displays.

Geography: IP Geolocation

A common further geography question people want to ask involves the source and destination city, country, or other geographical slices/dices. Because it's WAY more interesting to see "I have 100gbps of traffic going from Cleveland, Ohio to Timbuktu, Mali" than a chart that shows "traffic from 134.228.65.139 to 102.130.232.0."

Application/Service

In addition to slicing and dicing traffic by geography and network, it's also really useful to be able to ask questions about application traffic traversing the network. In this case, we're identifying an "application" by the combination of ports, protocols, and (sometimes) IP addresses.

Is this a gross approximation? You bet! But it's also fast, simple, and remarkably accurate in a large number of use cases. - i.e., TCP port 23 is "ssh", TCP port 443 is "Web". Yes, Leon and I both acknowledge that any application can decide to run on (almost) any port it wants to. But we must insist that you, dear reader, equally acknowledge that just because they CAN doesn't mean they DO.

Systems of record - "What is this IP address really?"

Many organizations have systems of record, like IP Address Management (IPAM) systems or device tracking systems such as NetBox. These tools - as well as application orchestration systems like Kubernetes - can be mined to get even more context on what different IP or even IP/port combinations are really doing on your network. Other backend systems, like DHCP or authentication services, can provide similarly useful data to map a user (or even a specific user's computer) to an IP.

As a real-world example that companies are still very much interested in, you could use those sources to enrich every monitoring data point from a device and include its physical location down to rack and position inside the rack. Another example: using those sources to be able to tellthat, right now, a given IP and port combination is running the auth1 service for application "foo2".

This data can be extraordinarily useful when merged with network telemetry, but there are a few challenges to be aware of:

- The volume of mappings can be very high. In some organizations, they can surge into the millions; and
- Modern applications and infrastructure, especially the infrastructure underlying "the cloud" and cloud-native uses of that infrastructure, can change rapidly - as much as every second. So live streaming updates may be required in order to get accurate data / attribution.

Business Metadata

Another valuable use case for enrichment of network telemetry is adding in context like customer or internal group/division association.

For service providers, that can be as simple as associating all telemetry coming from a specific interface that's connected to ("terminates a customer link", as us old ISP veterans say) as being for that Customer ID.

For both enterprises and service providers, sometimes customer traffic can be multiplexed inside larger interfaces or come in over the internet, and association by creating a customer<->IP address mapping will be required.

As an example, an enterprise may be trying to debugg a customer complaint about not being able to use a given application. Having the ability to query underlying network traffic by customer would illuminate whether that traffic is even reachingthe enterprise at all; or not making it to (or through) load balancers; or being blocked there; or having some other issue.

Another use case, with the sad-but-true observation that every customer is equal but some are more equal than others, is when network providers tag (i.e. enrich the data to show) priority customers. This then allows the provider to capture, display, and use the tags in their monitoring system, to do things like escalate alarms if issues occur with the aforementioned VIPs.

As valuable as this is, these types of enrichment often get tripped up because of a “system of record” challenge. There may not be a single place to get those mappings. It’s emblematic of how important this enrichment has become that, as part of the network automation efforts at many enterprises, a critical first step is to build a single (or federated) system of record to be authoritative for customer and other mappings.

Building and management enrichment

In modern infrastructures, things are constantly changing, even as fundamentally (to the networking practitioner) as information like which IP address is being used for what task or system - and where, and when.

Making sense of network data requires enriching it with application, location, BGP path, and other types of these dynamically changing metadata sources, and therefore enrichment can’t be done at the time you’re running a query within the monitoring solution. That enrichment has to be done live (or nearly so), at the moment your observability tool is perceiving and receiving the data.

We’ll talk more about this later but on solution to this is to leverage streaming databases and pipelines that can do “streaming joins” to add on all of these enrichment types.

The big things to know about selecting and feeding these are:

1. Some of these systems have limits that are way below the scope or velocity of network data and how it changes - for example, there are on the order of 1 million BGP routes and they can change at a rate of tens of thousands per second (or more). So static tables loaded every 5 minutes, or with limits of tens or hundreds of thousands of entries, won’t allow joining BGP routes, many enterprise application mappings, or other kinds of metadata sources.
2. Nobody is going to do this for you (ask us how we know). It’s your job to find and manage feeding the data to be joined in the enrichment phases, and this can be a bit of a puzzle hunt, especially across teams in enterprises where ownership of IPAM, customer mappings, application orchestrators, and systems of truth for network are widely spread across owners in many unrelated teams, departments, and even (sub) companies.

Normalizing Telemetry

Unifying and Normalizing Telemetry

For each kind of telemetry there are often multiple implementations and sources, which can each have different formats and meanings! This is, obviously, why observability tool vendors drink. After you collect it, a critical part of making it useful is to "normalize" it into a common representation. We covered normalization - both the need for it and some of the ways it's done - earlier in this chapter. We're mentioned it again here to put it into its proper context in terms of the whole "wrangling" process.

Normalizing Device Metrics

Wait, didn't we just mention "normalizing" above? Well, yes. But it was in the context of telemetry overall. Here we're specifically referring about what it takes to normalize ***device metrics***, which has its own set of nuances.

Network devices expose many metrics about their physical and logical state - for example, interface statistics, temperatures, routing table states and counts. The same metric, bytes sent over a given interface, might be available in different formats and protocols. For example:

- Via SNMP as: <ifMIB>
- Via Streaming Telemetry as: <openconfig target>
- Via API as: <JSON>
- Via the router's CLI as text: <sho int output>

To make sense of how many bytes are going through interfaces for a set of devices, it's often necessary to collect data by multiple methods across your devices.

I also want to point out that it's VERY possible to collect the same metric for the same time period from the same device using multiple methods. There are a variety of reasons for this - none of which are particularly satisfying - but I mention it here simply to reassure you that YOU are not doing something wrong if this happens to you. The main callout here is that it's something you need to watch out for.

Normalizing metric names

That leaves you with the job of normalizing those all to a common format - after the data has been ingested into your primary system, but before that data has been written to storage (i.e. the database). Because if you don't, you'll be forcing the users to swive between multiple systems displaying on multiple screens, trying to do it all in their head at query time.

Unfortunately, there's no IETF or multi-vendor standard on this. Like parenting, we're all just making it up as we go along. Of course, some of us have the benefit of years of hard-won experience (again, like parenting).

One common way to do this is to to build a translation layer and produce the output as a metric series, so that interface byte count above would become a series of bytes using, for example, a graphite-style metric name like:

```
network.<devicename>.interface.bytes
```

Normalizing metric measurements

There are two additional , and common, things you may need to adjust for when comparing metrics across devices and device types:

- Counter vs. rate - Many things are measured on network devices in counters - for example, just a total of bytes in or out across an interface. To turn that into a rate (bits/second) then requires observing the counter across two time points and dividing the delta in counter (total bytes seen) by the time window. (Sidebar note: This is why it's hard to get the same counter/rate from two systems observing the same network elements). You should also understand that even under the category of "counter" there are variations:
  - — Increase-only: the number goes up and up and never stops.
  - — Increase with a resetting: the number goes up to a fixed ceiling, and then "rolls over" to zero. Obviously that requires a bit of math because if the number "now" is bigger than the number "before", we have to realize there was a rollover and do some quick addition.
  - — Increase and decrease: the counter can move in both directions, representing the current state irrespective of the previous value.

- When you are using aggregated interfaces, some devices can report on the metrics for the aggregate as well as the "member" interfaces, but some can only report on the aggregate or the member interfaces. As a result, to let users (and our machine overlords) access the data consistently, some mapping is needed. Sometimes this is done in the collection process, and sometimes this is done in the normalization layer.
- Performance, when exposed via device metrics, needs special consideration, covered below.

Normalizing traffic data

There are many sources of data that all show "what went where and when". Some of these include:

- NetFlow, IPFIX, and sFlow from routers
- eBPF and flow from virtual and physical servers
- VPC Flow Logs from cloud providers
- and even logs from web services, load balancers, service meshes, and firewalls.

Just to be extra clear, within these divisions there is nevertheless a lot of overlap. For example web logs (somewhat obviously) have application context, and some even have more protocol-level information like the more network flow protocols. And of those network device protocols, sFlow knows about VLANs and MAC addresses but NetFlow v5 doesn't.

Breadth of traffic dimensions

As with network device metrics, traffic data can come in different formats like NetFlow v5, NetFlow v9, IPFIX, sFlow, web logs, or eBPF traffic observations.

Traffic records have many metric points embedded within one "row" or record, and are almost always representing counters.

The main complexity of normalizing traffic records is that different types of devices, and different "age" of protocols, report on a smaller or larger set of potential dimensions of the traffic data observed.

For example, almost all protocols and devices will report on some core elements like protocol, source and destination IPv4 addresses, and source and destination port numbers.

NetFlow v9, IPFIX, and sFlow will usually report on IPv6 addresses and VLAN numbers (if any), but NetFlow v5 has no understanding of those data types.

Almost all network forwarding devices will also report on TCP flags (for TCP traffic), but web logs won't have TCP flags because they'll usually be observed at a higher level of the protocol stack.

So how to combine traffic data from all of these sources usefully?

The key is to not be too fussy - particularly about NULLs, and to help users keep in mind how much of the network is observable in some of the more "advanced" dimensions.

For example, common questions about network traffic like "how much traffic was seen across the network yesterday" will be answerable using input data from each source.

But "how much *IPv6* traffic was seen yesterday" won't include slices of the network reported on by devices that only support NetFlow v5.

Sampling

A secondary complexity of normalizing traffic data is that there's just so much of it, especially at modern traffic levels. We already introduced the concepts behind sampling in Chapter 2, but to reiterate: sampling is required to avoid overwhelming the network devices and telemetry systems handling the data.

This is important to understand in two contexts:

- First, the data you receive may come pre-sampled. Some network flow protocols will record the sample rate (which can, on some implementations, change over time), and others do not. If you consume data from a telemetry bus this could be a further complication.

  By way of example, at Kentik we built our backend with the goal of robust observability (which includes sampling in it's many and splendored forms), and therefore we made sure to record and store the sample rate per record along with the data, so that we can adjust the results at query time. But (and this is not a humble-brag, it's just a fact) most backend database systems (especially the off-the-shelf varieties) don't - and often can't - do that.

So generally you will need to multiply the counters like number of bytes and packets by the sample rate at ingest time so the numbers are correct at query time.

- Second, you may need to down-sample (meaning sample the already sampled data) because the telemetry ingest rate is too high for your ingest, normalization, enrichment, pipeline, and/or storage and query systems. In these cases you'll need to do that sampling and pass on how much was re-sampled to do the "up sampling" of counters as described above.

Not over-counting (sometimes called "deduplication")

If you collect data from every device in your network, you'll probably have to deal with making sure that you don't multiply count the bytes flowing across it.

One way is to only collect flow from one "plane" of your network - i.e. inbound from the internet edge, OR outbound to your WAN, but not both.

Another (older) technique was to "deduplicate," where you would assume all the traffic flows were unsampled, send everything to a single big or logical collector, and keep just 1 copy for a given "flow hash" (combination of protocol, source and dest IP, source and dest protocol, for a given time range). Again, it's imperative to point out that the underlying assumption here was that all the traffic flows were unsampled.

But in a world where network devices are mostly generated sampled traffic data via NetFlow, IPFix, and sFlow, you will rarely see enough copies of the same traffic communication in the same time period, so deduplication doesn't really help.

A more modern way of dealing with the single-counting challenge is to "tag" each traffic source with what part of the network it is - backbone, data center edge, data center, cloud edge, cloud, internet edge, etc and then structure your queries so that they only ask about traffic across the one "plane" at a time mentioned above. This approach can be effective at scale and with very diverse networks but can require some planning in how you set up queries, alerts, and dashboards.

Normalizing Performance Telemetry

Most teams (and more to the point, most monitoring tools) combine performance telemetry data (like loss, latency, and throughput), and allow comparisons of these measurements across kinds of measurements.

Because the measurements are (almost always) in the same units, the challenge is not the same as we saw with flows. It's perfectly safe to store the number of latency metrics (for example) observed from different sources at a particular point in time; and then compare them later.

The challenge here is understanding that when latency is reported in network metrics or other kinds of telemetry, it's often measured differently from device to device.

For example, when determining "latency" (i.e. the time it takes for a packet or message to get from point A to point Z), some devices may look at TCP session setup time as the starting point; others may be reporting on active performance testing (like an ICMP ping); and others may be looking at a kernel's TCP timers. And because these numbers are collected and sometimes aggregated across devices, it really can appear to be apples-to-apples when it's not.

Usually people do compare these latency numbers without trying to adjust them, and honestly, that is (broadly speaking) OK. But it's important to keep in mind that any such comparisons should be treated as directional (i.e. pointing toward a potential issue, rather than not definitively identifying a specific problem) and not read too much into, for example, a 20ms latency difference between two 80ms measurements made by different methods.

Normalizing Events / Logs

Structurally, things that report in long strings of text are typically called "logs", but, to paraphrase "Animal Farm", some logs are more structured than others.

Besides certain parameters that (usually) appear in all logs like timestamp and sending source, the real work regarding logs is putting them into more normal (and normalized) schema. This makes them better (easier, faster, more performant) for querying, and has the added benefit of making them more "event"-like - telling you that a (thing like this) happened at (X time), with (these parameters).

In the networking world, there are some router device logs that are usually particularly interesting to look for and correlate - i.e. out of memory, killed a process, login or config event, routing session or interface down. Especially configuration events as those can often lead to temporary issues, or worse.

The more structured the events and your storage of them, the easier it will be to both search and correlate them with other telemetry to debug issues (or prove that potential issues aren't).

It's true to the point of being axiomatic that, for IT folks, metrics are often easier for many groups to fit into their mental model and watch at a high level. It's certainly true that metrics are easier for tools detect, collect, and alert on..

So to take large numbers of logs and make them more understandable, a great approach is to put some structure on them and then turn those logs into metrics - i.e. how many logins, interface transitions, or high memory reports per unit time.

Normalizing Other Telemetry Types

So far we've covered flow-type data, metrics, and logs. Those are certainly not the only types of data you will encounter. However, from this point forward, everything else is simply a variation on those three themes. For other telemetry types, you'll use some of the same "meta" processes as for the more core network telemetry types:

- Figuring out what meaning is contained in the telemetry type.
- Mapping how that relates to other telemetry that you have
- Often, summarizing or otherwise turning the new source of telemetry into something you already store and work with operationally
- Sometimes keeping the raw telemetry source

(Same general process - what is it, turn it into common intermediate representation, make some editorial semantic decisions)

Data Storage and Query

Data storage, ingest, and query requirements

Once data has been detected, captured, sampled, normalized, sampled, transported (and hopefully not folded, spindled, or mutilated, to hearken back to an old CBC show), it needs to be put someplace (i.e., stored). And beyond that, it's not enough to just HAVE the data, you probably want to DO things with it. Storing and working with telemetry requires a database (or, in newer speak, "data store").

There's no magic database out there - each one has advantages and disadvantages. And by disadvantage, I mean that if you try certain insert and query patterns, you can crush (and now we're back to the whole "spindle and mutilate" thing) just about any database. So it's critical to find the best tool for the job, depending on the shape of the telemetry and the anticipated or actual query patterns.

Dimensions of data that help guide choice of database

Not to get too spiritual about it, but data has a "shape" and even a "texture". This loosely refers to the scope of the data - the variety of fields, the number of tables, and such - and the granularity of that data. The shape and texture of the data, as well as that of the query patterns you'll use to extract and present the data, will help guide to the best choice of database.

Some key considerations are:

- How many events/second are being observed and need to be stored?
- How long do you want to store the data for?
- Do you ever need to delete data?
- Do you need the database to handle adding enrichment?
- How many columns or tag types do you need for the telemetry type? Few? Dozens? Hundreds? More?
- How many unique values will you have per column - or per unique combination of column (the "cardinality" of your data)

- Can you host the data in the cloud using a database as a service? Or do you need to self-host for compliance, cost, or other reasons?

The answers to these questions will provide you with context for the discussions below, which delve into which databases do better with varying shapes of telemetry data and query patterns.

Telemetry can still be easier than many kinds of data storage and querying (sometimes)

Despite the implied complexity of telemetry data and that complexity's impact on database selection, there are two aspects of telemetry that make database selection easier:

1. You're almost always "insert-only" - i.e., never deleting individual records, just letting them expire "off the end". There are some potential complications here around GDPR compliance and "right to forget" user information, but even the most sophisticated shops rarely incorporate that kind of deletion into their operational telemetry systems.
2. Some kind of telemetry (in particular metrics) can be aggregated by "rolling it up". After a period of time, granular data is no longer relevant or helpful. At that point, rather than having data points every minute (or less), telemetry can be grouped into a single record that averages all the data for the hour. After a longer period of time, those hourly averages might be summarized to a single daily average. For readers who have a passionate dislike of averaging (I'm right there with you), the point isn't the specific summarization method I just described, but rather that there's a process by which hundreds or thousands of records can be reduced down to a single (or at least fewer) amount. When you do time-based aggregation like that, you lose detail and the ability to ask questions of it. But the point is that this type of summarization is only done after the point at which the more grandular data is helpful or necessary; AND summarization often comes with the ability to move the granular data to a separate (cheaper, slower) data store for safekeeping long term.

   For some more simple things like interface utilization, this is rarely an issue as you're almost always trying to get some max, average, or percentile for a given time period, and aggregates suffice.

With traffic data though, rolling up summaries of how many bytes, when, where, and/or by what protocol can severely limit your ability to peer into the traffic because you'd only see the dimensions you rolled up. For example, if you only have rollups of top talkers by source IP, protocol, or source port and want to investigate what protocols a given source IP address was speaking on, you'd need a rollup combining source IP and protocol. Without the foresight to do that, you'd be out of luck.

Database Options

(Traditional) Relational Databases

Traditional relational databases like MySQL, Postgres, and Microsoft SQL Server are very flexible and can be used - in theory - for most kinds of telemetry, but - in practice - are not well suited for many of them at modern scales.

Relational databases are tempting to use because everyone has them running already and they'll let you run almost any kind of raw query (besides graph primitives) and also unlike some more modern hipster databases, can easily modify and delete data. And even with the databases which have the ability to run full-text search style queries, they're often not very performant, even with specialized indices.

NOTE

(connected to "modern hipster databases")

OOOOOHHH!! SHOTS FIRED!!

(or)

No you didn't!

- Leon

Yes I did.

- Avi

Unless configured to effectively be columnar data stores (more on this below), traditional relational databases have some major disadvantages when storing and querying the trillions of rows of data (no, we're not exaggerating) that you will get over time from modern network observability telemetry sources. Storing data row-by-row means that compression is limited, indexing overhead can be high enough to slow down queries or even block ingesting for seconds or even minutes, and without some fairly sophisticated partitioning, rotating out or deleting old data can also be very painful.

In the network world, it's generally better to use relational databases for metadata; for logs and events if they represent a very low volume; and potentially for network metrics like SNMP, but only if your query volume is limited - meaning you query on a weekly basis, as opposed to having dashboards reloading every second. Traffic data, high volume metrics, and high volume logs are better stored in other types of databases we'll talk about below.

Table 2-1. Summary Table

Feature	RDBMS Strength	RDBMS Limitation
Querying	Rich SQL support, joins, constraints	Slow for time-series and analytical workloads
Ingest Performance	Good for moderate workloads	Poor at bulk high-volume writes
Schema Design	Enforces structure and integrity	Inflexible for evolving or nested telemetry
Scalability	Fine for small/medium environments	Lots of effort to scale horizontally
Time Series Support	Possible with plugins (e.g., TimescaleDB)	Not native in most relational engines
Columnar Support	Possible with plugins for columnar storage	Not built-in, limiting compression and scanning speeds

When to Use Relational Databases for Network Telemetry.

- You're building a **metadata-rich** telemetry platform that requires **joins, lookups, and constraints**.
- Your use case focuses on **auditing, compliance, or configuration changes**, not massive streaming data.
- You operate at a **modest telemetry scale** (e.g., <10K inserts/sec) and need strong transactional guarantees.

When to Be Cautious.

- You need to ingest **millions of records per second** from real-time telemetry (e.g., NetFlow, sFlow, logs).
- You need **efficient rollups, aggregation, or historical trend analysis** across time windows.

- Your telemetry schema is **frequently evolving** or semi-structured (e.g., JSON logs, flow data).

Metrics/TSDB

Time series databases (TSDBs) have evolved to the point where they have become their own type of database, with most of that rapid evolution occurring in the last 15 years. Older systems like MRTG and RRDtool (which rose to prominance in the network monitoring space) started the concept of accumulating "rolled up" summaries of timestamped events. Some of these components, especially RRDtool, are still around in many enterprises.

In the last decade, more generic time series databases - like Prometheus, InfluxDB, and VictoriaMetrics - have taken over for network telemetry, and support large-scale time series data, with the caveate that the data in question should, ideally, not be very "wide" - closer to an SNMP IFMIB data point at a specific time from a specific device, along with some associated device tags; than to a traffic/flow record's worth of data "stuffed" into a metric report, with flow values represented as metric tags.

TSDBs generally fall down with regard to high cardinality (number of unique values). Unfortunately, cardinality limits can bite you (and your database) across not one, but three dimensions:

- Number of unique series (short example here)
- Number of unique values for each series (i.e., number of IP addresses)
- Tags and number of combinations of tags on each series

If you anticipate very high cardinality ("very high" can depend on the system, but it's on the order of tens to hundreds of millions of unique series; or hundreds of millions or more of unique values), then you may want to explore trying to store metrics data in column store databases that have fewer cardinality limits - but may be slower and more resource intensive for running dozens or more parallel queries. And modern dashboarding systems can sometimes instantaneously generate hundreds of such queries per active user per minute.

Table 2-2. Summary Table

Feature	TSDB Strength	TSDB Weakness
Data Model	Time-stamped metric storage	Poor handling of relational data
Performance	High ingest rate, optimized reads	High-cardinality + scaling challenges
Storage	Efficient compression, downsampling	Some TSDBs cause high IOPS when periodically compacting data
Query Language & Alerts	Time-based queries, native alerts	Limited complex analytics (e.g., joins)
Visualization & Integration	Native Grafana visualization support, ecosystem tools	Forming arbitrary relational-type queries can be tricky or impossible

When to Use a TSDB for Network Telemetry. With all of that said, let me break down exactly when a TSDB might be your best choice:

- You need real-time metrics, alerting, and short- to mid-term analytics.
- You are operating at scale but with bounded cardinality (e.g., per-interface or per-device metrics).
- You want tight integration with monitoring stacks like Grafana, Prometheus, or Telegraf, where it can be helpful to at least send summaries in metrics form of more complex network telemetry stored in other systems.

What TSDBs Aren't AS Good At. Conversely, here are the TSDB contra-indications, or times when a TSDB is not your best option:

- Your telemetry involves high-cardinality dimensions like traffic data (e.g., per IP - even IPv4 is 4B per source and dest IP), per flow, or even per src/dst port combo (that is 4B potential as well).
- You need rich contextual joins or long-term historical queries.
- Your environment demands high availability with complex query workloads (you may need hybrid storage or data lake integration).

Columnar

Columnar databases have risen to prominence in the last decade and a half, and became popularized for telemetry and analytics with

Google's paper on their Dremel database. Most enterprise "Data Lakes" or "Lakehouses" are now built on top of columnar databases.

Columnar systems look a lot like relational databases in terms of shape of the data and queries possible, but store data column by column instead of row by row, which has two key advantages:

1. The data is much more compressible on disk because columns often have more similar (and thus more compressible) data; and
2. Much less data needs to be read because only the columns of interest are retrieved

Self-run systems like DuckDB, Presto, and Clickhouse and cloud databases like RedShift and BigQuery are leading columnar alternatives.

Columnar databases are generally the best place to store network traffic data, unless you need the ability to run full text queries. For metrics (like SNMP and Streaming Telemetry), columnar systems can work, but will generally be slower since they don't have built-in rollup support necessary when you have many users loading complex dashboards with expectations of response times in seconds (or less).

Table 2-3. Summary Table

Feature	Columnar DB Strength	Columnar DB Limitation
Query Performance	Fast analytics on large datasets	Can be slower for simple time series queries because most lack built in aggregation/rollup primitives, or support for millions+ of materialized (cached) views
Storage Efficiency	Excellent compression, even with high cardinality	Write amplification, especially with small batches
Schema Flexibility	Supports joins, SQL, metadata enrichment	Does not support stored procedures; often does not support full SQL semantics
Ingest Pattern	Good for batch inserts	Not optimized for few-at-a-time inserts, work very poorly or not at all for delayed data
Historical Analysis	Ideal for long-term telemetry retention	-
Ecosystem	Integrates with data lakes, BI tools	-

When to Use Columnar Databases for Telemetry.

- You want to store and analyze massive volumes of historical telemetry (e.g., months of NetFlow, sFlow, or other traffic data).
- You need to enrich telemetry with metadata (e.g., ASN, geolocation, device inventory).
- Your team has the data engineering expertise necessary to manage ingestion, schema, and partitioning.

When NOT to Use Columnar Databases (i.e., What They Aren't As Good At).

- You require low-latency alerts or real-time dashboards.
- You have no pipeline to transform telemetry into a batch/columnar format.
- Your telemetry use case is primarily operational, not analytical.

Logs / Event storage databases

If you are storing log messages where the expectation is that they will be regularly (if not frequently) parsed (searching for substrings and/or breaking out sub-elements for various calculations), storing them in relational databases (for low volume) or columnar databases can work well - though doing so often requires adding a parsing layer to extract the regular data.

But for more varied or arbitrary log data, relational and columnar databases are typically very slow at the kind of full text search required to operationalize these logs.

Leading log databases include Elastic, Victoria Logs, and Splunk. However, it should be noted that there are both alternatives AND emerging add-ons to columnar databases that try to provide fast full text search at least on particular columns.

For structured logs, an interesting option is Loki. While not the best for full text search, Loki integrates well with Prometheus to generate metrics from recurring patterns in logs - and allow retrieving back logs that contributed to those metrics.

Graph/topology/config

For those readers who were raised on relational database theory, noSQL can feel like it was created by free-love hippies but it's still recognizable as a database. Graph databases, on the other hand, can feel like M.C. Escher dropped acid and tried to re-create dBase III.

For readers who struggle with the concept of what a graph database is, just think about a mindmap tool (if you've used one).

Graph databases are designed to allow the query (and therefore display) of relationships between different objects. It should be noted that the underlying data might be stored in tables, but more often it's stored in a structure that is distinctly noSQL-esque. So maybe it's NOT as foreign a concept as it seems at first.

Some kinds of network telemetry, particularly topology, can be stored in many kinds of databases, but are more easily queryable in graph databases like Neo4J or Amazon Neptune.

Some simple queries ("what's immediately adjacent to node X") can be done in relational databases, but "what is the average distance between nodes in my network" or "what nodes got further away from my core yesterday" are much harder and usually require scripted or multi-step queries. This makes graph databases a logical choice.

Despite their flexibility and ease of use for topology questions, many organizations only add native graph databases after maturing most of the rest of their network telemetry observability systems.

Table 2-4. Summary Table

Feature	Graph DB Strength	Graph DB Weakness
Relationship queries	Fast traversals and multi-hop paths. Easy to query for changes over time	Learning curve for query languages
Real-time telemetry support	-	Integration with telemetry pipelines is not as strong
Flexibility	Schema-less, extensible	Slower to ingest/update large batches
Tooling/Integration	Graph visualizers and APIs are available	Less mature for flexible querying than SQL/relational tools

When to Use Graph Databases for Topology.

- Your use case involves multi-hop analysis, impact tracing, or route computation
- You operate layered topologies (e.g., physical + logical, tenant overlays, SDN paths)
- You need to answer questions like:

— “What devices are between A and B?”

— “What changed between devices A and B?”

— “What’s the blast radius if router X fails?”

— “Which services depend on this VLAN?”

When Not to Use a Graph DB Alone.

- You need to process large-scale telemetry or time-series data (flows, counters, metrics)
- You’re focused more on analytics than modeling
- Your topology is simple, relatively static, or easily represented in a relational model

When You Really Do Need Both (i.e., Hybrid Approach). Use a graph DB for topology and relationships, and integrate it with:

- Time-series DB for real-time and trended metrics
- Columnar DB for traffic logs and long-term analytics
- Relational DB for inventory, config, and business metadata

Streaming Database

Streaming databases like Kafka’s ksqlDB and AWS’s Kinesis use different techniques that take way less RAM and CPU to process large amounts of data, and can generate results that look like relational database queries with massively less effort than relational databases.

The two main trade-offs are that the results are generally approximate (think 99.9% but still not 100%), and you have to know the queries in advance - you can’t go back in time and ask a different question because any data that’s not part of an existing query result is immediately thrown out.

This makes streaming databases useful as a technique for processing network telemetry when you have a stable set of questions, want to be very efficient, and effectively can work with a static set of rollups. Because most of the live data is thrown away it’s actually often more efficient - and a lot more flexible - than TSDBs at rolling up data.

A note on Kafka: Kafka and other streaming systems are usually a key part of modern enterprise observability systems and telemetry buses. We're talking here about ksqlDB, not Kafka's traditional data queueing/storage. While the full breadth of a Kafka implementation does have the ability to store (and therefore query) old data, please don't use this except for replaying raw data streams to components that have failed or are restarting. It's very un-performant to use it as a historic data store and query it like was one, and can actually create issues by interfering with live ingest of data (i.e., dropped telemetry).

Key-Value Stores

Another database option is key/value store databases like Redis or DynamoDB, which associate a value with various keys, and they do so generally very flexibly. Some of these systems sit underneath other databases but here we're talking about using them directly.

Key-value stores are not great for most types of telemetry. They can be useful for metadata, but lack relational primitives that would make querying and auditing metadata (i.e. interface name for a router interface, to be used to join with NetFlow data) useful. However, it's usually a better pattern to store metadata in relational databases.

Document Databases

Document databases like MongoDB, Couchbase, and others are jacks of all trades, and you can use them for raw telemetry or metadata, but converting data to JSON, storing, and querying it that way is rarely the most efficient way to handle the raw telemetry and it's generally also more approachable to store metadata in relational or other databases.

NOTE

"Rarely the most efficient"?!? Converting data to ASCII format and then using Java to process it is about the most un-green thing you could think of doing, computationally, short of building a whole new datacenter with only copper wiring, cooling it with freon, building a nuclear power plant next door for electricity, and then letting the reactor melt down.

Chapter Summary

You made it to the end of a very detailed, VERY intense section. Unless you are a particular type of monitoring and observability groupie, you are probably feeling a little anxious, if not overwhelmed. You might be asking how much of the previous chapter you, personally, are going to have to deal with.

First, take a deep, cleansing breath.

Second, understand that Avi and I included this chapter NOT because we feel everyone who uses observability will need to do these things; but rather because we felt it was important for you to know what is involved in network observability, what might (or might not) be happening under the hood of the tools you use or are considering.

We wanted you to be fully informed consumers, not just of network observability tools, but of network observability data itself. And a big part of being informed is understanding all the things that (might or might not have) happened to your data between the moment of collection and the moment of query.

About the Authors

Avi Freedman and **Leon Adato** have, collectively, over 70 years experience in the tech industry, with particular focus on networking, monitoring, and observability. Both recognize that, after the hard work of building a solution is done - whether that be a network, a datacenter, or an application - the hard work of keeping things running starts. And that's usually where the problems really start. Their decision to collaborate on this book arose first and foremost to share all the samples, examples, stories, and lessons they usually share in the booth at conferences, or in talks, or when helping customers; but also to provide a resource to the readers themselves: who might need to articulate those same lessons to colleagues, managers, or the odd (*very* odd) person at a dinner party.