Evolvable Network Telemetry at Facebook

Yang Zhou[†] Ying Zhang[‡] Minlan Yu[†] Guangyu Wang[‡] Dexter Cao[‡] Eric Sung[‡] Starsky Wong[‡]
[†]*Harvard University* [‡]*Facebook*

Abstract

Network telemetry is essential for service availability and performance in large-scale production environments. While there is recent advent in novel measurement primitives and algorithms for network telemetry, a challenge that is not well studied is *Change*. Facebook runs fast-evolving networks to adapt to varying application requirements. Changes occur not only in the data collection and processing stages but also when interpreted and consumed by applications. In this paper, we present PCAT, a production change-aware telemetry system that handles changes in fast-evolving networks. We propose to use a change cube abstraction to systematically track changes, and an intent-based layering design to confine and track changes. By sharing our experiences with PCAT, we bring a new aspect to the monitoring research area: improving the adaptivity and evolvability of network telemetry.

1 Introduction

Network telemetry is an integral component in modem, large-scale network management software suites. It provides visibility to fuel all other applications for operation and control. At Facebook, we built a telemetry system that has been the cornerstone for continuous monitoring of our production networks over a decade. It collects device-level data and events from hundreds of thousands of heterogeneous devices, millions of device interfaces, and billions of counters, covering IP and optical equipments in datacenter, backbone and edge networks. In addition to data retrieval, our telemetry system performs device-level and network-wide processing that generates time-series data streams and derives real-time states. The system serves a wide range of applications such as alerting, failure troubleshooting, configuration verification, traffic engineering, performance diagnosis, and asset tracking.

While our telemetry system can adopt algorithm and system proposals from the research community (e.g., [18, 27, 48, 50]), a remaining open challenge is *Change*. Changes happen frequently in our network hardware and software to meet the soaring application demands and traffic growth [16]. These changes have a significant impact on the network telemetry system. First, we have to collect data on increasingly heterogeneous devices. This is exaggerated as we introduce in-house built FBOSS [13], which allows switches to update as frequently as software. Second, we have growing applications (e.g., [1]) that rely on real-time, comprehensive, and accurate data from network telemetry systems. These applications introduce diverse and changing requirements for the telemetry system on the types of data they need, data collection frequency, and the reliability and performance of collection methods.

The changes this paper considers include not only the network events from the monitored data, but also those updates to the telemetry system itself: modification to monitoring intent, advance of device APIs, adjustment of frequency configurations, mutation of processing, and restructure of storage formats. Without explicitly tracking them in our network telemetry system, we struggle to mitigate their impact to network reliability. For example, a switch vendor may change a packet counter format when it upgrades a switch version without notifying Facebook operators. This format change implicitly affects many counters in our telemetry database (e.g., aggregated packet counters), leading to adverse impact to downstream alerting systems and traffic engineering decisions. This example highlights several challenges: (1) Production telemetry is a complex system with many components (e.g., data collection, normalization, aggregation) from many teams (e.g., vendors, data processing team, database team, application teams). A change at one component can lead to many changes or even errors at other components. As a result, when telemetry data changes, it is difficult to discern legitimate data changes from semantic changes. (2) Sometimes, we only detect the error passively when traffic engineering team notices congestion. Yet, we cannot diagnose it easily because the error involves many data. Even worse, it may only affect a small portion of vendor devices due to phased updates. Section 2 shares more such examples.

In this paper, we propose to treat changes as first-class citizens by introducing PCAT, a Production *Change-Aware* Telemetry system. PCAT includes three key designs:

First, inspired by the database community [8], we introduce the *change cube* abstraction for telemetry to explicitly track the time, entities, property, and components for each change, and a set of primitives to explore changes systematically. Using change cubes and their primitives, we conduct the first comprehensive study on change characterization in a production telemetry system (Section 3). Our results uncover the magnitudes and the diversity of changes in production, which can be used for future telemetry and reliability research.

Second, we re-architect our telemetry system to be change-aware and evolvable. In the first version of our telemetry system, we have to modify configurations and code at many devices every time a vendor changes the counter semantics or collection methods, or an application changes monitoring intents. To constrain the impact of changes, i.e., the number of affected components, PCAT includes an intent-based lay-

ering design (Section 4) which separates monitoring intents from data collection and supports change cubes across layers. PCAT enables change attribution by allowing network engineers with rich network domain knowledge to define intents while having software engineers building distributed data collection infrastructure with high reliability and scalability. PCAT then compiles intents to vendor-agnostic intermediate representation (IR) data model, and subsequently to vendor-specific collection models, and job models. The intent-driven layering design reduces the number of cascading changes by 54%-100%, and enables systematically tracking changes through the monitoring process.

Third, we build several change-aware applications that explore the dependencies across change cubes to improve application efficiency and accuracy. For example, Toposyncer is our *topology derivation* service that builds on telemetry data and serves many other applications. We transformed Toposyncer to subscribe to change cubes based on derivation dependencies and greatly reduce topology derivation delay by up to 118s. We leverage correlation dependencies across change cubes to enable troubleshooting and validation.

The main contribution of this paper is to bring the community's attention to a new aspect of telemetry systems—how to adapt to changes from network devices, configurations, and applications. We also share our experiences of building change-aware telemetry systems and applications that can be useful to other fast-evolving systems.

2 Motivation

To keep up with new application requirements and traffic growth, data center networks are constantly evolving [16]. As a result, changes happen frequently across all the components in telemetry systems, ranging from device-level changes, collection configuration changes, to changes in the applications that consume telemetry data.

Our first generation of production telemetry system was not built to systematically track changes. This brings significant challenges for telemetry data collection at devices, integration of telemetry system components, debugging network incidents, and building efficient applications. In this section, we share our experiences of dealing with changes in our telemetry system and discuss the system design and operational challenges for tracking changes.

2.1 Bringing changes to first-class citizens

We motivate the needs of treating changes as first-class citizen in network telemetry with a few examples.

1. Build trustful telemetry data. Many management applications rely on telemetry data to make decisions. However, in production, telemetry data is always erroneous, incomplete, or inconsistent due to frequent changes of devices and configurations. Moreover, there are constant failures in large-scale networks (e.g., network connection issues, device overload, message loss, system instability). Therefore, applications need to know which time range and data source are trustful and how to interpret and use the data. This requires tracking changes for each telemetry data value and semantics.

For example, we collect device counters at various scopes (e.g., interfaces, queues, linecards, devices, circuits, clusters). These counters may have different semantics with device hardware and software upgrades or network re-configurations. For example, we have a counter for 90th percentile CPU usage within a time window of a switch. When we change the switch architecture to mulitple subswitches [13], we set the counter as the average of 90th percentile CPU of subswitches. However, our alert on this counter cannot catch single subswitch CPU spikes that caused bursty packet drops. We need to know when to change the alerts based on counter changes.

2. Track API changes across telemetry components. Our telemetry system consists of multiple data processing components, which are independently developed by different vendors and teams. When one component changes its interfaces, many other components may get affected without notice. There are no principled ways to handle such changes across telemetry components. For example, vendor-proprietary monitoring interfaces often get changed without an explicit notification or detailed specification. This is because telemetry interfaces are traditionally viewed as secondary compared to other major features. However today cloud providers heavily rely on telemetry data for decisions in a fine-grained and continuous manner. If we do not update data processing logic based on device-level changes, the inconsistency may cause bugs and monitoring service exceptions.

In one incident, a routing controller had a problem of unbalanced traffic distribution, caused by incomplete input topology: a number of circuits were missing from the derived topology. This took the routing team and the topology team over three days to diagnose. The root cause was an earlier switch software upgrade that changed the linecard version from integer (e.g., 3) to string (e.g., 3.0.0). Such a simple format change was not compatible with the post-processing code that aggregated the linecard information into a topology. Thus, we missed several linecards in the topology, which then mislead TE decision and cause congestion in the network. This is not a one-off case, given many vendors and software versions coexist in our continuously evolving networks.

3. Debug with change-aware data correlation. As telemetry components keep evolving, it is hard to attribute a problem to a change using data correlation without explicitly tracking changes and their impacts. For example, when we fail to get a counter, the problem can come from data collection at the device, the network transfer, or both.

In production, we make changes in small phases: first canary on a few devices serving non-critical applications, then gradually on more devices to minimize disruptions to the network [13]. In one incident, there were a small number of devices with "empty data" errors for a power counter. The errors increased gradually and ultimately went beyond 1% threshold

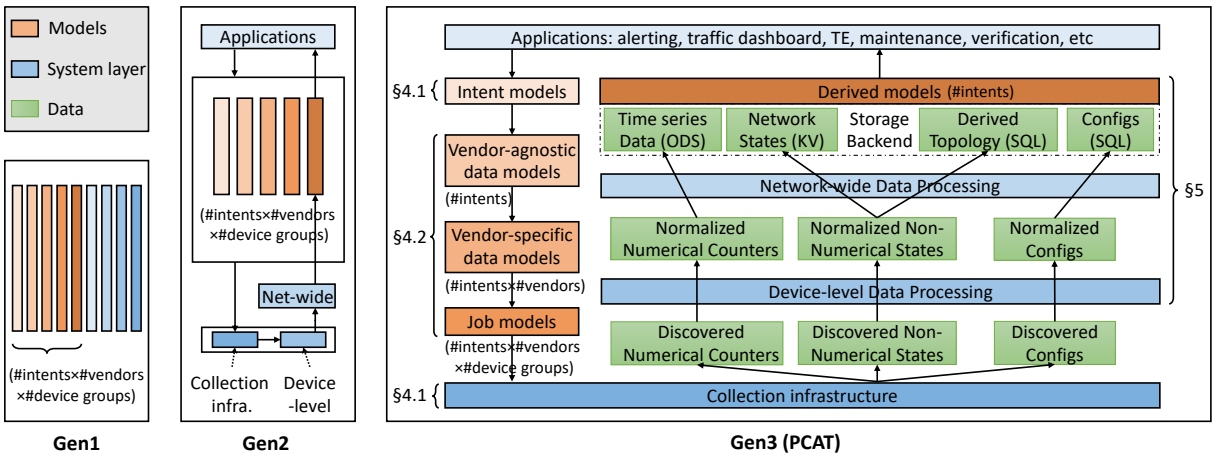


Figure 1: Generations towards change-aware telemetry.

after two weeks and triggered an alarm. This problem was difficult to troubleshoot due to its small percentage. We manually explored the changes through correlation: checking whether there were code changes before the failure, whether the failed devices shared a common region (indicating regional failure), a common vendor, or on common data types. We tried many dimensions of correlation and finally found the errors were mostly related to power and environment counters. The root cause was a vendor changing its format but the processing code could not recognize it. This example shows a tedious manual process of data correlation to debug problems because of gradual change rollouts. To improve debugging, we need to use changes to guide data correlation.

2.2 Lessons from Previous Generations

We now discuss our previous two generations of telemetry systems prior to PCAT and their limitations in handling changes.

Gen1: Monolithic collection script. In a nutshell, a telemetry system is a piece of code that collects data using APIs from the devices. Our first generation is naturally a giant script that codifies what counters to collect. It hardcodes the collection method, polling frequency, post-processing logic, and where to store the data. Figure 1 illustrates Gen1 as intertwined models and system blocks. It runs as multiple cron jobs, each collecting data from different groups of devices. This design is intuitive to implement but is not change-friendly. If a vendor changes the format of a counter, we need to sweep through the entire script to change the processing logic accordingly, and redeploy the new code to all monitors. It has high maintenance burdens as it relies on expert's deep understanding of the code to make changes. Further, tracking changes relies on version control system in the form of code differences, which do not reveal the intent directly.

Gen2: Decoupled counter definition from collection process. As our network expanded, the hulking script in Gen1 became hard to manage. We moved to Gen2, which separates the monitoring model (i.e., what counter to collect) from the actual collection code, shown as orange and blue boxes in Figure 1. The separation allows us to track changes to data types separately from the collection logic (e.g., sending requests, handling connections). However, the intent is still mingled with the vendor-specific counter definition. For instance, one may want to collect the "packet drops per interface". One needs to specify the exact SNMP MIB entry name and the specific API command. A low-level format change would result in updates on all model definitions. Moreover, the data collection system includes both the collection infrastructure and data processing logic. The data processing logics scatter across many places, e.g., when the data is collected locally at the collector, or before it is put into the storage. To change a piece of processing logic, we have to change many such places, which is cumbersome to track. In addition, when a piece of data is changed or is absent, tracing back on what causes the change is manual and tedious.

2.3 Challenges and PCAT Overview

Our experiences of previous two generations indicate three main challenges in handling changes: change abstraction, attribution, and exploration. To address these challenges, we build our Gen3 telemetry system – PCAT.

Change abstraction. In Gen1 and Gen2, changes were not stored structurally. They exist either as diffs in code reviews in Gen1, or logs to temporary files in Gen2. Without a uniform representation, each application needs to develop ad-hoc scripts to parse each data source. This leads to not only duplicate efforts but also missing changes or mis-interpretations. A uniform and generic change abstraction allows hundreds of engineers to publish and subscribe to changes to boost reliable collaboration without massive coordination overhead. In §3, we propose a generic abstraction called *change cube* to tackle this challenge.

Change attribution. The second challenge is the turmoil to ascribe the intent of the scattering changes. The solution involves a surgically architectural change to a multi-layer design, shown in Figure 1 and elaborated below.

Data collection. The first step is to collect data from de-

vices, called *discovered data*. There are three types: numeric counters, non-numeric states, and configurations (see Table 4 in Appendix). We use different protocols for collecting different data and for different devices: SNMP [10], XML, CLI, Syslog [28], and Thrift for our in-house switches [13].

Device-level data processing (normalization). The data is different in formats and semantics across devices, vendors, and switch software. This makes it difficult for applications to parse and aggregate the data from different devices. We use a device-level data processing layer to parse the raw data to a unified format across devices, vendors, and switch software.

Network-wide data processing. Next, we aggregate device-level (normalized) counters, states, and configurations into network-wide storage systems for applications to query. The normalized non-numerical states (as network states) are stored in a key-value store. We build a tool called **Toposyncer** which constructs *derived topology* from normalized non-numerical states. For example, from per-device data, we can construct the device, its chassis, linecard, as well as cross-device links.

Data consumer applications. There are many critical network applications that consume PCAT data. Network health monitoring and failure detection use monitoring data to detect and react to faults. Network control relies on real-time data for making routing and load balancing decisions [2, 38]. Maintenance and verification use telemetry data to compare network states before and after any network operations.

There are several advantages of the new design compared to previous generations. First, compared to Gen2, Gen3 dissects a monolithic data definition into three different types, each focusing on defining one aspect of the monitoring. The separation brings better scalability and manageability. We describe the details in §4. Second, we not only care about tracking changes in data format and code, but also need to attribute changes to the right teams (i.e., who/what authored the change). Change attribution builds the trust of the data for applications. It facilitates collaboration across teams towards transparent and verifiable system development. Gen3's intent-based layering design lets each team play by their strength and work together seamlessly. Specifically, the network engineers can leverage their rich domain knowledge and focus on intent definition, while software engineers focus on scaling the distributed collection system.

Change exploration. Many designs and operations require a clear understanding of the relations amongst changes. For example, to debug why a piece of data is missing, we always find the last time the data appears and check what has changed since then. We may find one change to be the cause, which could be caused by another change somewhere else. Similarly, when receiving a change of an interface state, we need to reflect the change on the derived topology and upper-layer applications. It motivates us to develop primitives for change exploration that serves many applications. We demonstrate the usage in real-time topology derivation in §5.

3 Changes in Facebook Network Telemetry

In this section, we define the change cube concept and explain how they are generated in this system, together with extensive measurement results by composing queries on top of the change cubes.

3.1 Change Cube Definition

To systematically handle changes in network telemetry, we leverage the concept of *change cubes*. Change cubes are used in databases [8] to tackle the data change exploration problem by efficiently identifying, quantifying, and summarizing changes in data values, aggregation, and schemas. Change cube defines a set of schemas for changes and provides a set of query primitives. However, changes in network telemetry are different from those in databases in two aspects: (1) Network telemetry generates streaming data with constant value changes, so the change cubes in network telemetry do not care about value changes but only changes in schema and data aggregation. (2) Network telemetry has frequent changes due to fast advances of hardware and software that result in data semantics changes.

Change cubes. We define a change cube to be a tuple $< Time, Entity, Property, Type, Dependency >$. We summarize each field of the change cube in Table 1 and explain below.

- *Time* dimension captures when the change happens. It depends on the granularity we detect changes, e.g., seconds, minutes, or days.
- *Entity* represents a measurement object, e.g. a switch, a linecard, as well as the models that describe what to measure and how.
- *Property* contains the fields or attributes of the entity that get changed. For example, a loopback IP address of a switch, an ingress packet drop of an interface.
- *Layer* dictates the layer or component in the telemetry system (in Figure 1) where changes happen. We discuss how we land in these choices in §4.
- *Dependency* dimension contains a list of other changes that this change is correlated with. Each item in the list is a $< ChangeCube,\ Dependency\ Type >$ pair. We support two dependency types: correlation dependency and derivation dependency. Derivation dependency means that a lower-layer change causes an upper-layer change. Correlation dependency means two changes on correlated entities or properties.

Primitives on change cubes. Next, we introduce the operators on the change cube, which are used to explore the change sets. We leverage the operators proposed in [8] but redefine and expand them in the context of telemetry systems.

- $Sort_f(C)$ applies function f to a set of change cubes C, on one or a few dimensions to a comparable value, and uses it to generate an ordered list of C. In our problem, sort is mainly used with time to focus on the most recent changes.
- $Slice_p(C)$ means selecting a subset of C where the predicate

Dimension	Sub category	Examples
Time	Multiple time granularities	Second, minute, hour, day
Entity	Intent model	High-level intent, e.g., packet drops at spine switches
	Vendor-agnostic data model	Counter scope, unit
	Vendor-specific data model	Format, API
	Job model	Collection channel, frequency, protocol
	Derived model	Derived network switch
Property	Model fields	IP address, network type
	Change attributes	LoC, reason
Layer	Application	Adding alert to detect a new failure type
	Network-wide processing	Topology discovery code logic
	Device-level processing	Normalization rule
	Collection infrastructure	Codebase for collection tasks
Dependency	Correlation dependency	BGP session and interface status
	Derivation dependency	Circuit is derived from two interfaces' data

Table 1: Change Cube Definition.

p is true. It is used to filter an entity or a property value.

- $Split_a(C)$ partitions C to multiple subsets by attribute a. An example is to split the changes by the layer to group changes according to where they occur. A reverse operator to *Split* is *Union*, which combines multiple change sets.
- $Rank_f(P_C)$ After we split C to multiple sets P_C, we further analyze these sets and rank them based on a function, e.g., cube size, the time span, the volatility.
- $TraceUp(c)$ and $TraceDown(c)$. These two operators are used with the *Dependency* field, which are new compared to [8]. The former traces the changes that the current change c depends on, and the latter traces the changes that depend on the c. They are useful for debugging through layers and validation across data.

Explicitly tracking changes in a structured representation eases the diagnosis process. Considering the second example in §2.1, when the switch software is updated, it populates a change cube to the database, indicating the API's return result has changed. Consequently, it triggers another change cube at the counter model level on this specific CPU counter. This change cube in turn propagates through the monitoring stack to job changes and retrieved data changes. The applications using the CPU counters can subscribe to such data change, which can then be notified immediately. The chain of change notifications eliminates the post-mortem debugging after the counter change causes application errors.

3.2 Changes in PCAT

Leveraging *change cubes*, we provide the first systematic study of changes and their impact on network telemetry systems. We populate change cubes of PCAT using multiple ways. For the data stored in database, we leverage our database change pub/sub infrastructure [39]. We subscribe to the telemetry objects' change log and translate them to change cubes. For code changes in collection infrastructure, data processing logics (both device-level and network-wide), and applications, we parse the logs in the code version control system to generate change cubes. Intent model, data model (both vendor-agnostic and -specific), and job model changes are codified and thus tracked through code changes [41]. They can be populated using the same way as other code changes. We store all change cubes to a separate database called *ChangeDB* and develop APIs to explore these changes.

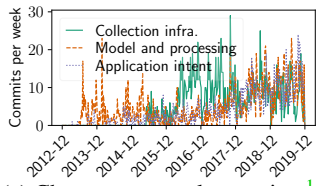
Queries	Formulas
Q1 (Fig. 2a)	$Sort_{Time(Week)}(Slice_{layer=\text{“application”}}(C))$
Q2 (Fig. 2a)	$Sort_{Time(Week)}(Slice_{entity=\text{“vendor-agnostic data model”}}(C))$
Q3 (Fig. 2c)	$\sum_c c.LoC, c \in Split_{Time(Week)}(Slice_{layer=\text{“application”}}(C))$
Q4 (Fig. 3a)	$Sort_{Time(Day)}(Slice_{entity=\text{“job model” \& property=“frequency”}}(C))$
Q5 (Fig. 3b)	$Split_{network\ type}(Slice_{entity=\text{“job model” \& property=“frequency”}}(C))$
Q6 (Fig. 3c)	$Split_{network\ type}(Slice_{entity=\text{“job model” \& property=“channel”}}(C))$
Q7 (Fig. 4a)	$Split_{blueprint\ type}(Slice_{layer=\text{“application” \& reason=“blueprint”}}(C))$
Q8 (Fig. 4b)	$Split_{vender}(Slice_{layer=\text{“application” \& reason=“new model”}}(C))$

Table 2: Queries used in §3.2.

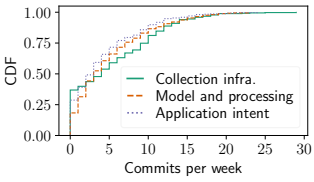
We analyze changes from the perspectives of devices, collection configurations, and application intents, over seven years (2012-2019). Our results below uncover surprisingly frequent changes and quantify the diverse causes of changes.

3.2.1 Change Overview

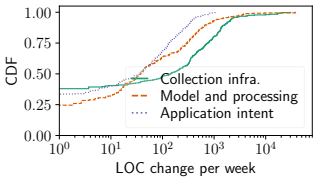
Change frequency. We first quantify the code changes of our monitoring system. We map one code commit to one change cube, involving multiple lines of code across multiple files. We group the changes into three categories according to where they happen in Figure 1: collection infrastructure (bottom layer), data & job models and processing (middle two layers), and applications, representing the infrastructure, data, and intent respectively. We construct queries using the primitives defined earlier. We put the actual query to generate the figures in Table 2. Q1 uses *Slice* to filter the changes in application layer, and sorts the changes by time. We replace the "application" with other values for changes in other layers. Q1



(a) Changes per week over time[1].



(b) Number of changes.

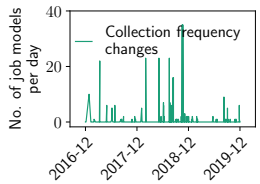


(c) Magnitude of changes.

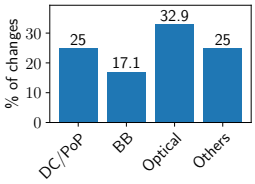
Figure 2: Change characteristics.

Change reasons	%
Collection infrastructure	67.9
Adding new devices	17.8
Topology processing	8.30
Data format	4.86
Counter processing	1.19

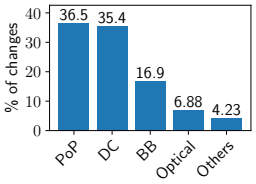
Table 3: Change categorization by change reasons.



(a) Freq. over time.

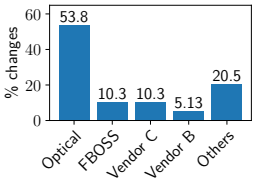


(b) Collection frequency.

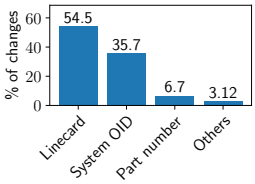


(c) Mngt. channel.

Figure 3: Collection configuration changes[2].



(a) Add new data type.



(b) Blueprint.

Figure 4: Network configuration changes.

can be compiled into the following SQL: `SELECT COUNT(*) FROM ChangeDB WHERE layer = "application" GROUP BY time_week ORDER BY time_week.`

Figure 2a shows the number of changes per week. We find that *types of changes vary greatly as the telemetry system scales*. More model and processing changes occur at the beginning (the year of 2013), as we begin by adding more counters to monitor. When the number of counters reaches a certain scale (the year of 2016), we realize the infrastructure needs better scalability. Thus there are more changes to refactor the collection infrastructure. Application intent follows the same trend as data changes, as adding new data is often driven by the need from applications.

Cumulatively across time, we show the average numbers of weekly changes of three categories in Figure 2b. They are on the same order of magnitude, with slightly more infrastructure changes. It can be as high as 25-30 changes per week. Note that each change is deployed on many switches and the changes it introduces to the network is significant.

Change magnitude. We quantify the magnitude of changes in terms of Lines of Code (LoC) using query Q3. While most code changes are not big, some changes could touch multiple lines due to consolidation of processing logic and refactoring. This is obtained by first getting a slice of changes of a given category, splitting the changes into weeks, and summing up the LoC property. Figure 2c shows that *collection infrastructure has larger changes and the application has changes with larger volatility*. We can dig into the volatility of each change set by computing its variance and use the *Rank* primitive. Both figures show there exists a significant number of large changes. For example, there are 27 weeks with more than 1000 LoC changes for collection infrastructure. However, as the industry's trend is to move away from monolithic changes to many small incremental changes [13], we expect to have more frequent small changes going forward.

Change reason categorization. We analyze the breakdown by reason of change using $Split_{reason}(C)$, which is obtained by parsing the commit log text and adding it to the ChangeDB. Table 3 shows that one major reason is collection infrastructure changes (67.9%). Adding new devices to the network is the second dominant reason (17.8%). Topology processing changes occupy 8.3%. The fourth reason is adjusting the data formats of collection models (4.86%). Lastly, 1.19% come from the device-level counter processing code.

3.2.2 Device-Level Changes

In our large-scale networks, we constantly add new vendors and devices to leverage a rich set of functions and to minimize the risk of single-vendor failures. The number of devices increased 19.0 times and the number of vendors increased 4.7 times as observed by PCAT in six years. Even with the same vendor, we gradually increase the chassis types, which have different combinations of linecard slots and port speeds. More choices of chassis types allow us to have fine-grained customization to our network needs. Furthermore, the number of chassis types grows from 26 to 129 (4.4×). In addition, our in-house software switch has tens of code changes daily and deploys once every few days [13].

3.2.3 Collection Configuration Changes

Collection frequency. Applications adjust the collection frequency to balance between data freshness and collection overhead. We first analyze collection changes by counting daily changes of collection frequencies over time, using query Q4. Figure 3a shows that there are constant collection frequency changes over time, with more frequent changes near December 2018 – because of tuning collection frequencies for newly-added optical devices. We analyze collection frequency changes by applying *Slice* on both the entity and the property. Interestingly, Figure 3b shows that *optical devices*

[1]The collection and processing infrastructure were not merged into the codebase before 2015-04; so its commits are non-trackable before that.

[2]The "Other" contains some changes that are hard to classify programmatically. The same applies to Figure 4.

change frequency more often (32.9%) because they cannot sustain high-frequency data polling and thus require more careful frequency tuning.

Management channel changes. PCAT collects data from the management interfaces at devices. As our management network evolves, we frequently reconfigure management interfaces (e.g., IP addresses, in-band vs. out-of-band interfaces). Backbone and PoP devices have multiple out-of-band network choices for high failure resiliency. Figure 3c breaks the IP preference changes into PoPs, DCs, and Backbones. *PoPs have more frequent channel changes (36.5%)* because PoPs are in remote locations and thus have more variant network conditions. Selecting the right channel is important to keep the device reachable during network outages so that we can mitigate the impact quickly.

3.2.4 Application Intent Changes

Data type changes. PCAT supports an increasing number of diverse applications over years. Applications may add new types of data to collect (e.g., to debug new types of failures), or remove some outdated data. Figure 4a shows how different vendors add new data types. Optical device vendors add more data (53.8%) because we recently start building our own optical management software and thus need more counter types. Indeed, optical devices generally have more types of low-level telemetry data compared to IP devices, e.g., power levels, signal-to-noise ratio. They are also less uniformed across vendors than IP devices.

Hardware blueprint changes. Hardware blueprint specifies the internal components (chassis, linecards) of each switch and determines what data to collect. Figure 4b shows the percentage of changes for hardware blueprints such as linecard map, system Object Identifier (OID) map, part number map, and others (e.g., OS regex map). These changes are due to network operations such as device retrofit and migration. They may cause data misinterpretation if not treated carefully.

4 Change Tracking in Telemetry System

In this section, we describe the layering design of our current intent-based telemetry system to help track changes.

4.1 Towards change-aware telemetry

Intent modeling. We use a thrift-based modeling language that empowers network engineers to easily specify their monitoring intents. Compared to other intent language proposed in academia [19, 31, 32], our language puts more emphasis on device state in addition to traffic flows, and defines actions in addition to monitoring. Our language contains three components shown in Figure 5.

- *Scope* captures both the device-level scope (e.g., Backbone Router) and network-level scope, (e.g., DC fabric network).
- *Monitor* specifies what to monitor in a vendor-agnostic way. For example, an intent could be capturing packet discard for the gold-level traffic class, which will get translated to a specific SNMP MIB entry or particular counters. In the left part of Figure 5, we describe the SQL-like query language. The *keys* are monitored metrics and the *entities* are time-series data streams and discrete events tables. We also support data aggregation functions such as *avg*, *count*, *filter*, which aggregate samples over time and devices.
- *Action* includes two types: Emitter and Detector. Emitters subscribe to *discrete network events* that are pushed from devices, and define actions upon receiving these events. Detectors allow us to write formulas for various time-series data, and set up a threshold for the formula value as the alerting condition. A detector example is shown in the right part of Figure 5; it defines a detector based on the key *gold_class_discards* which captures the packet drops for gold-class traffic on a physical interface. The discard is transformed to rate, and aggregated every 60 seconds. The alert is triggered if the threshold is greater than 0.01.

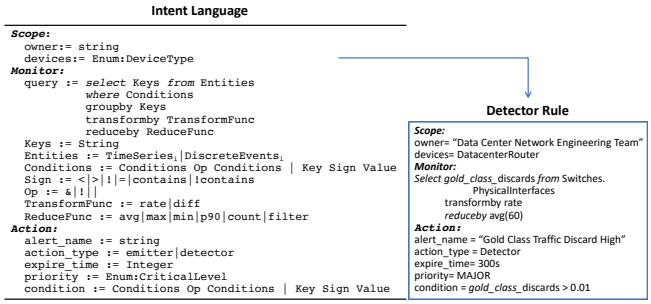


Figure 5: Intent model.

```
ModelDef(
  name='PhysicalInterface',
  properties=[PropertyDef(name='if_name',type=STRING,transform=NONE),
    PropertyDef(name='if_hc_in_octets',type=INT64,transform=RATE),
    PropertyDef(name='if_in_discards',type=INT64,transform=RATE),
    ...],
  children=[ModelDef(
    name='GoldQueueCounters',
    properties=[PropertyDef(name='queue_name',type=STRING),
      PropertyDef(name='packet_count',type=INT64),
      PropertyDef(name='byte_count',type=INT64),
      PropertyDef(name='packet_discards',type=INT64)]),
    ModelDef(
      name='SilverQueueCounters',
      ...),
    ...]),
  ...
```

Figure 6: Data model.

The intent model hides low-level changes. Vendors may change the queue drop counter names, or the mapping between queues and gold-class traffic may change. The intent configuration remains unchanged in both cases.

Runtime system. We handle heterogeneous intents with homogeneous software infrastructure. Thanks to separation, software engineers can focus on the runtime execution system to solve the hard system building problems: scheduling, load balancing, scalability, and reliability. The runtime execution system collects data from devices according to the model, which includes a distributed set of engines and a centralized controller to distribute jobs and collect data from these engines. The centralized controller fetches the latest collection and job models, combines with device information in our database, and generates a sequence of jobs to be executed

Figure 7: Collection method and job model.

with a given deadline. It dispatches jobs to designated engines based on load and latency. The engine executes the collection command, performs device-level processing, and sends data back to the corresponding storage. The system is heavily engineered to tackle the reliability and scalability challenges.

4.2 Change reduction w/ vendor-agnostic IR

Next, we zoom into the intermediate layer between the intent and the runtime. The high-level monitoring intent is translated to the intermediate representation data model, which gets mapped to the vendor-dependent collection model, and finally is materialized to the job model on each device. We emphasize that how the modular design principle is translated to different models in order to ***limit the impact of changes***.

Vendor-agnostic intermediate representation (IR) data model. The data model is created based on the *keys* field in the intent model. It specifies data schema in the following way, as shown in Figure 6.

- *Hierarchical.* We choose a tree structure as an intermediate representation, called the *model tree*. An example is shown in Figure 7. An `AggregateInterface` model has multiple child models, e.g., `PhysicalInterface`, `BGPSessions`. A `PhysicalInterface` also has multiple child models. Models are like templates waiting to be filled in. When they are materialized by actual monitoring data, we call them **objects**. By organizing the materialized objects in the same hierarchy as the model tree and adding a dummy root to connect up the top-level objects, we get a materialized **object tree**. The models define the data to be collected, which is derived from the `keys` field in the intent model.
- *Typed.* The data model defines the types of data to make interpretation of the data easier, e.g., `if_hc_in_octets` is the incoming traffic in octets.
- *Processing instruction.* It also defines basic processing primitives to go with the data using the *transform* field, e.g., computing a per-second rate from consecutive absolute counts. Both the type and the processing instruction are determined by the intent. Placing all the processing logic in a separated blob makes it much easier to track the changes in processing logic.

Vendor-dependent collection model. The IR model is further compiled down to vendor-specific counter names, specific commands to use, e.g., a CLI command, Thrift function name. Figure 7 shows two collection methods for the `GoldQueueCounters` data model: CLI and thrift. In each implementation, we define the collection API and the post-processing function in the *parser* field. We show an example of the CLI parser function that matches the regex in the output of a command on the vendor1 device. Creating this layer of model separately allows us a place to capture all changes due to vendor format and API changes, which are quite common.

Vendor-dependent job model. The job model combines the collection method with a concrete set of devices, shown in Figure 7. The *implementation* field matches with what is defined in the collection method. Instead of defining a job spec for each device, we group devices and apply the same job spec for all of them. Figure 7 uses *DeviceFilter* to define device role (e.g., rack switches), OS type, region, device state, etc. Job models are the input to the runtime execution to handle job scheduling and manage job completion. Job model captures the system aspects of changes. It can be adapted according to performance and scalability requirement, which is independently controlled from the intent or data specifications.

5 Change Exploration

Once PCAT collects data based on monitoring intents, we run device-level and network-level processing to report the data back to applications. Below, we build a few change-aware applications by exploring dependencies across change cubes.

5.1 Change-driven Topology Derivation

Toposyncer is our topology generation service, part of the collection infrastructure (see network-wide data processing in Figure 1). It creates *derived topology* from normalized device-level data (i.e., in vendor-agnostic format) (Figure 8). For example, from per-device data (e.g., interface counters, BGP sessions), Toposyncer constructs the device, its chassis, linecard, as well as cross-device links.

Toposyncer overview Toposyncer has four processes: *(1) Sync_device* constructs nodes with multiple sub-components: sub-switches, chassis, line cards (line 2-11 in Figure 9). It also derives device-level attributes such as power and temperature, control and management plane settings. *(2) Sync_port* derives physical and logical interfaces on each node and their settings (IP address, speed, QoS) (line 12-13). *(3) Sync_circuit* constructs cross-device circuits. A circuit is modeled as an entity with two endpoints, pointing to the interface of each end's router [41]. For each interface, it searches for all possible neighbors based on various protocol data, e.g., LLDP, MAC table, LACP table. In case some data source is incomplete, we search all data sources, independently identify all possible neighbors from each data source, and consolidate the results.

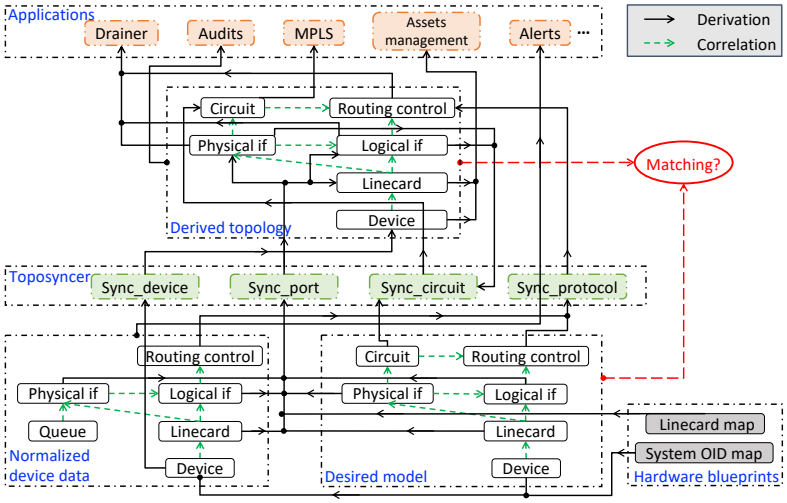


Figure 8: Data dependency graph. Toposyncer consumes the device data, desired model and hardware blueprints to generate derived data. PCAT verifies the derived data with the desired model.

```
1:  procedure DeriveTopology(Collection, Desired,
    HdwTemps, DependencyG)
2:      for d ∈ Desired.Devices do
3:          DeviceObj, dep = sync_device(Collection, d)
4:          DependencyG.add(dep)
5:          blueprint = getBlueprint(HdwTemps, d)
6:          for chassis_temp ∈ blueprint do
7:              derived_chassis = sync_chassis(Collection,
                chassis_temp)
8:              for linecard_model ∈chassis_temp do
9:                  derived_chassis.add(sync_linecard(Collection,
                    linecard_model))
10:             DeviceObj.addchassis(derived_chassis)
11:         DeviceObjs.add(DeviceObj)
12:     for d ∈ DeviceObjs do
13:         derived_ifaces = sync_port(Collection, d)
14:         for iface ∈ derived_ifaces do
15:             neighbors.add(findNeighbors(iface, Collection))
16:         circuits = sync_circuits(iface, neighbors)
17:         sync_protocol(Collection, circuits)
18: procedure UpdateTopology(UpdateQ, DependencyG)
19:     while UpdateQ ≠ ∅ do
20:         change_i = UpdateQ.pop()
21:         dependent_objs = change_i.Dependency
22:         update_func=findFunc(dependent_objs)
23:         update_func(dependent_objs, change_i)
```

Figure 9: Toposyncer algorithm

(4) Sync_protocol creates the protocol layers on top of the circuits, such as OSPF areas, BGP sessions and their states.

Toposyncer uses two additional data sources as templates to guide the construction: the desired model which defines the operator's intent topology and hardware blueprints which include hardware specifications, as shown at the bottom in Figure 8. Figure 9 shows the process. It uses desired device data (names, IP addresses) to decide what device to derive (line 2). Then, it uses the hardware blueprints and desired data to handle ambiguity. For instance, to figure out "what is this chassis", it first checks the discovered chassis name in raw data from the device. But often the discovered name is not uniquely mapped to a chassis but to several possible chassis versions, e.g., two versions with 4 linecards, one version with 8 linecards. Toposyncer cross-checks with hardware blueprints and picks the best match[3] (line 6-8). This process is similar to other topology services [29, 41], but we focus on derived models and how we populate them automatically from telemetry data.

Improve Toposyncer with change cubes. Our first implementation of Toposyncer did not utilize changes. It ran periodically against the latest snapshots of collected data at a fixed frequency (e.g., 15 minutes). This method leads to stale derived data, which affects the freshness and accuracy of upper-layer applications. Another challenge is debugging. When a piece of data (e.g., a circuit) is missing in the derived topology, it is hard to find out whether it is because of a raw data change, a normalized data change, a desired model change, a hardware blueprint change, or other reasons. We tackle these problems using change cubes and the dependency primitives below.

[3]When the guess is wrong, it exhibits as a discrepancy between desired and derived topology. We add alarming to detect such differences and involve humans to manually investigate.

Build change cubes. We generate change cubes for normalized data, desired model, hardware blueprint, as well as Toposyncer code changes, shown as each dotted box in Figure 8. We generate these cubes by parsing database transaction logs and model/code changes from version control system logs and publish them to ChangeDB. For example, when an operator changes the configuration of an SNMP MIB for a device, we generate a record to the DB.

Derivation dependency. We populate the derivation dependency across change cubes A and B if we derive data A from data B. In the above example, the MIB change will result in multiple change cubes of job models. We build the dependency between the MIB config change and the rest of job model changes. Figure 8 shows derivation dependency in solid arrows across objects in different layers (each large dash box representing a layer). The dependency exists between data objects as well as between code and data.

Subscription to change cubes. Toposyncer subscribes to the change cubes and invokes corresponding processing logic accordingly, shown in line 19-23. For example, *sync_port* subscribes to the device data (e.g., `Thrift_Fboss_Linecards`). `Snmp_entPhysicalTable`), and a hardware blueprint (i.e., linecard map). If the hardware blueprint changes, i.e., the same linecard name is mapped to a different hardware blueprint, the change cube will be published and *sync_port* triggers its function *sync_phy_iface* function on the impacted interfaces. Similar pub/sub relation is also built between applications and derived data. For instance, as shown in Figure 8, a drainer application subscribes to interface status and the routing control messages to determine if it is safe to perform an interface drain operation.

5.2 Improve Trust on Data Quality

Real-world telemetry data may contain dirty or missing data. By exploring the change history, one can better judge whether the current values are trustworthy. Observing patterns of data changes can help predict the occurrences of future changes and identify missing changes.

Correlation dependency. Amongst normalized device data or derived data, data have relationships between them, shown as dash arrows within each large dash box in Figure 8. The relationship represents the physical dependency across objects, such as "contain", "connect", "originate". Previous topology-modeling works Robotron [41] and MALT [29] focus on the desired model and the correlation dependency in it. The desired model is built for the purpose of capturing topology intents and generating configurations. Here we use the model together with change cubes to verify if the actual topology's change is legit by comparing it against the desired models. A change cube generated from the desired object should have a matching change cube in the derived object, and vice versa. This can be done with a *Slice* on the entity, *Sort* by time, and compare the *Entity* of the changes.

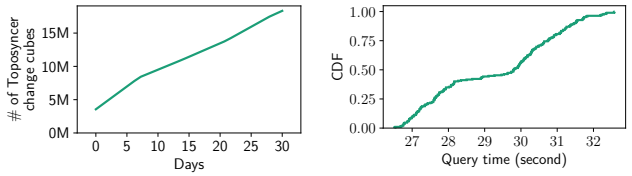
This correlation dependency can also be used for cross-layer validation of data quality. We implement *if-then* validation rules based on the correlation dependency on change cubes. We give two examples below. One use case is hard-stop fault detection. One rule is that *if* the logical interface fails (i.e., a specific change cube on a logical interface), *then* the routing session going through it will also fail (i.e., another change cube on the routing session must exist). If we observe significant errors at the lower layer but no upper failure, it indicates a measurement issue. In another use case, the aggregate interface consists of multiple physical interfaces. *If* a member physical interface reports packet errors, *then* the packet errors from aggregated interface should be larger than or equal to the physical interface errors. If the rule is not satisfied, it indicates some issues. These cross-layer dependencies can help us detect change-induced problems more quickly.

6 Evaluation

This section evaluates how the layer design of PCAT has helped with change tracking and how much benefit the change cube method has brought to use cases.

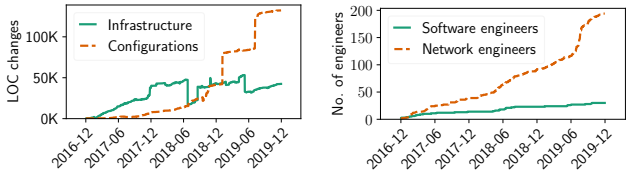
6.1 Change tracking implementation

First, we examine whether tracking all the changes is even feasible in a production environment. We show the change cube data volume grows with time in Figure 10a. Drawing from the experiences of Facebook's data infrastructure team, we employ a two-tier storage solution. We have an in-memory database to hold the change data for the most recent 30 days and have a disk-based SQL database for longer historical data. At the same time, the change data is published to our publish/subscribe system [4] for real-time propagation. Next, we



(a) # of change cubes over time. (b) Query distribution time.

Figure 10: Scalability and performance of change tracking.



(a) LoC changes over time. (b) Engineers over time.

Figure 11: Separating configurations with telemetry infra.

evaluate the performance of exploration using the primitives defined in §3.1, which is implemented using SQL statements. Figure 10b shows the query distribution time for data stored on disk, most of which centers around 27-32 seconds, due to the large data volume. For shorter duration of data in memory, it takes less than one second.

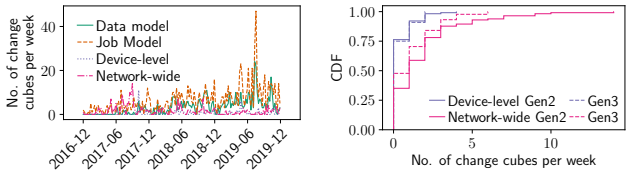
6.2 Benefits of separation

Analyzing change data over time helps us evaluate the long-term benefit of the layer design. We show it from three aspects.

Decoupled evolvement of configurations and infrastructure. We categorize changes broadly to configuration changes vs. infrastructure changes. We quantify the magnitude of the change using the Lines of Code (LOC) change. Figure 11a shows that the changes for configurations are 3.1 times more than core collection infrastructure changes. The sudden jumps for configurations in January 2019 are due to adding a large set of optical devices, which was not monitored by PCAT. The second increase around July 2019 is due to the migration to Gen3, resulting in a large number of new models added. The result shows that we increase the monitoring scope by configuration layer changes with a stable infrastructure.

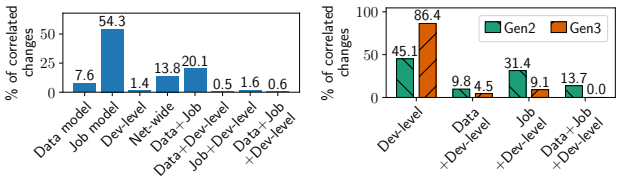
Scaling with divided responsibility. The separation in software systems has a long-term impact on the organization growth and people aspects. In Figure 11b, we analyze the change authors and categorize by their roles. It shows the number of network engineers who have made changes to configurations is increasing at a much faster pace than software engineers, with 7.2 times more people recently. The increase around June 2019 is due to both migration to Gen3 and adding more optical devices to monitor. It is clear that both of these changes are carried out by network engineers. It shows that PCAT enables network engineers to work on different network types while a small number of software engineers maintain infrastructure. It will boost a healthy collaboration environment where each team can play by their strength.

Confining the impact of changes. We use the number of change cubes as an approximate of the volume of changes.



(a) Change cubes across time. (b) CDF of change frequency.

Figure 12: Change cube frequency.



(a) Overall correlations. (b) Device-level correlations.

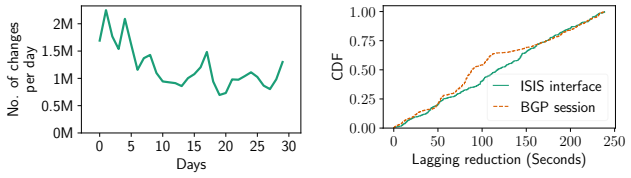
Figure 13: Correlated changes.

Figure12a shows its trends across time for data models, job models, device-level processing, and network-wide processing. The maximum numbers range from 15 to 50 for different categories. The models (data and job) have more changes due to the frequent intent changes. The infrastructure layers (device and network-wide processing) are more stable. Recently there are more data model and job model changes, because of Gen2-to-Gen3 migration. To directly illustrate the benefit of modular design in Gen3 (§4.2), Figure12b compares the frequency of change cubes for device-level and network-wide processing in Gen2 and Gen3 (after 2019-02). We observe that the average change frequency for network-wide processing in Gen3 is 38.1% lower than Gen2, while device-level remains similar. This means the modular design in Gen3 further prevents the changes in lower data model and job model layers from impacting upper processing layers, confining the impact of lower-layer changes. Note that we discounted the changes due to Gen2-to-Gen3 migration to have a fair comparison.

Reducing correlated changes. We find change cubes that occur close in time as correlated changes (e.g., data and job models are modified in the same commit). We show that PCAT's way of separating layers and models has helped reduce correlated changes. We first present the breakdown of different correlation combinations in both generations in Figure 13a. The largest combination is data and job, accounting for 20.1%. It is because adding new devices requires adding both data and job models. There are a small fraction of changes that require updating data, job, and device-level processing all together. Most of them are due to adding some specific counters that require special processing. Figure13b further breaks down all correlated changes related to device-level processing for Gen2 and Gen3. It shows that Gen3 has significantly reduced the correlated changes by 54.1%, 71.0%, and 100.0% (i.e., the second-to-last bar pairs) accordingly.

6.3 Benefits of change-driven Toposyncer

The first benefit is explicitly tracking changes in a centralized manner. Figure14a shows the magnitude of the changes over



(a) No. of changes to Toposyncer. (b) Lagging reduction.

Figure 14: Change-driven Toposyncer.

time to Toposyncer. Note that this is much higher than the changes presented earlier, since it includes the changes of raw data for network states.

The second benefit lies in the efficiency and accuracy improvement to applications. We evaluate it using the lagging time, i.e. the time between the change happening and when changes are reflected in derived topology by Toposyncer. Figure 14b shows the topology derivation is much more timely: reducing 118.76s lagging time for ISIS interface updates, and 108.93s for BGP session state updates, averagely.

7 Lessons and Future Directions

We discuss our lessons from building PCAT and the opportunities for future research.

Efficiency vs. adaptivity. We work closely with vendors to improve the efficiency of data collection primitives at switches (similar to academic work on reducing memory usage and collection overhead with high accuracy [24,26,46]). However, pursuing efficiency brings us challenges on adaptivity. Different devices have different programming capabilities and resource constraints to adopt efficient algorithms. Introducing new primitives also adds diversity and dynamics to upper layers in the telemetry system. For example, we work with vendors to support a sophisticated micro-burst detection on hardware. However, if only a subset of switches supports this new feature, applications need complex logic to handle detected and missed micro-bursts. Thus we have a higher bar for adopting efficient algorithms due to adaptivity concerns.

To support diverse data collection algorithms, we need a full-stack solution with universal collection interfaces at switches and change-aware data processing and aggregation algorithms. Recent efforts on standardizing switch interfaces such as OpenConfig [3] is a great first step but does not put enough emphasis on standardizing telemetry interfaces. Recent trends on open-box switches (e.g., FBOSS [13]) bring new opportunities to develop adaptive telemetry primitives.

Trustful network telemetry. Telemetry becomes the foundation for many network management applications. Thus we need to know which data at which time period is trustful. However, building a trustful telemetry system is challenging in an evolving environment with many changes of devices, network configurations, and monitoring intents. Fast evolution also introduces more misconfigurations and software bugs. Explicitly tracking change cubes and exploring their dependencies in PCAT is only the first step.

We need more principled approaches for *telemetry verification and validation* across monitoring intents, data models, and collection jobs. Compared with configuration verification work [7, 35], telemetry verification requires quantifying the impact of changes to the measurement results. One opportunity is that we can leverage cross-validations across multiple counters covering the same or related network states or across aggregated statistics over time. For example, we collect power utilizations (watts) from both switches and power distribution units (PDUs). In this way, we can validate the correctness of these utilization counters by comparing the PDU value with the sum of switch values.

Telemetry systems are complex time-series databases. We can leverage provenance techniques [12] to support change tracking, data integration, and troubleshooting. One challenge is that we cannot build a full provenance system due to vendor-proprietary code and network domain-specific data aggregation algorithms. There are also unexpected correlation dependencies across data.

Integration between telemetry and management applications. Our production networks are moving towards self-driving network management with a full measure-control loop. PCAT shows that changes bring a new complexity to the measure-control loop. Control decisions not only affect the network state that telemetry system captures but also the telemetry system itself. For example, an interface change may affect a counter scope. A traffic engineering control change may affect data aggregation because traffic traverses through different switches. These telemetry data changes in turn affect control decisions. We need to identify solutions that can feed control-induced changes directly into the telemetry systems.

Another question is how to present large-scale multi-layer telemetry data to control applications. Rather than providing a unified data stream, control applications can benefit from deciding what time, at what granularity, frequency, and availability level for data collection and the resulting overhead and accuracy in the telemetry system. One lesson we learned is to have the telemetry data available when it is mostly needed. For example, the network's aggregated egress traffic counter, which is collected at the edge PoPs, is a strong indicator of the business healthiness. To ensure its high availability, we need to give control applications the option to transfer the counter on more expensive out-of-band overlay networks. Moreover, we may extend the intent model to explicitly express the reliability-cost tradeoffs and adapt the tradeoffs during changes. We also need new algorithms and systems that can automatically integrate data at different granularities, frequencies, and device scopes to feed in control applications.

8 Related Work

Network evolvement. Several existing works have also pointed out the importance of considering changes. Both Robotron [41] and MALT [29] discuss it in the context of topology modeling, but miss the practical challenges of network monitoring. [16] discusses network availability during changes, while we focus on telemetry systems during changes.

Other monitoring techniques. PCAT is a passive approach. Active measurement injects packets into the network [14, 17, 18, 34, 49], and they are complements to passive measurement. The design principle of PCAT to handle changes can be applied to existing monitoring systems [20, 25, 36, 44, 48, 50], languages and compilers [9, 19, 32, 33]. PCAT also benefits from recent software-defined measurement frameworks [25, 27, 32, 46, 48]. For example, similar to OpenSketch [48], PCAT frees network engineers from configuring different measurement tasks manually. PCAT 's intent model design borrows ideas from the query language in Marple [32]. There are many memory-efficient monitoring algorithms [22, 23, 26, 30, 46] that focus on the expressiveness and performance of network monitoring. They provide adaptivity but only to a limited type of new queries, resource changes, or network condition changes. Here, PCAT focuses on a broader set of adaptivity (e.g., adaptive to counter semantics changes, data format changes, and more).

Dependency in network management. Dependency graph has been widely used for root cause localization [5, 6, 37, 42, 43, 47, 50]. Statesman [40] captures domain-specific dependencies among network states. We share some similarities but use dependency to tackle the change propagation.

Techniques from database and software engineering. Data provenance [12, 15] encodes causal relations between data and tables in metadata. Several works [11, 45] apply provenance to network diagnosis. [8] proposes the change cube concept and applies it to real-world datasets. All the above works focus on data face-value. On the other hand, software engineering community studies the problem of how a change in one source code propagates to impact other code [21, 51]. Ours looks at changes from telemetry systems from both data, configurations, and code.

9 Conclusion

This paper presents the practical challenge of a monitoring system to support an evolving network in Facebook. We propose explicitly tracking changes with change cubes and exploring changes with a set of primitives. We present extensive measurements to illustrate its prevalence and complexity in production, then share experiences in building a change-aware telemetry system. We hope to inspire more research on adaptive algorithms and evolvable systems in telemetry.

Acknowledgments

We thank our shepherd Chuanxiong Guo and the anonymous reviewers for their insightful comments. Yang Zhou and Minlan Yu are supported in part by NSF grant CNS-1834263.

References

[1] Express backbone. https://engineering.fb.com/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/.

[2] Introducing proxygen facebook c++ http framework. https://code.fb.com/production-engineering/introducing-proxygen-facebook-s-c-http-framework.

[3] OpenConfig YANG model. http://www.openconfig.net/projects/models/.

[4] Scribe. https://github.com/facebookarchive/scribe.

[5] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 419–435, 2018.

[6] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. *ACM SIGCOMM Computer Communication Review*, 37(4):13–24, 2007.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.

[8] Tobias Bleifuß, Leon Bornemann, Theodore Johnson, Dmitri V Kalashnikov, Felix Naumann, and Divesh Srivastava. Exploring change: a new dimension of data analytics. *Proceedings of the VLDB Endowment*, 12(2):85–98, 2018.

[9] Kevin Borders, Jonathan Springer, and Matthew Burnside. Chimera: A declarative language for streaming network traffic analysis. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 365–379, 2012.

[10] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. Simple network management protocol (snmp). Technical report, 1990.

[11] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 115–128. ACM, 2016.

[12] James Cheney, Laura Chiticariu, Wang-Chiew Tan, et al. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases*, 1(4):379–474, 2009.

[13] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 342–356. ACM, 2018.

[14] Cisco. Ip slas configuration guide, cisco ios release 12.4t. http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipsla/configuration/12-4t/sla-12-4t-book.pdf.

[15] Mahmoud Elkhodr, Belal Alsinglawi, and Mohammad Alshehri. Data provenance in the internet of things. In *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 727–731. IEEE, 2018.

[16] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 58–72. ACM, 2016.

[17] Nicolas Guilbaud and Ross Cartlidge. Google localizing packet loss in a large complex network. Nanog57, Feb 2013.

[18] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[19] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 357–371, 2018.

[20] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, volume 14, pages 71–85, 2014.

[21] Ahmed E Hassan and Richard C Holt. Predicting change propagation in software systems. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 284–293. IEEE, 2004.

[22] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 113–126, 2017.

[23] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 576–590, 2018.

[24] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 404–421, 2020.

[25] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 311–324, 2016.

[26] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 334–350. 2019.

[27] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.

[28] Chris Lonvick. The bsd syslog protocol. Technical report, 2001.

[29] Jeffrey C Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with modeling network topologies at multiple levels of abstraction. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 403–418, 2020.

[30] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Scream: Sketch resource allocation for software-defined measurement. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–13, 2015.

[31] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 129–143, 2016.

[32] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.

[33] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. Compiling path queries. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 207–222, 2016.

[34] Yanghua Peng, Ji Yang, Chuan Wu, Chuanxiong Guo, Chengchen Hu, and Zongpeng Li. detector: a topology-aware monitoring system for data center networks. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 55–68. USENIX Association, 2017.

[35] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 953–967, 2020.

[36] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 123–137. ACM, 2015.

[37] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Passive realtime datacenter fault detection and localization. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 595–612, 2017.

[38] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 418–431. ACM, 2017.

[39] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter

Xie. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, May 2015. USENIX Association.

[40] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A network-state management service. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 563–574, 2014.

[41] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 426–439. ACM, 2016.

[42] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 233–248, 2016.

[43] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: active device and link failure localization in data center networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 599–614, 2019.

[44] Mea Wang, Baochun Li, and Zongpeng Li. *sFlow: Towards resource-efficient and agile service federation in service overlay networks*. IEEE, 2004.

[45] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 395–420, 2019.

[46] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575. ACM, 2018.

[47] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. dshark: a general, easy to program and scalable framework for analyzing in-network packet traces. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 207–220, 2019.

[48] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI*, volume 13, pages 29–42, 2013.

[49] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252. ACM, 2012.

[50] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.

[51] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

APPENDIX

The first step of PCAT is to collect data from devices, which we call discovered data. There are three types of data including numeric counters, non-numeric states, and configurations. Table 4 shows the examples for each category.

Types	**Categories & examples**	**Impact of software upgrades**
Counters	*Device utilization:* CPU&memory utilization, routing table size, etc	Ambiguity between percentage and absolute values.
	Device internal status: Interface error counter, power supply temperature, fan speeds, linecard version, optical CRC error counter, etc	XML format gets changed; linecard version format changes from integer to string.
	Packet processing counters: Packet drops, errors, queue length, etc	Ambiguity of interface stats meaning.
	Protocol counters: BGP neighbor received routes, etc	General empty data error.
States	*Interface state:* Interface up, down, drained, configured IP address, MAC address, etc	Hex-decimal change causes MAC address retrieving error.
	Protocol state: BGP neighbor state, etc	State meaning ambiguity.
Configs	BGP policy, queuing algorithm, etc	Raw config format changed.

Table 4: Different discovered data in PCAT.