Concord: Learning Network Configuration Contracts

Ryan Beckett
Microsoft Research

Francis Y. Yan
UIUC

Raghunadha Reddy Pocha
Microsoft

Vineesh V. Raj
Microsoft

Ayyub Shaik
Microsoft

Siva Kesava Reddy Kakarla
Microsoft Research

Abstract

Misconfiguration is frequently cited as a leading cause of service disruptions and outages. To prevent misconfiguration, we introduce network *contracts*—lightweight configuration checks that run efficiently, localize errors to specific lines, and require no heavyweight modeling of network protocols. We develop a tool Concord to learn contracts automatically from example network configurations. By checking these learned contracts against new or changed configurations, Concord finds likely configuration bugs before they can impact the network. Key to our approach is a scalable algorithm for learning "relational" contracts that capture complex dependencies between configuration settings. We deployed Concord as part of a cloud-based configuration management service and evaluated its scalability, coverage, precision, and utility on two large real-world configuration datasets.

CCS Concepts: • **Networks** → **Network reliability**; **Network manageability**; • **Computing methodologies** → **Rule learning**; • **Information systems** → **Association rules**.

Keywords: Configuration validation, Misconfiguration detection, Network reliability, Association rule learning

ACM Reference Format:
Ryan Beckett, Francis Y. Yan, Raghunadha Reddy Pocha, Vineesh V. Raj, Ayyub Shaik, and Siva Kesava Reddy Kakarla. 2026. Concord: Learning Network Configuration Contracts. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3767295.3769338

1 Introduction

Configuration plays a crucial role in managing the behavior, performance, and security of networks [15, 17, 22, 31, 33, 52]. Configuration policies are employed extensively across the network stack, including in mobile base stations [5, 36] that transform radio signals into packets, routing protocols that guide packet transmission across the Internet [20, 52], and application-layer network orchestration frameworks [22, 31] that dynamically adapt resources to meet traffic demand.

However, configuring networks is fraught with challenges as configuration-related outages remain a major pain point for network operators, service providers, and enterprises alike [19, 47, 49, 51, 57, 60, 65]. Misconfigurations can impact millions of users and beget significant financial losses for organizations. Recently, major cloud providers [47, 57, 60] as well as airline, financial, e-commerce, and social media companies [49, 65] have all suffered from configuration-related network outages, and misconfigurations are a leading causes of outages in production networks [14, 26, 42, 48, 50, 61, 64].

One approach to solve the configuration problem is to validate their correctness *proactively* with formal verification. While formal verification provides strong guarantees of correctness and has found success in specific contexts such as the packet forwarding [6, 7, 30, 37–39, 44, 46, 63, 67] and for specific protocols [4, 8–10, 23, 25, 34, 55, 56, 59], many real networks utilize dozens of complex protocols with thousands of diverse configuration settings, the majority of which lack formal models and efficient verification tooling. Moreover, a complete set of network invariants is often unknown [35].

Conversely, although misconfiguration-related outages often manifest as violations of critical end-to-end invariants (such as loss of connectivity between an end user and a service), the root cause of many such outages is often extremely simple in hindsight [35, 36, 42]. For instance, an operator might erroneously set a timer value or disable an important switch interface [36], ultimately leading to a cascading network-wide failure. The remarkably simple nature of many such errors suggests that even lightweight validation could identify many misconfigurations.

Based on this observation, in this paper we propose a new approach to proactive configuration validation based on configuration *contracts*. Contracts are lightweight syntactic rules that easily checked against the configuration text of each device locally. For instance, a contract might state that if a line in the configuration sets the IP address of a loopback interface, then there must be a corresponding line that configures a static route for that IP address (§2). Contracts are typically *network-specific* as they capture key invariants implicit in configurations, which differ network-by-network.

EUROSYS '26, April 27–30, 2026, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769338

Contracts are effective at identifying many common misconfigurations, but forgo strong guarantees of correctness in exchange for several practical benefits. They are (1) *scalable* to large networks with thousands of devices due to being checked locally within each configuration file, (2) *actionable* in that they localize errors to specific configuration lines, (3) *protocol-agnostic* in that they do not require deep semantic modeling of specific network protocols, and (4) *learnable* automatically from operational configurations.

We present the design and implementation of Concord, a tool for learning contracts from and validating contracts against network configurations. In its learning mode, Concord analyzes the text of network configurations and infers a set of statistically likely contracts from their use. In its checking mode, Concord efficiently evaluates contracts (either learned or manually specified) against new or modified configurations to identify and localize potential bugs.

Some prior works learn "rules" for some simple configuration formats [41, 45, 53, 54, 66] modeled as a finite set of keys and values (*e.g.*, the key `max_connections` might have value `64` for MySQL [53]). For instance, Minerals [41] analyzes BGP configurations to extract some features into this format (*e.g.*, key `bgp_session_is_ebgp` with value `1`).

These simple models both (1) require extensive configuration preprocessing to extract features and (2) are not well suited to learn complex patterns typically observed in network configurations, such as those involving dependencies between repeated elements (*e.g.*, multiple interfaces), hierarchy (*e.g.*, a tag is part of a VLAN's configuration), and complex data structures (*e.g.*, a prefix filter lists).

To learn a richer set of contracts from network configurations, Concord takes a different approach. It requires no preprocessing and instead applies directly to configuration source text by modeling configurations more generally as a list of patterns (one per line) with data arguments. It then learns a simple yet expressive set of contracts over the resulting list of possibly repeated patterns of the form:

> *for every line1 matching pattern1, there exists a line2 matching pattern2 with values related by formula F.*

Relational contracts establish insightful connections between collections of configuration elements such as "each BGP session is configured over a valid interface." To learn such rules Concord adapts the framework of association rule learning [1, 28, 53] to identify statistically likely contracts.

Existing rule learning algorithms [1, 28] enumerate all possible candidate contracts. For Concord's new model, however, this approach is fundamentally unscalable, as (1) the number of candidate contracts scales super-linearly with the number configuration lines (millions for large networks), and (2) it can generate a commensurately large number of contracts. To address these issues, we introduce relation-finding data structures for common network data types that asymptotically reduce the number of candidate contracts to evaluate. We also implement a novel contract minimization algorithm using graph transitive reduction [2].

We deployed Concord as part of the Continuous Integration Continuous Delivery (CI/CD) pipeline of a cloud-based configuration management service to validate configuration changes. We then evaluated Concord against two large real-word datasets: mobile edge datacenters and a wide-area network totaling several million lines of configuration. For each dataset, we assessed Concord's scalability, its configuration coverage, its learning precision, and its practical utility.

Contributions. We make the following contributions:

- We motivate the use of lightweight *contracts* to proactively validate network configurations. We demonstrate that contracts are *scalable*, *actionable*, *protocol-agnostic*, and *learnable* from existing configurations.
- We describe the design, implementation, and production deployment of Concord, an efficient tool for both learning and enforcing network configuration contracts.
- We demonstrate that Concord executes quickly on configurations with millions of lines, exhibits a linear scaling trend, learns contracts that cover the majority of the configuration lines, achieves a precision of over 90% for most contract categories, and catches real misconfigurations.

Ethics. This work does not raise any ethical issues.

2 Overview of Concord

In this section, we present an overview of configuration contracts, discuss the challenges with learning configuration contracts and provide an overview of Concord.

Figure 1 shows example Arista configurations for mobile near edge datacenters. These datacenters consist of commercially available off-the-shelf (COTS) servers, network switches (configurations shown), and storage databases. The correctness of the configurations is critical, as they control routing between the user, mobile core network functions (NFs), cloud management infrastructure, and the Internet.

The correct operation of these datacenters relies on the switch configurations maintaining numerous contracts. For instance, the line `ip address 10.14.14.34` configures the IP address for the `Loopback0` interface, while another line `seq 10 permit 10.14.14.34/32` ensures the interface has connectivity.[1] In another example, each port channel (*e.g.*, `interface Port-Channel110`) uses BGP EVPN with a specific MAC address (*e.g.*, `00:00:0c:d3:00:6e`). While not required in general by the vendor (Arista), for the datacenter's design the port channel's name *must* correspond with the last segment of the MAC address in hexadecimal (*i.e.*, `110` in decimal is equal to `6e` in hex) for correct operation.

Unfortunately, a complete set of contracts is typically unavailable, undocumented, or unknown. Even when operators attempt to author contracts, doing so at scale for configurations with tens of thousands of lines and maintaining those

[1] All configuration values have been anonymized for security reasons.

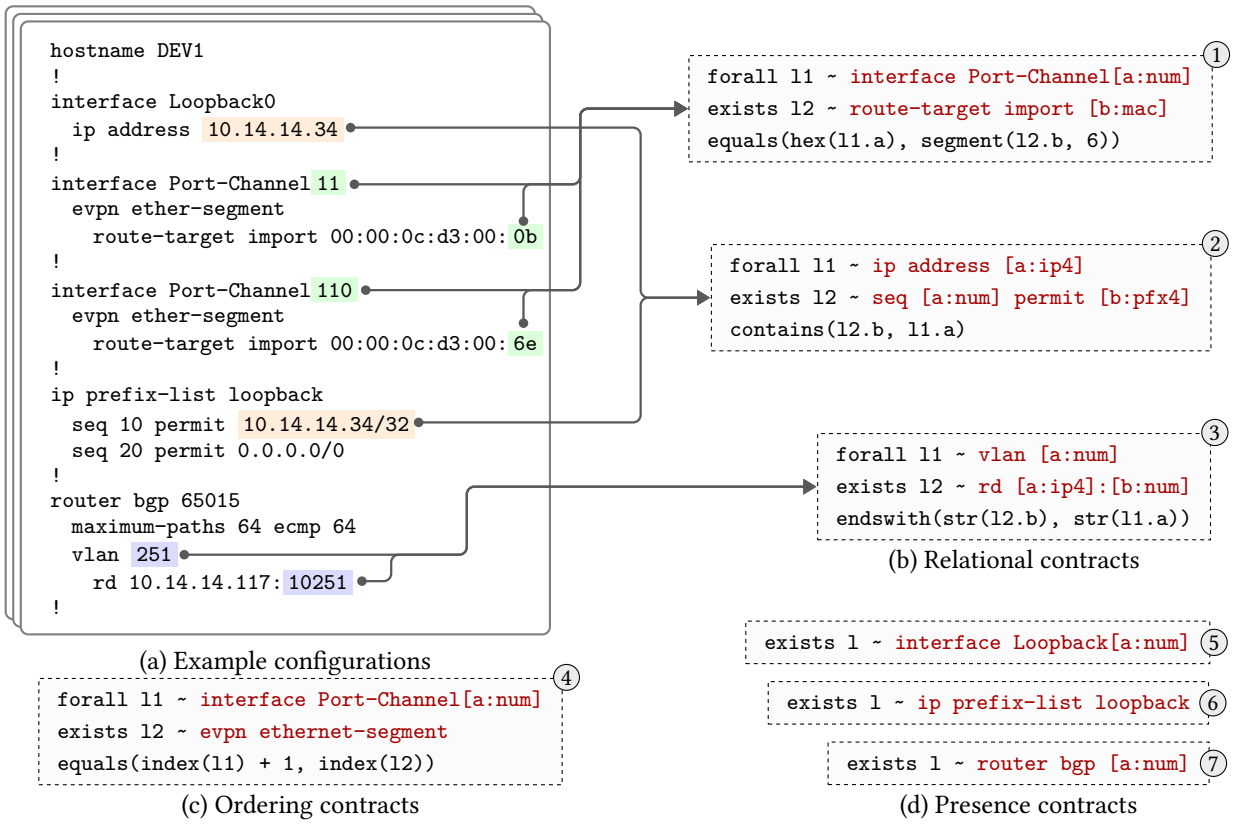

Figure 1. Example of Concord. Simplified configuration snippet (a) and contracts learned using Concord. Example contracts relate values across configurations (①, ②, ③) and define ordering (④) & presence (⑤, ⑥, ⑦) requirements. The syntax `l ~ ip address [a:ip4]` means that line `l` matches pattern `ip address [a:ip4]` where `a` is an IPv4 address. Relational contracts state that for every line `l1` matching some pattern, there exists a line `l2` in the same configuration matching another pattern such that their values are related in some way. For example, in ① the port channel number converted to hexadecimal (`hex(l1.a)`) is equal to the last segment of the MAC address from another line (`segment(l2.b, 6)`).

contracts as the network and configurations evolve is a herculean task. During our discussions with the maintainers of the edge data center configurations, they stated that they easily spent over 60% of their time trying to maintain such a set of contracts. They also pointed out that they at least know the current configurations are working as intended since they are deployed and running in production without issue, which motivates our problem:

Can we learn contracts directly from configurations?

Challenge 1: Complex configuration structure. While there is some prior work on learning "rules" for simple configuration formats, existing works do not learn the kinds of complex relationships found in most network configurations. Specifically, prior works model configurations as a set of unique keys and their values (e.g., the key `max_connections` might have value `64`) [53, 54, 66]. While they can handle some simple configuration formats (*e.g.*, MySQL database), they cannot analyze the complex structures and policies frequently found in network configurations such as lists (*e.g.*, `ip prefix-list loopback`) or repetitive elements (*e.g.*, `Port-Channel11` and `Port-Channel110`). For instance, they cannot represent even simple contracts such as "every loopback interface has an IP address configured."

As a concrete example, consider contract ② that Concord learned in Figure 1. The contract says that for each line `l1` in the configuration that matches pattern `ip address [a:ip4]`, there exists another line `l2` in the configuration that matches pattern `seq [a:num] permit [b:pfx4]`, where the IPv4 prefix in `l2` (`b`) contains the IPv4 address from `l1` (`a`). In essence, this captures the dependency that every interface IP address is permitted by some rule in the prefix list for security.

Challenge 2: Vendor-agnostic analysis. Recent works Diffy [36] and SelfStarter [35] perform some structural analysis of configurations to find bugs. However, they rely heavily on domain-specific parsing and algorithms. Even just parsing vendor-specific configurations into a suitable format is a challenging task for these tools since configurations consist of thousands of commands with specialized syntax, and are

constantly changing with the release of new firmware, OS versions, and configuration options.

When we tested Batfish [21], an open-source tool with the most comprehensive configuration parsers available, on the configurations in Figure 1, it only recognized 50% of the lines. Any downstream configuration analysis is thus *not even possible* for half of the configuration using these tools.

Concord instead treats configurations as unstructured text and does not require vendor-specific parsers. However, extracting structure from unstructured text is challenging. Different hardware vendors embed crucial information in ad hoc data formats and with implicit hierarchical structure. For instance, the route distinguisher `10.14.14.117:10251` in Figure 1 employs unconventional syntax and the loopback prefix list `ip prefix-list loopback` defines a configuration block that comprises multiple entries.

Challenge 3: False positives. Many relationships exist between values in different parts of the configuration, such as the loopback address and prefix list described earlier. Some of these relationships may be intentional, while others might occur purely by coincidence. For instance, the line `seq 20 permit 0.0.0.0/0` from the example contains a default prefix that coincidentally contains the IP address of the route distinguisher `10.14.14.117`. However, this relationship is neither meaningful nor predictive of bugs. Naively learning such contracts leads to false positives.

Challenge 4: Learning contracts at scale. Finally, today's network configuration files are *massive* with thousands or even tens of thousands of lines per device and hundreds to thousands of devices. Learning meaningful contracts while sifting through millions of lines of configuration presents a significant challenge. Rule mining methods are based on the concept of finding *frequent item sets*, which suffer from super-linear time complexity, making their direct application to large configurations infeasible. Moreover, these approaches are "noisy"—simply enumerating all valid rules.

How Concord works. Concord consists of two phases as shown in Figure 2. In the first phase it learns a set of contracts from *training* configurations as well as any other metadata files (*e.g.*, containing network or device state). It treats all inputs as unstructured text and attempts to extract likely contracts. In the second phase, it evaluates the contracts against a set of *test* configurations to find violations.

Concord's Assumptions. Concord assumes most training configurations are correct and that contracts persist over time. It tolerates some training configuration errors; if a majority of training examples share the same misconfiguration, Concord may learn an incorrect contract. However, validation against correctly configured instances typically surfaces these as anomalies for operator review. When best practices change or networks are redesigned, contracts can become invalid and require relearning. Some coincidental patterns are inevitable (we minimize them in §3); operators can readily spot invalid contracts, and anomalous ones are flagged pre-deployment for quick dismissal or update—far less work than inspecting full configurations.

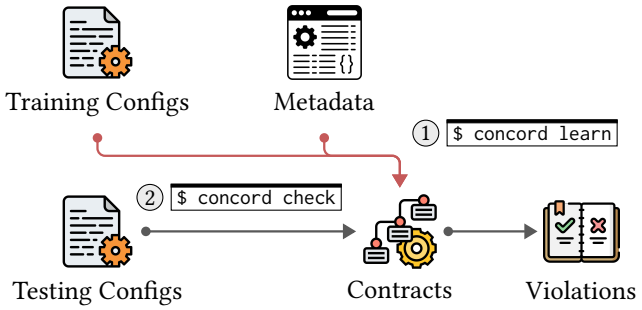

Figure 2. Concord workflow. `concord learn` ① infers a set of contracts from training configuration files and any system metadata. `concord check` ② applies the contracts to test configurations and report violations.

3 Contract Learning & Checking

To learn contracts effectively, Concord applies context embedding (§3.1) and data extraction (§3.2) phases to transform configuration text into a sequence of typed "patterns" and extracted data values. Concord then employs a variation of association rule learning [1] (§3.4, §3.5) to learn likely contracts from their use. It learns contracts asymptotically more efficiently through relation-aware search data structures (§3.5), reduces likely false positive contracts (§3.5), and minimizes the set of learned contracts (§3.6). Concord incorporates external data sources (§3.7) into its analysis, checks configurations efficiently (§3.8), and measures the coverage of its contracts when applied to configurations (§3.9).

3.1 Context embedding

Learning contracts without prior knowledge of vendor-specific syntax is challenging. Simply treating each line as an individual "unit" of text is ineffective for complex data formats. The line `ip address 10.14.14.34` from Figure 1 shares a connection with the `ip prefix-list loopback` line only because the IP address is set in the context of `interface Loopback0`. Similarly, many configuration formats like JSON and YAML organize data hierarchically, and this structure is not apparent at the line level.

To address this challenge, for each configuration, Concord first infers a data format *category*, such as JSON, YAML, Indentation, Unknown, etc. It then performs a simple context embedding pass to inject context into each line such as the JSON path or indentation hierarchy. For JSON, this involves including the "object keys" leading to the value and for indentation-based formats such as from Figure 1 it involves maintaining a stack of "parents" for each indentation level. We show this transformation in Figure 3.

This approach is effective because, despite the existence of thousands of distinct configuration dialects, the number of

pattern	parameters
/hostname DEV[a:num]	{a ↦ 1}
/!	
/interface Loopback[a:num]	{a ↦ 0}
/interface Loopback[num]/ip address [a:ip4]	{a ↦ 10.14.14.34}
/!	
/interface Port-Channel[a:num]	{a ↦ 11}
/interface Port-Channel[num]/evpn ether-segment	
/interface Port-Channel[num]/evpn ether-segment/route-target import [a:mac]	{a ↦ 00:00:0c:d3:0b}
/!	
/interface Port-Channel[a:num]	{a ↦ 110}
/interface Port-Channel[num]/evpn ether-segment	
/interface Port-Channel[num]/evpn ether-segment/route-target import [a:mac]	{a ↦ 00:00:0c:d3:6e}
/!	
/ip prefix-list loopback	
/ip prefix-list loopback/seq [a:num] permit [b:pfx4]	{a ↦ 10, b ↦ 10.14.14.34/32}
/ip prefix-list loopback/seq [a:num] permit [b:pfx4]	{a ↦ 20, b ↦ 0.0.0.0/0}
/!	
/router bgp [a:num]	{a ↦ 65015}
/router bgp [num]/maximum-paths [a:num] ecmp [b:num]	{a ↦ 64, b ↦ 64}
/router bgp [num]/vlan [a:num]	{a ↦ 251}
/router bgp [num]/vlan [num]/rd [a:ip4]:[b:num]	{a ↦ 10.14.14.117, b ↦ 10251}

Figure 3. Context embedding and data extraction from the example configuration from Figure 1. Configuration lines are pre-processed using an `indent`-based strategy. The extended path allows Concord to differentiate between similar lines that appear in different contexts. The original line is in red, and all extracted parameters from the lexer are shown on the right.

ways to structure hierarchical information in configurations is generally very small. We also note that while structural context can enhance Concord's learning, it still learns useful contracts in the absence of any embedding (§5).

3.2 Pattern and value extraction

Concord applies a lexing pass to separate each line into a pair of a typed pattern and a parameter map. In the example from Figure 1, Concord infers the format category as `indent`-based and captures the indentation hierarchy in the line, shown in Figure 3. We use the `/` separator to denote the parents based on the path. The choice of `/` for the separator is unimportant since Concord will treat the embedding line as uninterpreted text, and thus any text separator may be used depending on the domain. For instance, the line `/router bgp 65016/vlan 251` has pattern `/router bgp [num]/vlan [a:num]` with parameter map {a ↦ 251}. [2] These abstract "patterns" are useful because they identify configuration lines with only minor differences and enable learning contracts like "every loopback address is permitted by a prefix list."

The lexer defines a set of basic types using regular expressions (*e.g.*, numbers, IP addresses) and allows users to provide additional regular expressions to identify known domain-specific configuration objects like interface names. This step is optional and low-effort, but provides users with a way to refine how data values are captured by the tool. Table 1 shows examples. Internally, Concord stores data values in corresponding Rust programming language data types. For instance, it saves IP addresses using the `IPAddress` class, which is used later for efficient contract learning (see §3.5). The lexer is also extensible and supports custom regular expression patterns to allow users to define domain-specific patterns (*e.g.*, file paths).

We choose not to capture or bind variables for the embedded context from the parent configuration elements. This choice is due to the observation that Concord will learn many contracts between the captured parameters, few of which are meaningful. Any real relationship between the

Token	Regular Expression	Rust type
[iface]	([aA]e|[eE]t|...)-?[0-9]+	String
[descr]	description .+	String
[bool]	true|false	bool
[num]	[1-9][0-9]*	BigInt
[hex]	(0x|0)[0-9]+	BigInt
[mac]	[0-9a-zA-Z]+(:[0-9a-zA-Z]+){5}	MacAddress
[ip4]	[0-9]+(\.[0-9]+){3}	IPAddress
[pfx4]	[0-9]+(\.[0-9]+){3}/[0-9]+	IPNetwork

Table 1. Example lexer patterns that extract Rust data types. User-defined patterns are above the dotted line.

[2] Note: The lexer creates a typed pattern for the entire embedded line but extracts values only for the original text. This is a simple optimization since relationships may be found instead with original parent line.

parent and another line will be captured directly with the bound variables from that parent line.

3.3 Background on item set mining

After pre-processing the configurations to construct a sequence of patterns and data values, Concord efficiently learns statistically likely contracts using the framework of association rule learning [1]. Association rule learning is a data mining technique that discovers significant relations between data examples. It applies to *item sets* where one assumes the existence of a set of items $\mathcal{I}$ and a set of transactions $\mathcal{T}$, where each $T \in \mathcal{T}$ is a subset of the items with $T \subseteq \mathcal{I}$. The goal is to learn rules of the form $X \rightarrow Y$ meaning that transactions with item set X also contain item set Y. The significance of a rule is quantified by two metrics: *support* (S), the percentage of transactions in which both item sets co-occur, and *confidence* (C), the percentage of item sets in which the rule holds true.

In our setting, one can view each configuration as a transaction T and the typed patterns as comprising $\mathcal{I}$. However, there are several important differences. First, network configurations can contain ordered and repeated patterns such as multiple interface definitions. Second, each pattern carries with its data values such as IP prefixes or MAC addresses.

These differences necessitate learning a richer set of rules such as those from Figure 1. Additionally, prior algorithms for frequent item set mining such as Apriori [1] and FP-Growth [28] generate frequent item sets with high support first and then enumerate all candidate contracts between items within these sets to evaluate their confidence. This exhaustive search cannot scale to network configurations that are often thousands of lines long.

3.4 Configuration contract mining

Table 2 gives an overview of the contract categories Concord learns, examples of each contract, and the misconfigurations that the contract identifies. We selected the categories presented empirically, by observing their use in capturing common misconfiguration patterns. It is also easy to extend Concord to incorporate new categories. We now discuss how Concord learns each contract efficiently.

Relation contracts. Relational contracts such as those from Figure 1 (①, ②, and ③) identify dependencies between configuration elements. They quantify over all lines matching a particular pattern and have the logical form: if `forall l1 ~ p1` then `exists l2 ~ p2` such that `F`, where `F` is a formula over the values captured in `l1` and `l2`. Relational contracts detect misconfigured dependencies. We delve into their implementation in §3.5.

Present contracts. Contracts like ⑤, ⑥, and ⑦ of the form `exists l ~ p` state that the configuration must contain at least one line `l` that matches a given pattern `p`. These contracts, although simple, ensure that configurations do not miss any essential components. To learn them, Concord tracks every pattern used in each configuration and extracts those that appear in more than $C\%$ of the configurations.

Contract	Example	Misconfigurations
Present	Figure 1 (⑤, ⑥, ⑦)	Missing lines from a file
Ordering	Figure 1 (④)	Reordered, missing lines
Relation	Figure 1 (①, ②, ③)	Invalid dependencies
Type	`[ip6]` instead of `[ip4]`	Mistyped setting/value
Sequence	Filter lines are sequential	Missing/reordered lines
Unique	The router ID is unique	Copy paste, resource reuse

Table 2. The contracts Concord learns, usage examples, and common misconfigurations they identify.

Ordering contracts. Ordering contracts, such as ④, dictate that whenever a line `l1` matches pattern `p1`, the next/previous line `l2` must match pattern `p2`. Concord only considers ordering contracts for immediate successor/predecessor lines. This restriction facilitates quick learning and reduces noise since ordering contracts will "chain" together naturally to form blocks of lines that must appear together. For each configuration, we calculate the frequency of each successive pair of patterns (`p1`, `p2`). If `p1` and `p2` appear in at least $S\%$ of configurations and `p2` always follows `p1` in at least $C\%$ of the configurations, then Concord learns the contract.

Type contracts. Misconfigurations often occur due to simple type errors, such as using an IP prefix instead of an IP address [53]. We write type contracts as the non-existence of patterns, like `!(exists l ~ ip address [pfx4])`. Learning type contracts is subtle, as multiple types may be allowed for patterns, such as `ip address [ip4]` and `ip address [ip6]`.

Concord creates a *type-agnostic* representation for each pattern by replacing typed parameters with an untyped version. Both `ip address [ip4]` and `ip address [ip6]` become `ip address [type]`. Next, Concord calculates the frequency of each type used for every parameter in each untyped pattern. A type is considered invalid if it appears infrequently in fewer than $(100 - C)\%$ of uses. For example, Concord deems `ip address [bool]` a type error.

Sequence contracts. Sequence contracts are similar to ordering contracts and apply to number (`[num]`) parameters. They assert that the values across all instances of a given parameter are equidistant. For example, the pattern `seq [a:num] permit [b:pfx4]` has the sequence contract `sequence(a)`, as its values include 10, 20, 30, etc. A parameter is deemed sequential if it appears in at least $S\%$ of configurations and is sequential in at least $C\%$ of the configurations. Like ordering contracts, sequence contracts can identify missing sequential elements in configurations.

Unique contracts. The unique contract captures parameters with globally unique values across all configurations. For example, `hostname DEV[a:num]` possesses the contract `unique(a)`, as each configuration has a distinct name. Unique

contracts identify resources that should not be reused, such as a unique IP address, and can prevent copy-paste errors.

3.5 Learning relational contracts

Recall that relational contracts from Figure 1 have the form: if `forall l1 ~ p1` then `exists l2 ~ p2` such that `F`. A naive approach to finding these contracts is to calculate C and S for every pair of patterns `p1` and `p2`, every pair of captured values, and every formula `F`. This approach is infeasible, as real configurations have tens of thousands of parameters (see §5.1), leading to an explosion of candidate contracts.

A key observation is that the type of data relationship facilitates efficient search for contracts. For instance, consider the `contains` contract from Figure 1 (②) stating that a prefix value `[pfx4]` contains an IP address `[ip4]`. When searching for candidate `contains` relationships, we can find the possible matching prefixes for a given IP address in logarithmic time using a prefix trie data structure.

Thus, to efficiently identify possible relationships we employ fast relation search data structures. For contains contract, we construct a prefix trie from all configuration prefix values as illustrated in Figure 4. For equality contract, we build a hash table that maps each data value to the set of patterns and parameters it appears in. For affix (startswith and endswith) contract, we build a string trie.

We construct a lookup data structure for each relation type in a single pass over all configuration parameter values. In a second pass, we enumerate possible relationships, for instance reporting all matching (IP, prefix) pairs in the trie. If a relationship holds for a pair of patterns in a configuration and exceeds a score threshold (see below), then it is valid for that configuration. Concord learns the contract that is valid for at least S% and C% of configurations.

Data transformations. For learning, it is often useful to consider data transformations. For instance in ①, the contract `equals(hex(l1.a), segment(l2.b, 6))` related the port channel number to the last MAC segment by converting the former to hex and extracting the latter from the MAC address. To support such cases, Concord has a set of data transformations for each parameter type (*e.g.*, an IP octet) and enumerates all such transformations prior to search. The default transformation `id` is the identity function. We find that a small set of transformations (e.g., type conversions) is typically enough to cover many useful types of contracts.

Reducing false positives. Not every contract that holds across examples reflects operator intent. Contracts involving common patterns or values may arise coincidentally, whereas those involving rarer or more diverse values provide stronger evidence of intentionality. For example, numbers like `0–10` appear frequently in configurations and are more likely to yield spurious matches, while a port with value `3394` is far less likely to arise by chance and thus more likely to indicate a meaningful relationship.

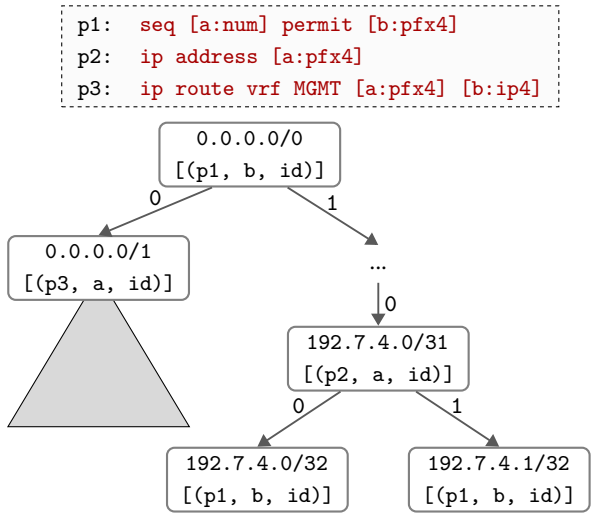

Figure 4. Example prefix trie for efficient lookup of `contains` relationships. The trie maps each `[pfx4]` value to its pattern, parameter, and transformation.

To address this, Concord employs a dynamic scoring and ranking mechanism that evaluates each potential contract along two dimensions:

- *Instance-level informativeness.* Each relation instance is scored by how unlikely it is to occur coincidentally. For example, the prefix `0.0.0.0/0` is assigned a score of 0 since it trivially contains all IPv4 addresses, while more specific prefixes (*e.g.*, `/24`) receive higher scores. For numerical values, a step function increases scores with distance from `0`, reflecting that values like `3852` are less likely to co-occur randomly than common values such as `1`. While heuristic and domain-agnostic (*e.g.*, `8888` may be common despite being large), this scoring aligns well with operator practices in real-world settings.
- *Diversity-based aggregation.* Scores are aggregated over unique values, rewarding rules that generalize across diverse instances rather than repeating the same coincidence. For example, a rule that holds for values {`5`, `6`, `9`, `11`} is more credible than one that always holds only for `5`.

Contracts whose cumulative scores exceed a configurable threshold are retained. This heuristic approach—penalizing uninformative matches and rewarding diversity—effectively filters spurious contracts while preserving useful ones. As illustrated in Figure 1, this is why Concord learns ② but rejects a spurious contract between `rd 10.14.14.117:251` and `seq 20 permit 0.0.0.0/0`.

3.6 Relational contract minimization

In some instances, the contract set Concord learns after filtering is still *huge*. Redundant contracts are undesirable because they increase validation time, make it difficult for humans to understand the contracts, and make it harder to

debug multiple violations. This is especially prominent in the case of transitive relations (where if pattern p1 implies p2 and p2 implies p3, then p1 must imply p3) such as equality, startswith, and endswith. This problem is fundamental: n patterns with mutual equality relations can have n^2 valid contracts between them, one for each pair of patterns.

Since the ultimate purpose of learning contracts is to find bugs, we observe that we can discard many contracts without affecting this bug-finding ability. Consider the patterns p4, p5, and p6 from Figure 5. Each has a named parameter (a) and Concord identifies equality relations between all of them. Rather than learn all six combinations of contracts, we use the transitivity of equality to eliminate redundant, implied contracts and learn a reduced contract set:

- `forall l4 ~ p4, exists l6 ~ p6, l4.a == l6.a`
- `forall l6 ~ p6, exists l5 ~ p5, l6.a == l5.a`
- `forall l5 ~ p5, exists l4 ~ p4, l5.a == l4.a`

If the configuration contains a bug, such as omitting a line for pattern p5, it will still trigger a violation (second contract).

We reduce the problem of contract minimization to that of graph transitive reduction [2]. The goal of transitive reduction is to replace a graph G, with a new graph G' over the same nodes. G' minimizes the number of edges while maintaining the reachability of G. In other words, for all nodes n_1, n_2, a path exists from n_1 to n_2 in G iff a path exists in G'. Intuitively, this reduction corresponds to contract minimization because we aim to minimize the number of edges (contracts) while preserving reachability (bug finding).

Concord creates a node for each (pattern, parameter, transformation) in the configuration, illustrated in Figure 5. It connects two nodes with an edge e if it learns a relational contract between those pairs. Concord computes the strongly connected components (SCCs) of the graph to identify nodes with complete connectivity (*e.g.*, p4, p5, and p6) and replaces all edges within this group with a simple cycle. Then, Concord collapses each SCC into a single node, resulting in a directed acyclic graph, and executes a transitive reduction algorithm to further eliminate edges. In Figure 5, this optimization reduces the original 20 contracts to 9, and frequently reduces a quadratic number of contracts (in the number of related patterns) to a much small linear number.

3.7 External data sources & metadata

External information can provide additional context when checking configurations. For example, a configuration might refer to a file path, that may not be valid in the system environment [66]. Automation generated the configurations in Figure 1 from metadata files that describe the expected policies (see §5.5). If the automation is buggy, the configurations may be wrong despite being internally consistent.

Concord allows users to supply arbitrary "metadata" files (*e.g.*, the output of the `ls` command) to enhance learning. We apply same context embedding and data extraction phases to these files and append their lines to each configuration.

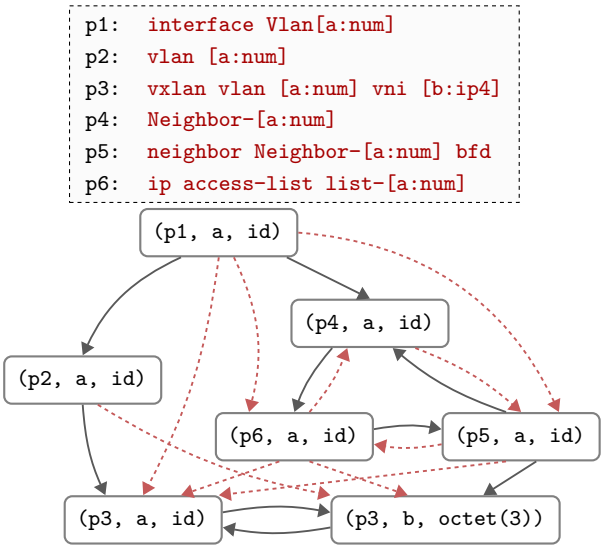

Figure 5. Contract minimization for patterns p1 through p6. Each (pattern, parameter, transformation) pair forms a node in the graph, and Concord adds an edge between two nodes with an for `equals` relation. Dashed red lines are removed.

The same learning algorithms discover any relationships between the configurations and the metadata.

3.8 Contract checking

Checking contracts learned from §3 against new or changed configurations lets Concord proactively identify bugs. Concord prepares the test configurations following the same steps outlined in § 3.1 and § 3.2, converting the test configurations into (pattern, parameter) pairs. Then it evaluates each contract. For example, to check ①, after encountering the line `interface Port-Channel11`, Concord finds lines matching the pattern `route-target import [b:mac]`. If none have parameter `b` that satisfies the contract then it reports an error with the configuration line numbers and values involved.

3.9 Measuring coverage

Since Concord's primary goal is to learn as many contracts and thereby catch as many bugs as possible, a natural question is how well the learned contracts "test" the configurations. If a contract never checks a given configuration line, we cannot expect to find bugs related to that line [12, 62]. Furthermore, knowing which configuration lines remain untested can guide the development of new categories of contracts and relationships to improve Concord.

While there are many known techniques for measuring the coverage of executable code (*e.g.*, branch coverage), configurations are generally not directly executed. Instead, to measure the completeness of the learned contracts, we define a simple and natural notion of *configuration coverage*:

A configuration line is covered if removing it would violate at least one contract.

For the contract ③ from the example in Figure 1, the line `rd 10.14.14.117:10251` is *covered* since removing it would result in an contract violation given the existence of line `vlan 251`. Concord summarizes the percent of configuration lines covered and also reports the coverage of each line.

4 Implementation

Concord is a command line tool implemented in 6557 lines of Rust. It offers two modes: `concord learn` and `concord check`. In `concord learn` mode, it accepts a file glob pattern defining training configuration files, an optional file specifying custom regular expression patterns for its lexer, and an optional glob pattern for metadata files. It then generates a file with a set of learned contracts in JSON format. In `concord check` mode, it takes the same inputs along with the contract file to check and reports all violations.

Concord accepts any configuration formats and includes special pre-processors for context embedding of common formats like JSON, YAML, and indentation-based formatting. During checking, it summarizes the coverage according to §3.9. Concord outputs a JSON file containing contract violations with specific lines and generates a user-friendly HTML output for viewing, filtering, and searching the violations. Operators can provide feedback through this user interface to suppress false positive contracts in the future.

The tool exposes three parameters that control learning:

- **Support (S):** the minimum number of configurations in which a pattern must appear. By default, $S = 5$, ensuring that contracts generalize beyond a handful of examples.
- **Confidence (C):** the required fraction of supporting instances in which the contract must hold. We set a high default of $C = 96\%$ to promote learning robust contracts while tolerating some noise or exceptions.
- **Heuristic scoring threshold:** used to filter spurious patterns, as described in §3.5.

Together, these parameters allow Concord to capture not only universal rules but also *non-universal contracts* that apply to subsets of devices. For example, a pattern that appears in 20 configurations and holds in 96% of those is retained, even if it does not hold globally across the dataset. Such flexibility is important in practice, since real networks often have operational drift or intentional role-specific variation.

These parameters are configurable, and tuning depends on the dataset. In a WAN with 50—100 routers per role drawn from a common template, using high S and C values helps filter a few outliers caused by misconfigurations. In more heterogeneous datasets, looser settings may be more appropriate. As with other association rule mining approaches, parameter selection balances precision and coverage.

Concord also includes command line flags to specify the parallelism level and enable a *constant-learning* mode to infer order and present contracts for exact line text. For instance, it can identify that a prefix list must include an exact set of lines. Finally, the implementation abstracts relation-learning data structures (*e.g.*, prefix trie) behind a simple interface, making it easy to implement new relationships.

Dataset	Lines	Patterns	Parameters	Learn	Check
E1	$O(10^3)$	981	761	0.1s	0.1s
E2	$O(10^4)$	169	72	0.1s	0.1s
W1	$O(10^5)$	744	559	1.5s	4.3s
W2	$O(10^5)$	4899	13936	4.0s	5.1s
W3	$O(10^5)$	2701	2684	0.9s	0.7s
W4	$O(10^6)$	9340	7944	16.0s	27.0s
W5	$O(10^6)$	4131	2545	12.0s	22.0s
W6	$O(10^6)$	9255	9432	33.3s	21.9s
W7	$O(10^5)$	3115	2930	1.4s	0.9s
W8	$O(10^4)$	707	232	0.1s	0.3s

Table 3. Dataset overview. The configuration lines, extracted patterns and parameters, `concord learn` runtime, and `concord check` runtime for each dataset. The exact role names and line counts are anonymized.

5 Evaluation

To evaluate Concord, we collected two large production network configuration datasets: mobile edge datacenter configurations and wide-area network configurations. We aimed to answer four questions regarding Concord's effectiveness:

- **RQ1:** Can Concord **scale** to large configurations?
- **RQ2:** Does Concord achieve high test **coverage**?
- **RQ3:** Are Concord's learned contracts **correct**?
- **RQ4:** Is Concord **useful** in practice?

5.1 Overview of datasets

We collected two extensive configuration datasets for different types of networks. These networks vary significantly in size (from thousands to millions of lines), characteristics, and structure (*e.g.*, features and syntax).

Mobile edge datacenters. The mobile near edge DC configurations are described in § 2. We specifically focus on switch configurations that manage packet routing between users, mobile core network functions (NFs) in the datacenter, cloud management infrastructure, and the public Internet. These DCs employ a leaf-spine architecture with varying SKUs (*e.g.*, 8 vs. 16 ToRs, 100G vs. 400G) based on deployment requirements. The configurations are generated according to the user policies and target SKU, where a SKU (stock keeping unit) defines a pre-packaged, standardized solution that includes hardware, software, topology, and workflows.

Wide-area network. We analyzed a large WAN, operated by a major cloud provider. This WAN has thousands of routers with millions of configuration lines. Devices are

assigned roles like edge routers and route reflectors. To optimize cost, performance, and avoid vendor lock-in, the WAN uses various hardware vendors for different roles.

Summary of networks. Table 3 presents an overview of each dataset's scale and characteristics. We divide the edge configurations into two categories: E1 for "leaf" devices and E2 for ToR devices. Similarly, we categorize the WAN into eight device roles. The table includes the number of lines, unique patterns, and parameters extracted by Concord from the configurations. We ran all experiments on a machine with 64GB RAM and a 3GHz Intel i9 CPU with 14 cores.

5.2 RQ1: Scalability of Concord

We evaluate the scalability of the tool by analyzing the runtime for both `concord learn` and `concord check` in Table 3. The table displays the time taken by the tool to learn contracts for each dataset. As we can see, Concord is *fast*, with `concord learn` taking under 34 seconds and `concord check` completing in under 22 seconds in all cases, even on the largest dataset W6 that consists of millions of lines of configuration. These times are inclusive of all aspects of Concord, including file parsing, context embedding, data extraction, contract mining, contract minimization, and contract checking. These times could also easily be reduced by using more cores as both contract learning and checking are parallelized.

Effectiveness of optimizations. To test the effectiveness of our fast relation finding data structures on contract learning and confirm our belief that naive contract learning is ineffective at scale, we disabled these optimizations and attempted to learn contracts by brute force (i.e., enumerating and checking all candidate contracts). Doing so unsurprisingly leads to non-termination of `concord learn`, which we timed out after 1 hour, for every WAN configuration dataset.

Scaling trend of Concord. To examine how Concord scales with the number of configurations, we used the large WAN datasets from Table 3 and created variable-sized subsets of the configurations. We then normalized the number of configurations and running time (combined `learn` and `check` time) and plotted the normalized runtime against the normalized network size (Figure 6). Concord scales nearly linearly with the configurations.

5.3 RQ2: Coverage of Concord

For each dataset, role, and contract category, we record the number of contracts learned with `concord learn` in Table 4. Using those contracts, we apply `concord check` to the same configurations to measure the test coverage (**Cov**) of the contracts for that dataset—that is, what percentage of the configuration lines would be tested by the learned contracts. For almost all datasets, only a few thousand contracts cover over 50% of the millions of configuration lines. Edge datacenter datasets have higher coverage (over 84%).

In addition to the total coverage for each dataset, we also show the coverage contributed individually for each contract category and dataset in Table 5. We can see that the coverage per contract category can vary greatly. For instance, the `unique` contract provides 18.2% in W5, and only 0.8% in W8. The present, ordered, and equality relational contracts provide the consistently highest coverage across all datasets, while the affix and type invariants provide the least coverage.

The type contract by design cannot increase configuration test coverage due to our choice of definition of coverage. In particular, removing a line from a configuration *cannot* lead to a violation of a type contract since it only identifies mis-typed lines that exist in the configuration.

The affix relation learns the fewest contracts and thereby has the lowest coverage. This is perhaps in part due to the particularities of the datasets evaluated. We expect such relational contracts to be more useful in configurations that use data objects such as file paths (*e.g.*, a file must be an extension of a directory configured elsewhere in the configuration).

While Concord only needs to learn a few thousand contracts to test the majority of lines in the configurations, manually authoring thousands of contracts and maintaining them over time would be infeasible for humans in practice.

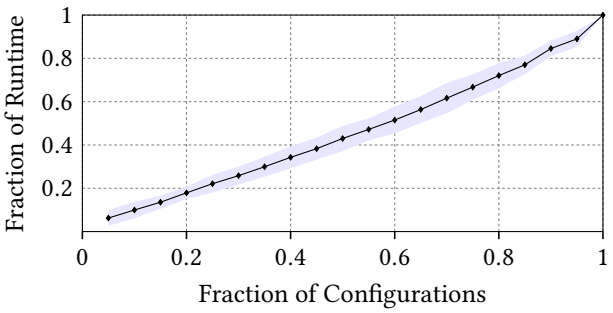

Figure 6. Concord exhibits a linear scaling trend. The shaded region represents the standard deviation.

Dataset	Present	Ord	Type	Unq	Seq	Relational			Cov
						E	C	A	
E1	979	1543	356	90	0	644	175	9	84.5%
E2	169	300	60	14	2	26	14	6	87.4%
Total	1010	1630	367	95	2	666	184	9	–
W1	380	974	300	38	9	280	19	2	66.3%
W2	540	1359	1642	51	0	3148	15	4	71.0%
W3	1845	3868	951	39	4	487	102	4	62.1%
W4	2741	12159	3598	269	29	2431	369	4	52.0%
W5	2964	6294	1065	80	23	715	256	4	59.6%
W6	3897	10070	2923	256	19	2442	380	4	49.9%
W7	1171	3148	1001	67	0	546	247	1	60.5%
W8	678	1234	165	23	0	117	94	0	68.6%
Total	7565	22313	7699	587	42	9217	869	10	–

Table 4. Contracts learned and coverage for each contract type and dataset. Total is all unique contracts.

Dataset	Present	Ord	Unq	Seq	Relational		
					E	C	A
E1	23.4%	64.0%	7.6%	0.0%	28.3%	6.4%	2.8%
E2	14.3%	67.9%	6.6%	7.0%	24.8%	1.4%	7.6%
W1	10.4%	39.5%	4.3%	9.4%	13.1%	0.7%	1.2%
W2	21.9%	64.5%	6.8%	0.0%	8.1%	0.2%	0.1%
W3	25.9%	46.4%	1.3%	0.4%	20.5%	0.7%	0.0%
W4	23.0%	42.2%	2.2%	0.5%	16.4%	0.8%	0.0%
W5	25.2%	43.5%	18.2%	1.0%	23.1%	8.2%	0.0%
W6	32.7%	42.2%	1.9%	0.1%	15.4%	0.8%	0.0%
W7	26.7%	40.8%	1.0%	0.0%	23.0%	1.1%	0.0%
W8	19.5%	65.8%	0.8%	0.0%	49.7%	0.1%	0.0%

Table 5. Concord coverage by specific contract category.

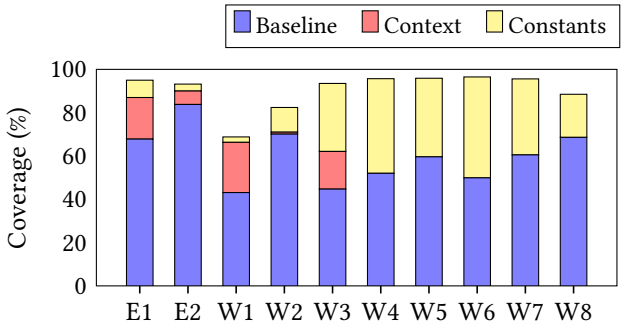

Figure 7. Effect of context embedding (§3.1) and constant learning (§4) on coverage for each dataset.

Coverage improvement. We measure the effect of both context embedding (§ 3.1) and constant learning (§ 4) on the contract coverage in Figure 7. For each dataset, these two optimizations lead to significant improvements in the overall coverage of the learned contract set. Several roles in the WAN (W4–W8) experienced no improvement from context embedding because they used a "flat" vendor syntax that already contains the complete context in each line. The lower baseline coverage in the WAN is due to many globally defined policies with "magic" constants that are disconnected from the rest of the configuration. The constant learning option helps identify and learn many of these policies.

Manually investigating the remaining untested configuration lines, we found that a majority are due to policies such as static routes and shared risk link groups, whose uses are both unique per device and simultaneously unrelated to the rest of the configurations.

Effectiveness of contract minimization. We evaluate the effectiveness of contract minimization (§ 3.6) by computing the contract reduction factor for relational rules per dataset. The reduction factor ranges from 22.3× for W6, to a minimum of 2.5× for W2, as shown in Figure 8.

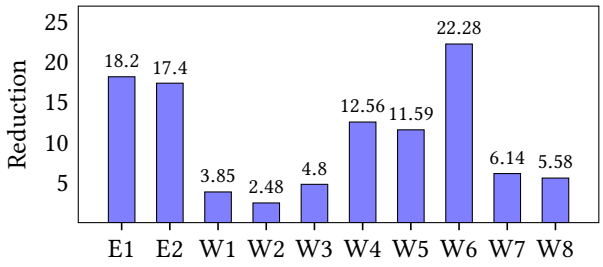

Figure 8. Effect of Concord's contract minimization algorithm for each network configuration dataset.

5.4 RQ3: Precision of Concord

Ideally, the learned contracts should reflect the actual constraints within configurations, meaning they should exhibit low rates of false positives (learning invalid contracts) and false negatives (omitting valid contracts). However, assessing these rates is challenging since we do not have knowledge of all the correct contracts. For this reason, similar to prior work [53, 54, 66], we focus on measuring the precision of Concord (of the contracts it learns, how many are valid). However, even measuring precision is challenging since for each dataset, Concord learned hundreds to thousands of contracts per category, making it impractical to manually assess the correctness of each one.

To estimate the precision, we needed to sample and manually review a large subset of the contracts. A natural question is: how many samples is enough for a statistically significant estimate of the true precision? Prior works on rule learning such as ConfigV [53] use experts to review a small, ad hoc number of learned rules (*e.g.*, 70). To do better, we attempt to obtain an estimate of the precision first, and then use this estimate to solve for the number of contracts we need to review manually for statistical significance.

To obtain an initial rough estimate of the precision, we turned to large language models (LLMs), specifically, GPT-4. For each dataset and contract category, we construct a prompt, including two demonstration examples. We instruct the LLM to act as an expert in analyzing network configurations, with the task of evaluating whether a given contract is likely valid for a specified role. We request a score from 1 to 10, with 10 indicating certainty that the contract is valid. We adopt the chain-of-thought [58] prompting technique to have the LLM provide a justification for its score. An example prompt for the equality contract is shown in Figure 11.

Estimating sample size through LLM scoring. Figure 9 displays the cumulative distribution function (CDF) of the scores given by the LLM for each contract category and dataset. We estimate a contract as a true positive if its score ranges from 6 to 10.

From these results, we compute the number of samples that must be manually reviewed to achieve a 95% confidence interval with a 5% error rate. In practice, this required a

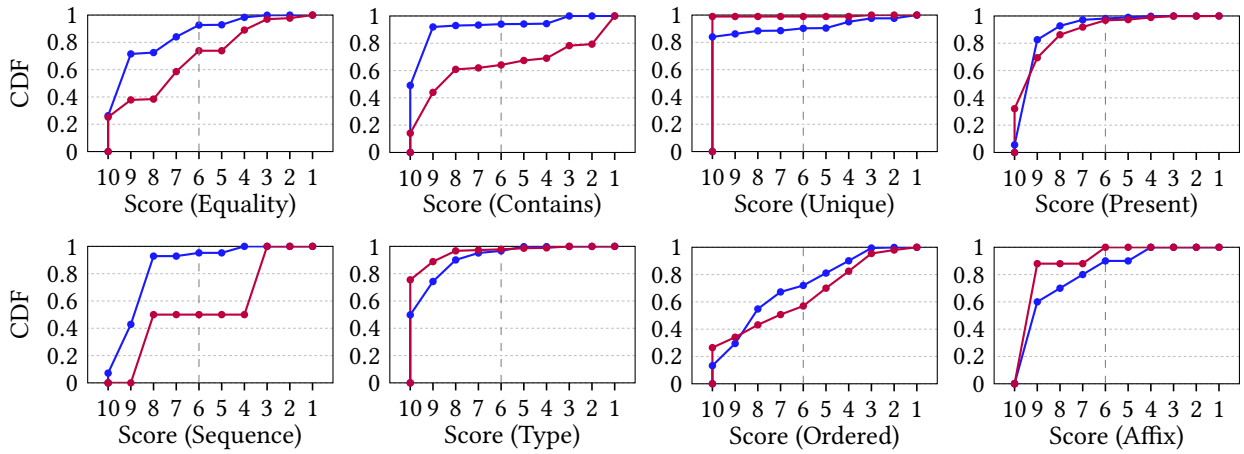

Figure 9. LLM scores for each contract type for WAN (blue) and edge networks (red). Scores (6–10 as true positives) provide an initial precision estimate to guide the sample size for statistically significant manual review.

substantial manual effort: we reviewed and labeled **1243** contracts as true or false positives.

For each contract type, we estimate the required *sample size for a proportion* using the following formula [16, 40]:

$$n = \frac{Z^2 \cdot p \cdot (1-p)}{E^2},$$

where Z is the z-score for the desired confidence level (1.96 for 95%), p is the estimated proportion of true positives (derived from the LLM scores), and E is the margin of error. The resulting n indicates how many samples must be examined to estimate the true positive rate with the specified accuracy.

Since we sample from a finite population of contracts, we apply the *finite population correction (FPC)*:

$$n_{adj} = \frac{n}{1 + \frac{n}{N}},$$

where N is the total number of contracts for a given type.

Table 6 reports the adjusted sample sizes (n_{adj}) and corresponding error rates (E) by category. While our target was 95% confidence with 5% error, in some cases like ordered contracts, the initial calculation suggested reviewing over 500 contracts. To balance feasibility, we capped the manual review at 150 per category. This cap slightly increased the error rate, but it never exceeded 10%. For categories with fewer than 10 contracts, we reviewed them all.

Manual estimate of precision. Table 7 shows the estimated precision from manual review, which is consistently very high with the exception of ordering contracts. The lower precision for ordered contracts is caused by the automation using a consistent, fixed format, like the sequence of an interface's description, IP address, and MTU, even though the order is technically interchangeable. In our production service, we simply disable ordering contracts (see §5.5).

Dataset		Present	Ord	Type	Unq	Seq	Relational		
							E	C	A
Edge	n_{adj}	102	150	36	13	2	150	122	9
	E	5%	10%	5%	5%	0%	7%	5%	0%
WAN	n_{adj}	42	150	70	122	29	150	86	10
	E	5%	10%	5%	5%	5%	6%	5%	0%

Table 6. Number of samples manually examined (n_{adj}) and the corresponding error rate (E) by specific contract category for 95% confidence of true positive rate.

Example contracts. We showcase several of the more simple and intuitive example contracts that Concord learned in Table 8. For instance, the learned contract

```
forall l1 ~ ...RFC1918/seq [a:num] permit [b:pfx4]
exists l2 ~ ...PRIVATE/seq [a:num] permit [b:pfx4]
equals(l1.b, l2.b)
```

states that the defined internal address space for each device includes private (RFC 1918) address space. While these examples are meant to that many contracts are simple intuitive, Concord also learned thousand of other contracts involving specific low-level policies (e.g., a relationship between legacy IGP configuration and NTP servers), which would have been challenging to write or extract by hand.

5.5 RQ4: Utility of Concord

We deployed Concord as part of the Continuous Integration, Continuous Deployment (CI/CD) pipeline for a cloud-based network device configuration service used to manage the mobile edge datacenter deployments described in §2. The service takes user-defined network policies (metadata) and generates low-level device configurations for these policies.

Dataset	Present	Ord	Type	Unique	Seq	Relational		
						E	C	A
Edge	100	38	100	100	100	86	100	100
WAN	100	71	94	90	100	92	98	80

Table 7. Concord precision (in %). Ordered contracts have lower precision as the tool learns fixed line order from generated configurations, though they are interchangeable. We disable Ordered contracts by default.

Dataset	Contract Description
Edge	The next hop addresses of management static routes are for well-defined management interfaces.
WAN	Inbound and outbound perimeter ACLs have symmetric destination and source address filters.
WAN	Prefix-lists defining internal address space subsume those that define bogon (RFC 1918) address space.
WAN	If certain BGP group policies are configured for IPv4, then they are also configured for IPv6.
WAN	Every interface address should be unique across all the interfaces and devices in a role.

Table 8. English descriptions of a selected subset of simple and intuitive contracts that Concord learned. Concord also learned thousand of other contracts involving specific policies, whose details we do not reveal.

The service must support numerous SKUs and vendor platforms and contains complex logic to optimize switch resources through the configuration based on the SKU and platform. For instance, a low-end 100G SKU uses different configuration than a high-end 400G SKU. Moreover, the correctness of this service is critical, as any configuration errors can blackhole traffic and disconnect client network.

Prior to Concord, to ensure the correctness of the generated configurations, service maintainers manually curated and authored custom contracts and automation to validate different aspects of the configurations. However, maintaining these contracts over time and understanding what the "gaps" existed in these contracts presented a major challenge.

To address these challenges, we worked with the maintainers to deploy Concord as part of the CI/CD pipeline for the service as depicted in Figure 10. When a developer initiates a pull request for the service, the CI/CD pipeline will run the service infrastructure code both pre-change and post-change to generate configuration files from various test user policies. Concord uses the pre-change configurations to learn contracts, which it then checks against the post-change configurations. Contract violations block the pull

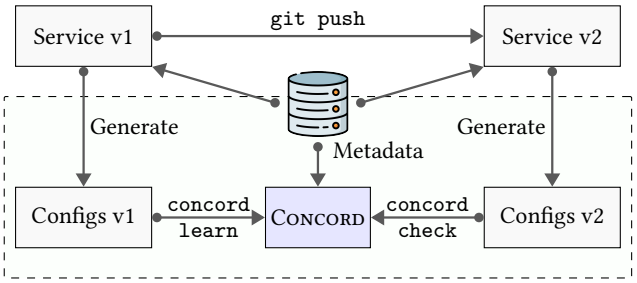

Figure 10. CI/CD workflow for the mobile edge datacenter configuration management service.

request pending manual review as they may indicate software regressions. Since Concord's deployment, the service has experienced no major outages due to configuration. However, we replayed several past incidents to see if Concord would prevent them, of which we discuss just three now.

Example 1: Missing route aggregation. A change to a data definition in the code base (*i.e.*, an array of strings to an array of structs) introduced a bug where a struct was not checked for the null value. In this case, the service omitted the configuration summarizing specific BGP routes with an aggregate at the spine switches. The spine switches were configured to filter non-aggregate routes, resulting in traffic being dropped for the entire fabric. When replayed, Concord flagged the issue early through a violation of the contract:

```
forall l1 ~ /ip route vrf Mgmt [a:pfx4] [b:ip4]
exists l2 ~ /vrf Mgmt/aggregate-address [a:pfx4]
contains(l2.a, l1.b)
```

which reported the absent aggregate line due to the presence of a static route with a particular next hop IP address.

Example 2: MAC broadcast loop. As part of an effort to reduce cost to customers, the service introduced a new SKU with fewer switches. This new SKU required specific configuration changes to connect storage directly to spine devices, and those changes were mistakenly extended to an earlier SKU. The changes added erroneous layer 2 configuration that created a broadcast loop during MAC learning. This caused the switch CPU to monopolize resources and eventually led to the fabric blackholing all traffic, including management traffic the operators used to troubleshoot the problem. For the existing SKU, Concord learned the contract:

```
forall l1 ~ /router bgp/vlan [a:num]
exists l2 ~ @meta/nfInfos/vrfName/vlanId/[a:num]
equals(l1.a, l2.a)
```

between the spine configuration and the network metadata file. It correctly detected that several added VLAN configuration blocks should not exist, catching the issue.

Example 3: Multiple VRFs. In a final case, a software bug pushed incorrect configuration to a virtual routing and forwarding (VRF) element. This caused multiple VRFs to

learn the same routes over BGP EVPN and created a conflict in the router forwarding information base (FIB). As a result, both northbound traffic and southbound could be forwarded incorrectly. Concord could identify this misconfiguration through multiple ordering contracts such as:

```
forall l1 ~ redistribute connected
exists l2 ~ neighbor [a:ip4] peer-group OPT-A
equals(index(l1) + 1, index(l2))
```

The erroneous configuration was inserted between these lines and thus broke this ordering contract.

6 Related Work

Concord is related to several threads of research:

Network verification. Verification can find and even prove the absence of bugs in network configurations. Verifying a network requires both (1) formal mathematical modeling of the specific protocols and systems being configured and (2) a complete set of contracts defining the intended behavior of the network. Neither is easily obtained. Verification has found success in some narrow networking contexts such as data plane (packet forwarding) [6, 7, 30, 37–39, 44, 46, 63, 67] and for specific protocols such as BGP [4, 8–10, 23, 25, 55, 56, 59] and DNS [34]. However, extending it protocol-by-protocol is an arduous process [11].

Because Concord does not model the semantics of network protocols and because its contracts are primarily text-based, it provides no guarantees of correctness. This stands in contrast to the work on network verification. Instead, one should view Concord as a best-effort configuration validator. In exchange, contracts have other advantages including being scalable, actionable, protocol-agnostic, and learnable.

Configuration linters. Configuration "linters" also take a heuristic approach to finding bugs. RCC [20], for example, applied handcrafted heuristics to find issues in BGP router configurations. SelfStarter [35] identified bugs in BGP route filters, prefix lists, and ACLs by finding template outliers, and Diffy [36] generalized this approach to hierarchical JSON files. Another work [3], extracts configuration components (*e.g.*, interfaces, ACLs, VLANs) and relationships from JSON-based configurations by inferring types and names using attribute overlap, identifying cycles in the resulting graph as recurring motifs to detect misconfigurations. However, it is limited to structured JSON data. All these works either manually target specific policies [20, 35] or require specific formats [3, 36]. Concord is the first to learn complex structured contracts from plain text.

Configuration contract mining. Prior works like ConfigC [54], ConfigV [53], and Encore [66] learn rules similar to our ordering and relational contracts, but assume a model where configurations are pre-parsed as pairs of unique keys and values (e.g., the key `max_connections` has `64`). Minerals [41] takes a similar approach, but for the BGP protocol. Aurora [45] applies a majority-rule approach to optimize LTE/5G configurations which are also key value pairs. In contrast, Concord treats configurations as unmodified text and uses a more expressive model of configurations as a list of patterns that carry data arguments. It then learns more expressive relational contracts of the form "forall line1 matching pattern1 there exists a line2 matching pattern2 with related data values". Another line of work Config2Spec [13] tries to learn end-to-end reachability invariants specifically for packet forwarding in the context of routing protocols. Our work focuses primarily on learning syntactic contracts that have the aforementioned benefits of being scalable, actionable, protocol-agnostic, and easier to learn.

Association rule learning. Concord broadly falls into a data mining paradigm known as association rule learning, which identifies frequent relations between items in datasets. Classic algorithms such as Apriori [1] and FP-Growth [28] employ a two-step approach—first generating frequent item sets and then exhaustively searching for valid relations within these sets. However, this approach is well known to be computationally expensive, rendering it impractical for configurations with thousands of lines. Concord extends conventional association rule learning with multiple techniques to efficiently filter candidate contracts and scale.

7 Conclusion

We presented Concord, a tool that learns contracts from example network configurations. By checking these contracts against configuration changes, Concord detects bugs before they can impact the network. Our approach does not require protocol-specific modeling, making it applicable to a wide range of network configurations. We demonstrated the effectiveness of Concord through evaluation on two large real-world configuration datasets. We showcased its ability to scale when analyzing and learning from large configurations with millions of lines, to learn contracts with high configuration coverage, to achieve a high true positive rate, and to detect bugs in practice.

References

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. 207–216.

[2] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. 1972. The transitive reduction of a directed graph. *SIAM J. Comput.* 1, 2 (1972), 131–137.

[3] Sara Alam, Devon Lee, and Aaron Gember-Jacobson. 2022. Poster: Identifying Syntactic Motifs and Errors in Router Configurations Using Graphs. In *2022 IEEE 30th International Conference on Network Protocols (ICNP)*. IEEE, 1–2.

[4] Timothy Alberdingk Thijm, Ryan Beckett, Aarti Gupta, and David Walker. 2023. Modular control plane verification via temporal invariants. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 50–75.

[5] Edoardo Amaldi, Antonio Capone, Federico Malucelli, and Francesco Signori. 2002. UMTS radio planning: Optimizing base station configuration. In *Proceedings IEEE 56th Vehicular Technology Conference*, Vol. 2. IEEE, 768–772.

[6] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotić, Carsten Varming, and Blake Whaley. 2019. Reachability Analysis for AWS-Based Networks. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 231–241.

[7] Ryan Beckett and Aarti Gupta. 2022. Katra: Realtime Verification for Multilayer Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 617–634. https://www.usenix.org/conference/nsdi22/presentation/beckett

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 155–168. https://doi.org/10.1145/3098822.3098834

[9] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control Plane Compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 476–489. https://doi.org/10.1145/3230543.3230583

[10] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2019. Abstract Interpretation of Distributed Network Control Planes. *Proc. ACM Program. Lang.* 4, POPL, Article 42 (dec 2019), 27 pages. https://doi.org/10.1145/3371110

[11] Ryan Beckett and Ratul Mahajan. 2020. A General Framework for Compositional Network Modeling. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20)*. Association for Computing Machinery, New York, NY, USA, 8–15. https://doi.org/10.1145/3422604.3425930

[12] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.

[13] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. 2020. {Config2Spec}: Mining network specifications from network configurations. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 969–984.

[14] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and discovering software configuration dependencies in cloud and datacenter systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 362–374. https://doi.org/10.1145/3368089.3409727

[15] Cisco. 2023. Basic Router Configuration. https://www.cisco.com/c/en/us/td/docs/routers/access/800M/software/800MSCG/routconf.html. (2023). [Online; accessed 30-March-2023].

[16] William G Cochran. 1977. Sampling techniques. *John Wiley& Sons* (1977).

[17] Oracle Corporation. 2023. MySQL. https://www.mysql.com/. (2023). Accessed: 2023-11-01.

[18] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Baobao Chang, Xu Sun, Lei Li, and Zhifang Sui. 2024. A Survey on In-context Learning. (2024). arXiv:cs.CL/2301.00234 https://arxiv.org/abs/2301.00234

[19] Evolven. 2022. Downtime, Outages and Failures - Understanding Their True Costs. https://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html. (2022). Accessed: March 26, 2023.

[20] Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP configuration faults with static analysis. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. 43–56.

[21] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 469–483. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel

[22] Cloud Native Computing Foundation. 2023. Kubernetes Documentation. https://kubernetes.io/docs/home/. (2023). Accessed: 2023-11-01.

[23] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 300–313. https://doi.org/10.1145/2934872.2934876

[24] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 39, 13 pages. https://doi.org/10.1145/3597503.3608134

[25] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. 2020. NV: An Intermediate Language for Verification of Network Control Planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 958–973. https://doi.org/10.1145/3385412.3386019

[26] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-Anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. 2014. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM symposium on cloud computing*. 1–14.

[27] Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2024. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 34, 13 pages. https://doi.org/10.1145/3597503.3623306

[28] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. *ACM sigmod record* 29, 2 (2000), 1–12.

[29] Hangfeng He, Hongming Zhang, and Dan Roth. 2022. Rethinking with Retrieval: Faithful Large Language Model Inference. (2022). arXiv:cs.CL/2301.00303 https://arxiv.org/abs/2301.00303

[30] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time Network Verification Using Atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 735–749. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/horn-alex

[31] Istio 2023. Istio Configuration. (2023). Retrieved Sep 19, 2023 from https://istio.io/latest/docs/ops/configuration/

[32] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R. Lyu. 2024. LILAC: Log Parsing using LLMs with Adaptive Parsing Cache. *Proc. ACM Softw. Eng.* 1, FSE, Article 7 (jul 2024), 24 pages. https://doi.org/10.1145/3643733

[33] Juniper Networks. 2023. CLI User Guide for Junos OS. https://www.juniper.net/documentation/us/en/software/junos/cli/index.html. (2023). Accessed: April 2, 2023.

[34] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. 2020. GRoot: Proactive Verification of DNS Configurations. 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 (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 310–328. https://doi.org/10.1145/3387514.3405871

[35] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. 2022. SCALE: Automatically Finding RFC Compliance Bugs in DNS Nameservers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 307–323. https://www.usenix.org/conference/nsdi22/presentation/kakarla

[36] Siva Kesava Reddy Kakarla, Francis Y. Yan, and Ryan Beckett. 2024. Diffy: Data-Driven Bug Finding for Configurations. *Proc. ACM Program. Lang.* 8, PLDI, Article 155 (Jun 2024), 24 pages. https://doi.org/10.1145/3656385

[37] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 99–111. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/kazemian

[38] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 113–126. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian

[39] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 15–27. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid

[40] Leslie Kish. 1965,. *Survey Sampling*. John Wiley & Sons,, Newyork,.

[41] Franck Le, Sihyung Lee, Tina Wong, Hyong S. Kim, and Darrell Newcomb. 2006. Minerals: Using Data Mining to Detect Router Misconfigurations. In *Proceedings of the 2006 SIGCOMM Workshop on Mining Network Data (MineNet '06)*. Association for Computing Machinery, New York, NY, USA, 293–298. https://doi.org/10.1145/1162678.1162681

[42] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 599–613.

[43] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9, Article 195 (jan 2023), 35 pages. https://doi.org/10.1145/3560815

[44] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, USA, 499–512.

[45] Ajay Mahimkar, Zihui Ge, Xuan Liu, Yusef Shaqalle, Yu Xiang, Jennifer Yates, Shomik Pathak, and Rick Reichel. 2022. Aurora: conformity-based configuration recommendation to improve LTE/5G service. In *Proceedings of the 22nd ACM Internet Measurement Conference (IMC '22)*. Association for Computing Machinery, New York, NY, USA, 83–97. https://doi.org/10.1145/3517745.3561455

[46] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. *SIGCOMM Comput. Commun. Rev.* 41, 4 (aug 2011), 290–301. https://doi.org/10.1145/2043164.2018470

[47] Nextgov. 2021. Commercial Cloud Outages Are a Wake-Up Call. https://www.nextgov.com/ideas/2021/03/commercial-cloud-outages-are-wake-call/172731/. (2021). Accessed: 2023-11-01.

[48] David Oppenheimer, Archana Ganapathi, and David A Patterson. 2003. Why do Internet services fail, and what can be done about it?. In *4th Usenix Symposium on Internet Technologies and Systems (USITS 03)*.

[49] Raymond Pompon. 2021. BGP, DNS, and the fragility of our critical systems. https://www.f5.com/labs/articles/cisotociso/bgp-dns-and-the-fragility-of-our-critical-systems. (2021). Accessed: 2023-11-01.

[50] Ariel Rabkin and Randy Howard Katz. 2012. How hadoop clusters break. *IEEE software* 30, 4 (2012), 88–94.

[51] Teri Radichel. 2023. About the 5-hour Microsoft Outage. https://medium.com/cloud-security/about-the-5-hour-microsoft-outage-18d47543769d. (2023). Accessed: 2023-11-01.

[52] Yakov Rekhter, Susan Hares, and Tony Li. 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271. (Jan. 2006). https://doi.org/10.17487/RFC4271

[53] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. 2017. Synthesizing configuration file specifications with association rule learning. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–20.

[54] Mark Santolucito, Ennan Zhai, and Ruzica Piskac. 2016. Probabilistic automated language learning for configuration files. In *Computer Aided Verification: 28th International Conference, CAV 2016, Proceedings, Part II 28*. Springer, Springer, Cham, Toronto, ON, Canada, 80–87.

[55] Alan Tang, Ryan Beckett, Steven Benaloh, Karthick Jayaraman, Tejas Patil, Todd Millstein, and George Varghese. 2023. Lightyear: Using modularity to scale bgp control plane verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 94–107.

[56] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. 2021. Campion: Debugging Router Configuration Differences. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 748–761. https://doi.org/10.1145/3452296.3472925

[57] Liam Tung. 2019. Azure global outage: Our DNS update mangled domain records, says Microsoft. https://www.zdnet.com/article/azure-global-outage-our-dns-update-mangled-domain-records-says-microsoft/. (2019). Accessed: 2023-11-01.

[58] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

[59] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of

Border Gateway Protocol Configurations with an SMT Solver. *SIGPLAN Not.* 51, 10 (oct 2016), 765–780. https://doi.org/10.1145/3022671.2984012

[60] Kurt Wise. 2017. High Number of AWS Misconfigurations Leaves Huge Security Holes. https://virtualizationreview.com/articles/2017/04/19/aws-misconfigurations-leaves-huge-security-holes.aspx. (19 Apr 2017).

[61] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 244–259.

[62] Xieyang Xu, Weixin Deng, Ryan Beckett, Ratul Mahajan, and David Walker. 2023. Test Coverage for Network Configurations. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1717–1732.

[63] Hongkun Yang and Simon S. Lam. 2016. Real-time Verification of Network Properties Using Atomic Predicates. *IEEE/ACM Trans. Netw.* 24, 2 (April 2016), 887–900. https://doi.org/10.1109/TNET.2015.2398197

[64] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. 2011. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 159–172.

[65] Iris Zarecki. 2019. 19 of the worst IT outages in 2019 – A Recap of Being Let Down. https://www.continuitysoftware.com/blog/19-of-the-worst-it-outages-in-2019-a-recap-of-being-let-down/. (2019).

[66] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 687–700. https://doi.org/10.1145/2541940.2541983

[67] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. APKeep: Realtime Verification for Real Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 241–255. https://www.usenix.org/conference/nsdi20/presentation/zhang-peng

LLM prompt

Task: As an expert in network configurations, your primary responsibility is to evaluate the validity of relationships between lines in network configuration files. Each relationship is defined by two patterns. These patterns contain critical parameters highlighted within triple brackets (`[[[]]]`). Your task is to determine whether, if a configuration line matches pattern1, there should correspondingly be another line that matches pattern2, and crucially, whether the values within the triple brackets of both patterns are identical. You are required to provide a score ranging from 1 to 10, where 10 indicates the highest confidence in the validity of the relationship. Along with your score, please provide a detailed explanation justifying your assessment.

Background: Each configuration pattern includes parameters specified using square brackets. These parameters can be of various types such as `[num]` for number, `[ip4]` for an IPv4 address, `[ip6]` for an IPv6 address, `[prefix4]` for an IPv4 prefix, etc. Importantly, each configuration pattern also features a key parameter highlighted within triple brackets (`[[[]]]`), which is crucial for determining the relationships between configuration lines.

Parent Context Indicator: Patterns starting with triple quotes (`'''`) indicate their parent context in the configuration. For example, `'''/interface Vxlan[num]/''' vxlan udp-port num` suggests that the pattern `vxlan udp-port num` falls under the `interface Vxlan[num]` context. The portion following the triple quotes is what you should assess.

Example 1:
Input:

```
{
  "pattern1": "'''/community-set [redacted]/'''[num]:[[[num]]]",
  "pattern2": "flow monitor-map IPFIX_[redacted]_IPV[num] cache timeout rate-limit [[[num]]]",
  "transform1": [],
  "transform2": [],
  "condition": {
    "type": "Equality"
  },
  "Roles": [
    "W3"
  ]
}
```

Explanation:
The relationship described involves linking BGP community set values with IPFIX flow monitoring parameters in W3 network devices, potentially for enterprise routing. It specifies that a number representing a rate limit in an IPFIX configuration should match a number in a BGP community set, suggesting that routes tagged with specific communities might have customized flow monitoring settings. This could be relevant in complex networks where traffic from certain routes requires specialized monitoring due to policy, security, or performance reasons. However, the connection between these two configurations — BGP communities and flow monitoring — is unusual and not typically practiced, as they generally serve distinct functions within network management. Thus, while the setup is technically possible and could be useful in specific scenarios, it is not a standard or widely recommended configuration practice.

Score: 3

Example 2:
...

Query input:
...

Figure 11. LLM prompt for scoring the equality contract. We have redacted the exact names used in the configurations. Here, we utilized in-context learning (ICL), a simpler yet effective alternative to fine-tuning, to harness LLMs for performing downstream tasks [18, 24, 27, 29, 32, 43]. It has task-specific natural language prompt to engage the LLM in a targeted evaluation. Each prompt comprises three main parts: (1) Instruction: a directive explaining the task to the LLM (Task, Background, Parent Context Indicator), (2) Demonstrations: examples of the input and expected output (Example 1, Example 2), and (3) Query: the input to analyze (Query input).