New Evolution of Hoyan: Enhancing Scalability, Usability, and Accuracy for Alibaba's Global WAN Verification

Yifei Yuan, Fangdan Ye, Yifan Li, Jingkai Zhang, Mengqi Liu, Yuyang Sang, Ruizhen Yang, Duncheng She, Zhiqing Ye, Tianchen Guo, Xiaobo Zhu, Xinji Tang, Li Jia, Zhongyu Guan, Lingpeng Su, Ci Wang, Ruiyang Feng, Shuo Wu, Zhonghui Xie, Cheng Jin, Peng Zhang, Qing Ma, Xianlong Zeng, Dennis Cai, Ennan Zhai

Alibaba Cloud

Abstract

The network verification system Hoyan has been deployed for Alibaba Cloud's wide-area network (WAN) for years and achieved considerable success in preventing misconfiguration-caused network incidents. However, recent years have seen the emergence of new challenges in *scalability*, *usability*, and *accuracy* for Hoyan. This paper presents the new evolution of Hoyan to address these challenges. First, to support the large increase in the number of routers and prefixes on our WAN, Hoyan's simulation has evolved from a centralized fashion to a distributed framework, which improves the efficiency by 5 times and can scale to $O(10^4)$ routers, millions of prefixes, and billions of flows. Second, to improve Hoyan's usability in checking route change intents, we developed a specification language *RCL*, which supports the easy specification and automatic verification of route change intents. Third, to ensure high accuracy, we enhanced Hoyan's accuracy diagnosis framework, which helped us identify and fix dozens of implementation and modeling issues. Hoyan is used on a daily basis for our WAN. It supports $O(100)$ verification requests each week, prevents $O(10)$ incidents each year, and helps reduce the percentage of misconfiguration-caused network incidents from 56% to 5%.

CCS Concepts

• **Networks** → **Network reliability**; **Network manageability**.

Keywords

Network Verification; Distributed Simulation; Intent-Based Checking

ACM Reference Format:

Yifei Yuan, Fangdan Ye, Yifan Li, Jingkai Zhang, Mengqi Liu, Yuyang Sang, Ruizhen Yang, Duncheng She, Zhiqing Ye, Tianchen Guo, Xiaobo Zhu, Xinji Tang, Li Jia, Zhongyu Guan, Lingpeng Su, Ci Wang, Ruiyang Feng, Shuo Wu, Zhonghui Xie, Cheng Jin, Peng Zhang, Qing Ma, Xianlong Zeng, Dennis Cai, Ennan Zhai. 2025. New Evolution of Hoyan: Enhancing Scalability, Usability, and Accuracy for Alibaba's Global WAN Verification. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3718958.3754343

SIGCOMM '25, Coimbra, Portugal

ACM ISBN 979-8-4007-1524-2/25/09
https://doi.org/10.1145/3718958.3754343

1 Introduction

Alibaba Cloud maintains a global wide area network (WAN) interconnecting its tens of datacenters and also connecting with ISP peers. The WAN supports core services such as e-commerce and cloud computing, serving billions of customers worldwide. Thus, it is extremely important to ensure the availability and reliability of this network.

In the daily operation of the network, one of the biggest challenges is to correctly configure the network with respect to the network operators' high-level intents, especially when making changes to the network, such as changing route policies and replacing routers for maintenance. Indeed, misconfiguration was once the primary root cause of our network incidents, accounting for 56% of them [29]. To address this challenge, we initiated the Hoyan project [52] in 2017, aiming at building a network verification system to check the correctness of configurations for planned network changes before implementing the change into the production network.[1] The primary goal of Hoyan was to accurately simulate the protocols running on the WAN (including BGP and IS-IS), in order to generate the potential RIBs after the planned network change. The primary properties that Hoyan targeted at were the reachability properties for the control plane (*e.g.*, a route X advertised from router A can reach another router B) and data plane (*e.g.*, a packet P sending from router A can reach another router B). Since the deployment of Hoyan in 2018, most reachability related change incidents on our WAN have been successfully detected and thus prevented ahead of time. Afterwards, traffic load related change incidents have become one of the major types of change incidents. In response, starting from 2020, we gradually built traffic simulation and verification capabilities (including two sub-systems Jingubang [26] and Yu [27]) into Hoyan, to meet the verification needs for flow paths (*e.g.*, all flows on path A should be moved to path B) and traffic loads (*e.g.*, no link would be overloaded after the change).

While Hoyan has achieved considerable success in validating the correctness of configurations for the WAN, recent years have seen the emergence of a range of new challenges in *scalability*, *usability*, and *accuracy*, driven by the rapid evolution of the network infrastructure and the ever-increasing operational requirements. In this paper, we present the new evolution of Hoyan, in response to those challenges.

Scalability. Hoyan was initially designed to check the reachability of $O(10^4)$ high-priority prefixes used in several high-risk changes (*e.g.*, new prefix announcement) on our WAN which had hundreds of routers in 2017 [52]. Since then, the scale requirement for Hoyan

[1] While we focus on network changes in this paper, Hoyan is also used in a broad range of scenarios. See §6 for more details.

Table 1: The scale requirement has increased several times while that of run time has reduced significantly.

	# Routers	# Prefixes	# Flows	Run Time Req.
2017	hundreds	$O(10^4)$	N.A.	hours
2024	> 2000	$O(10^6)$	$O(10^9)$	minutes

has increased several times in terms of the network size, and the number of prefixes and flows to be simulated. Table 1 summarizes the change of scale requirements from 2017 to 2024. First, our network grows rapidly in recent years, mainly due to to the recent deployment of our next-generation WAN. As a result, the number of routers increases about three times from a few hundred to more than 2000. Looking forward, our operators wish to use Hoyan to check the WAN with all connected datacenter networks (DCN) in order to check cross-region risks (*e.g.*, a configuration change in DC A should not leak a private route to DC B via the WAN), which would lead to another order of magnitude increase in the network size. Second, in addition to the high-priority prefixes, our operators wish to simulate and verify the behavior of *all* $O(10^6)$ prefixes residing on our network, in order to reason about the complete control plane for all types of network changes (see Table 2 for a complete list of change types). Third, traffic simulation has been developed in Hoyan to reason about the traffic load properties in the last two years [26, 27], which requires Hoyan to efficiently simulate the forwarding of $O(10^9)$ flows. Despite the significant increase in scale, the run time requirement for each change verification request has been significantly reduced from hours to minutes, for the high requirement of urgent change verification cases (see §6 for more details).

This substantial increase in scale poses significant scalability challenges to Hoyan. Figure 1 shows the simulation time of the original Hoyan, which ran on a single server with parallelization, with the increase of prefixes. It required more than 30 minutes to simulate all prefixes for the WAN, which cannot meet our operational requirement. In addition, when extending to DCNs (only the core layer with $O(10^4)$ routers), it could only simulate 30% of the total prefixes, and failed to complete the simulation for 40% of prefixes due to memory exhaustion, indicating its limitations for future extensions.

To support such a large scale, Hoyan's route and traffic simulation has evolved from a centralized approach to a *distributed* framework, [2] where a simulation task runs on a set of servers collectively, and each server hosts only a small set of routes/flows at a time. Our distributed simulation framework is able to complete a simulation task in minutes, which is 5 times faster than the original centralized approach. Meanwhile, it can scale to tens of thousands of routers, millions of input routes and billions of input flows.

Usability. As the intents of network changes differ significantly from one another, basic reachability properties do not suffice to specify the change intents in many change scenarios. Table 2 summarizes all 12 types of changes that Hoyan is required to support, where 9 of them require intent specification beyond reachability.

[2]While the recent work [44] proposed a modular approach to scale BGP control plane verification, it is hard to apply it to our WAN because (1) its assumptions do not hold in our operation (*e.g.*, specifying local constraints) and (2) we need to verify data plane and traffic loads in addition to BGP.

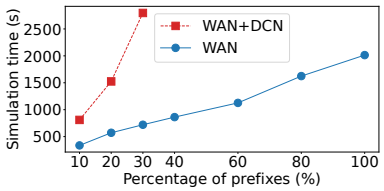


Figure 1: The original Hoyan needs 30 minutes to simulate all prefixes on WAN, and cannot scale to WAN+DCN for future requirements.

As a result, for changes with more complex intents, we used to hard code the verification logic into Hoyan or rely on the operator's manual inspection of the simulation results. Both approaches are time-consuming and error-prone. By analyzing past years' changes, we identified three types of intents: route change intents (*e.g.*, the community of routes with community C1 should be changed to C2), flow path change intents (*e.g.*, flows on path A should be moved to path B), and traffic load change intents (*e.g.*, no overloaded links). Since those intents have fundamentally different abstractions, our operators wish to specify them separately for better usability. In our recent work, we designed Rela [50] for flow path change intent specification; for traffic load change intents, the operators simply specify the intended thresholds. However, neither can specify the *control plane* route change intents, which are required in 6 types of changes (labeled with * in Table 2).

To enhance the usability for control plane change verification, we design a specification language, *RCL*, aimed at offering an intuitive abstraction and user-friendly language for our operators, so they can easily specify route change intents. *RCL* allows to specify the relation between the RIBs before and after a change with simple transformation and evaluation of RIBs. With *RCL*, most route change intents can be easily specified by our operators and then automatically verified, significantly improving the usability of Hoyan and reducing manual errors. *RCL* is used on a daily basis and supports the verification of $O(10)$ changes each week.

Accuracy. To ensure that Hoyan accurately simulates the routing and forwarding behavior of our WAN is particularly challenging in practice. First, given the high complexity and scale of our system, there are a wide variety of sources for inaccurate simulation results. For example, the collected routes may be incomplete, the parsing may be flawed for specific vendors' configuration formats. Second, our WAN deploys a rich set of network features with various vendors (*e.g.*, BGP, IS-IS, segment routing or SR, policy-based routing or PBR). Correctly modeling those features, especially when some of them are interpreted differently by different vendors (which is often referred to as vendor-specific behavior, or VSB [52]), is becoming increasingly difficult. Third, given the large scale and complexity of our WAN, analyzing the root cause of an inaccurately simulated result (*e.g.*, inaccurate simulated traffic load on a link) often requires experts' domain knowledge and manual effort, making it complex and time-consuming.

The original Hoyan supports accuracy diagnosis for route simulation of BGP and IS-IS; we further enhance it to diagnose both route and traffic simulation with SR and PBR, which (1) automatically validates Hoyan's accuracy via cross-validation, and (2) analyzes the

Table 2: All 12 types of network changes together with their example change intents that Hoyan needs to support. Change types shown in bold require more expressive intent specification than reachability; those labeled with * need specification for control plane route changes.

Category	Change type	Example change intents
OS maintenance	**OS upgrade (*)**	All routes remain unchanged (including the prefix and attributes of a route).
	OS patch (*)	
Configuration maintenance	**Route attributes modification (*)**	Routes with a specific attribute value C1 (*e.g.*, communities) should be changed to another attribute value C2, while other routes remain unchanged.
	Static route modification	The static route should reach the given set of routers.
	PBR modification	Flows on path A should be moved to path B.
	ACL modification	All matching flows should be blocked.
Network deployment	**Adding new links (*)**	1. The number of prefix P's next hops should increase; 2. flows traversing the link group should use the new link for ECMP.
	Adding new routers (*)	1. Routes on the new router should be the same as other routers in the group; 2. flows travering other routers in the group should also traverse the new router.
	Topology adjustment	Flows on path A should be moved to path B.
Business demand	New prefix announcement	The target prefix should reach the given set of routers.
	prefix reclamation	The target prefix should not appear on all routers.
	Traffic steering (*)	1. Prefix P's next hops should be changed from A to B; 2. flows on path A should be move to path B; 3. no links are overloaded.

Table 3: Hoyan's key evolution.

	Original [52]	New
Simulation	single server; parallel	distributed
Intents	reachability	+route/path/traffic load intents
Accuracy support	BGP, IS-IS	+SR, PBR

root causes with Hoyan's automation and experts' domain knowledge. This framework helps us identify 9 types of real-world issues and 16 new VSBs, significantly improving the accuracy of Hoyan.

We summarize the key evolution of Hoyan in Table 3. In the following, we first provide essential background on Alibaba Cloud's WAN, and then present the high-level overview of Hoyan's new architecture (§2), followed by detailed technical solutions and experience addressing the aforementioned challenges (§3-5). We also share the deployment experience of Hoyan (§6) and future opportunities (§7).

Ethics. This work does not raise any ethical issues.

2 Background and Overview

We first show Alibaba Cloud's global WAN background, and then present the overview of Hoyan's new architecture.

2.1 Background

Our WAN's control plane is in a distributed setting that runs various protocols, including BGP, IS-IS, and segment routing (SR). Starting from 2023, the WAN has been upgraded rapidly into the next generation based on IPv6 and segment routing over IPv6 (SRv6) [25]. As of December 2024, this WAN contains more than two thousand routers (each router has thousands of lines of configuration commands), hosts millions of prefixes, and carries billions of flows.

The WAN deploys a wide variety of monitoring systems for topology, configuration, route, and traffic monitoring. Below, we describe the route and traffic monitoring systems, in order to provide necessary background for Hoyan's route and traffic simulation.

Route monitoring system. The route monitoring system collects BGP routes from all routers in the WAN. The major approach of this system is to set up BGP connections with all routers, so that the router can advertise its BGP routes to the connected BGP agent. As an ongoing effort, we also actively deploy BGP route monitoring protocol (BMP) [40] onto the WAN, allowing the system to directly collect all routes from each router's BGP RIB.

Traffic monitoring system. The traffic monitoring system uses Netflow [11] and sFlow [35] to collect detailed information of flows received on each interface of a router, such as values of the 5-tuple (*i.e.*, source and destination IP/port, protocol), the timestamp of report time, and the flow's total traffic volume between two reports. The monitoring system also collects the total traffic volume each interface receives in a unit time via Simple Network Management Protocol (SNMP) [14] to monitor the traffic load on each link.

2.2 Overview of Hoyan's New Architecture

Figure 2 shows the high-level overview of Hoyan's new architecture. We first describe the change verification part shown on the left with colored background. At a high level, given a change plan (*i.e.*, planned topology changes and commands for targeted routers' configuration change), Hoyan first runs *simulation* for the *updated* network model (*i.e.*, the topology and configurations after the change) and then runs *verification* on the simulated results (*e.g.*, RIBs) to check whether they satisfy the given change intents. Hoyan relies on a range of monitoring systems (shown in gray boxes) to provide the current network topology, routers' configuration, and input routes/flows for simulation. To improve the efficiency of each change verification request, Hoyan adopts a two-phase design: the pre-processing services (shown in green boxes) run periodically

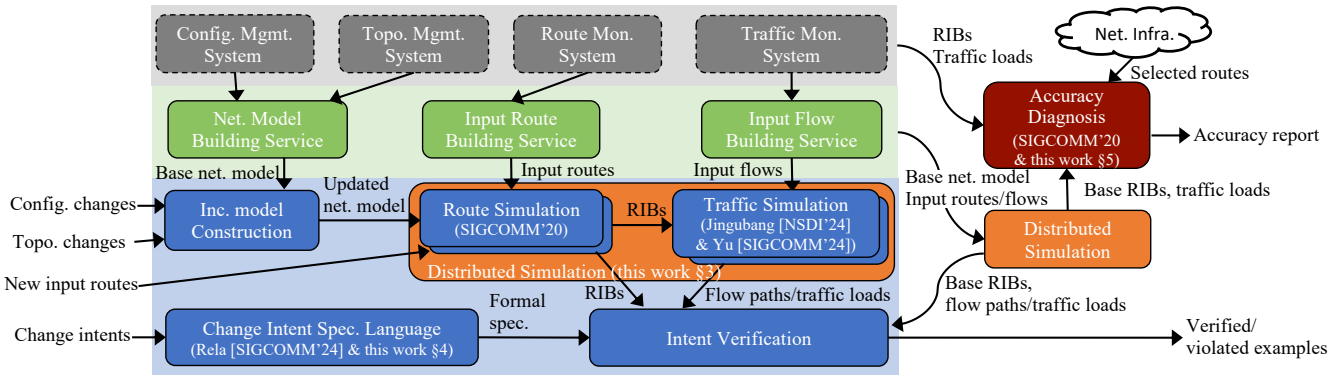


Figure 2: The high-level overview of Hoyan's new architecture. The left part with colored background is for change verification, and the right part is for Hoyan's accuracy diagnosis framework to ensure its high accuracy.

each day while the services for change verification (shown in blue boxes) are triggered for each change verification request.

In the pre-processing phase, the network model building service obtains the current configurations of all routers and parses them into Hoyan's internal model. Then with the current network topology, the service builds a base network model for future change verification uses, avoiding the costly re-parsing of all routers' configurations for each change verification request. The input route/flow building services build the input routes/flows based on the raw data from the corresponding monitoring systems. First, the services load the raw route/flow data from the monitoring systems. Then, they filter the monitored routes/flows based on a set of pre-defined rules (*e.g.*, ignore the routes from a VRF with no external BGP peers) to collect the input routes/flows that are injected into the network. Next, they further process these input routes/flows (see §3 for more details) and finally store them on our cloud storage [1].

In the change verification phase, given a change plan specifying the topology changes and the commands for configuration changes, Hoyan first parses the commands (typically a few hundred to a few thousand lines of commands) and constructs the updated network model incrementally by applying the specified changes to the pre-computed base network model. Then, Hoyan runs route simulation for the updated network model on the pre-computed input routes to generate the potential RIBs for all routers after the change. In the scenario of new prefix announcement, Hoyan also takes the new input routes to be injected into the network for simulation. Next, based on the simulated RIBs, Hoyan takes the pre-stored input flows for traffic simulation, in order to generate the forwarding paths for all flows and the traffic load for all links, in the updated network. The route and traffic simulation is conducted in a distributed fashion as described above, significantly improving the scalability and efficiency of Hoyan. To formally verify the correctness of the network change, Hoyan allows the network operators to specify the intents of the network change in our change intent specification languages, and then automatically checks whether the simulated RIBs, flow paths, and traffic loads can satisfy the formally specified intents. When needed by the specification, Hoyan also loads the pre-computed base RIBs, flow paths, and traffic loads for its verification. In the case where the intents are not satisfied, Hoyan generates a set of concrete counter examples (*e.g.*, routes or flow paths) demonstrating the violation of the intents.

To ensure high accuracy of Hoyan's simulation, we enhance Hoyan's accuracy diagnosis framework, shown on the right with white background. First, each day Hoyan runs the simulation for the based network model on the input routes and flows, to generate the base RIBs, flow paths, and traffic loads. Then Hoyan compares the simulated results with those collected by the monitoring systems. In case of missing data in the route monitoring system, Hoyan also compares with the live network for selected routes. Based on the comparison results, Hoyan outputs an accuracy report, indicating the incorrectly simulated routes and links with inaccurate simulated loads, so that experts can analyze the root cause for them with the help of Hoyan's automation (*e.g.*, building the propagation/forwarding graph of a route/flow).

3 Distributed Simulation

In this section, We first describe Hoyan's original simulation workflow to provide essential background. Then, we present the design of the new distributed simulation framework, followed by the performance evaluation, showing that the distributed simulation framework not only significantly improves the efficiency but also achieves high scalability.

3.1 Original Simulation Workflow

Route simulation. The goal of route simulation is to simulate the propagation of all input routes in order to generate the RIBs of all routers. Hoyan supports all protocols running on our network, such as BGP, IS-IS, and SR, with BGP being the most important yet complex one.

Specifically, for BGP simulation, Hoyan runs a fixpoint algorithm simulating the message-passing process of BGP route propagation. In each round, a router receives a set of incoming routes, processes them according to the ingress route policy such as dropping certain routes and modifying some routes' attributes, installs the routes into its RIB, and finally advertises the updated best route for each prefix to its neighbors after applying the egress route policy (for add-path enabled routers, multiple routes would be advertised [47]). The fixpoint algorithm terminates when no incoming routes are received by any router (within 20 rounds for our WAN).

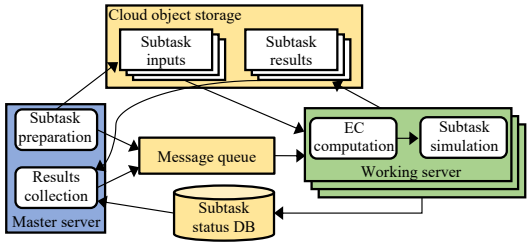


Figure 3: Hoyan's distributed simulation framework.

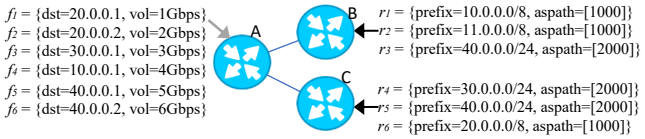


Figure 4: An example network with input routes at router B and C, and input flows at router A.

A key practical challenge of route simulation is efficiently simulating the large number of input routes in our network (*i.e.*, $O(10^7)$ input routes for all $O(10^6)$ prefixes). Our key technique is to identify *equivalence classes* (ECs) of input routes. Particularly, we consider two input routes to be equivalent if (1) they are injected into the same router (in the case that multiple VRFs are configured on the router, it is required that the routes are on the same VRF), (2) their prefixes have the same matching results across all prefix sets in the network and trigger the same aggregate prefixes on all routers, and (3) they have the same values for all BGP attributes (*e.g.*, communities, AS path). By leveraging ECs, Hoyan only needs to simulate one route for each EC and is able to reduce the number of input routes by ~4 times for our WAN.

Traffic simulation. After generating the RIBs for all routers, Hoyan then simulates the forwarding process of all input flows by following each router's RIB in order to generate the forwarding paths for each flow and further compute the traffic load for each link.

Similar to route simulation, Hoyan relies on the equivalence class technique to reduce the prohibitively large number of flows that need to be simulated. Specifically, for efficient computation of ECs, we classify two flows into an EC if their longest-prefix matches on all RIBs are the same; thus, all flows in an EC share the same forwarding paths with one another, and Hoyan only needs to simulate one flow for each EC [26]. In practice, the EC technique is highly effective and can reduce the number of flows by two orders of magnitude.

3.2 Distributed Simulation Framework

From the original simulation workflow, we see that the route/traffic simulation is based on a per-prefix/flow basis, which offers a natural opportunity to conduct the simulation in a distributed fashion by splitting a simulation task into a set of subtasks, where each subtask simulates a subset of inputs and is executed on a server. Since traffic simulation relies on the results from route simulation (*i.e.*, RIBs), how to reduce the dependency of traffic simulation subtasks on route simulation subtasks remains a key challenge for high performance. Below, we first describe the general framework for route/traffic simulation, and then present a heuristic to reduce the dependencies.

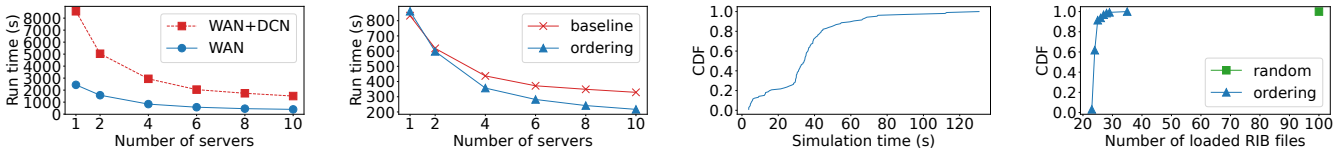
The general framework. Figure 3 shows the general workflow of Hoyan's distributed simulation framework. A simulation task is assigned to a server (referred to as the master server below), which prepares the subtasks by splitting the inputs into multiple disjoint subsets, so that each subset of inputs can be independently simulated as a subtask on a working server. Thus, a route simulation subtask runs the simulation only for a subset of input routes and a traffic simulation subtask runs the simulation only for a subset of input flows. Each subtask's input is uploaded to our cloud object storage as a separate file.

After all subtasks are prepared, the master server pushes a message for each subtask, including its metadata (*e.g.*, reference to the subtask's input and the network snapshot), into a message queue (MQ). Each message is then consumed by a working server listening to the MQ, which then loads the corresponding subtask's input from the cloud storage, runs the subtask by applying the EC technique, and updates the running status of the subtask on a database. Once the subtask is finished, the working server writes the results into the subtask's result file on the cloud storage. For route simulation, the result file contains the RIBs of all routers generated by the input routes. For traffic simulation, the result file contains the forwarding paths of the input flows and the traffic load of links (for the input flows' traffic volume).

Lastly, the master server keeps monitoring the status of all subtasks. When all the subtasks are finished, the master server collects their results if needed (*e.g.*, aggregating each link's traffic load across subtasks). When a subtask fails, the master server resends a message back to the MQ to rerun it.

Reducing subtask dependencies. To simulate the forwarding paths of input flows in a traffic simulation subtask, a working server needs to know the routes that each flow matches on each router. A naive approach would load the resulted RIBs from all route simulation subtasks. However, this approach incurs not only high network I/O cost due to the large volume data transfer, but also high memory footprint on the working server, reducing the performance of simulation. Given a traffic and route simulation subtask T and R respectively, we say T depends on R, if there exists a route in R's generated RIBs that an input flow in T can match. Thus, if T does not depend on R, there would not be *any* route that can match *any* flow in T, and the working server can safely ignore (and thus not to load) R's results without affecting the correctness of the subtask T.

Based on this observation, we design a heuristic to split the input routes and flows in order to reduce the traffic simulation subtasks' dependencies on route simulation subtasks. The high-level idea is to split the input routes and flows following the same ordering (thus we call it *ordering* heuristic). Specifically, for route simulation, we first order all input routes by the last IP address in the prefix (which is done offline in the input route building service), and split the routes into subsets following this order (routes with the same prefix are in the same subset) in the subtask preparation phase. To efficiently determine whether a future traffic simulation subtask may depend on a route simulation subtask R, we compute the range of IP addresses covered by the routes in R, and record it in the



(a) (b) (c) (d)

Figure 5: Distributed simulation for $O(10^6)$ prefixes, $O(10^7)$ input routes, and $O(10^9)$ flows on WAN ($O(10^3)$ routers) and WAN+DCN ($O(10^4)$ routers): (a) The run time for distributed route simulation; (b) The run time for distributed traffic simulation; (c) The CDF of subtasks' run time; (d) The CDF of loaded RIBs in traffic simulation.

subtask DB. Similarly, for traffic simulation, we order all input flows by the destination addresses (which is done offline in the input flow building service), and split them into subtasks following this order (in the subtask preparation phase). To determine whether a traffic simulation subtask T may depend on a route simulation subtask R, we simply check if the range of all input flows' destination address in T overlaps with the pre-recorded range of R.

Example. Consider the example in Figure 4. For route simulation of the input routes r_1-r_6, we first order them and obtain the list [r_1, r_2 r_6, r_4, r_3, r_5]. Suppose we split them into two subtasks R1 and R2, then R1 contains r_1, r_2 and r_6, while R2 contains r_4, r_3 and r_5, based on the ordering of those routes. The range of R1 is simply [10.0.0.0, 20.255.255.255], while that of R2 is [30.0.0.0, 40.0.0.255]. For the traffic simulation of flows f_1-f_6, by ordering the flows based on the destination addresses, we obtain the list [f_4, f_1, f_2, f_3, f_5, f_6]. If we need to split the flows into two subtasks T1 and T2, then T1 contains {f_4, f_1, f_2} and T2 contains {f_3, f_5, f_6}, based on the ordering. Since the range of the destination addresses of {f_4, f_1, f_2} is [10.0.0.1, 20.0.0.2], which only overlaps with the range [10.0.0.0, 20.255.255.255], T1 only needs the results from R1. Similarly, T2 only needs the results from R2.

End-to-end performance evaluation. We evaluate the distributed simulation framework using 10 servers (with 96-core 2.50GHz processor and 791 GB RAM)[3]. The blue line in Figure 5(a) shows the run time of distributed route simulation using various number of working servers for our WAN. We split a route simulation task into 100 subtasks. The simulation time decreases with the increase in the number of working servers, as expected. Particularly, when using 10 servers, the simulation time is only 6.6 minutes, which is ~5 times faster than the original centralized simulation (shown as the blue line in Figure 1), and meets the run time requirement for most changes on our WAN. To evaluate whether this framework can scale to even larger sizes with DCNs, we further measure the end-to-end simulation time on the WAN with all the core-layer routers in our DCNs, which consists of $O(10^4)$ routers in total. As shown in the red dashed line in Figure 5(a) labeled with WAN+DCN, with such a hyper scale, Hoyan can still complete the simulation in 25 minutes on 10 servers (recall that original Hoyan failed the simulation due to out-of-memory), which demonstrates the future-proof scalability of the framework.

We run traffic simulation for $O(10^9)$ flows on our WAN by splitting the task into 128 subtasks (this number of subtasks is chosen to evenly split the flows). The blue line with triangle markers in Figure 5(b) shows the end-to-end simulation time (with the ordering heuristic). When using 10 servers, Hoyan can complete the simulation task in 3.6 minutes, which is 4 times faster than the single-server setting and also meets our operational requirements. For comparison, we further evaluate the simulation time by disabling the ordering heuristic and loading all RIB files (referred to as baseline). The simulation time increases 52% when using 10 servers, showing the effectiveness of the heuristic.

Cause of the diminishing returns. In the distributed simulation evaluation, the end-to-end simulation time did not decrease linearly as the number of servers increases. This is mainly due to the highly uneven simulation time requirements of each subtask. Figure 5(c) plots the cumulative distribution function (CDF) of the run time of all route simulation subtasks; the shortest subtask can be completed within 4 seconds, while the longest takes >2 minutes. As a result, balancing the workload to achieve linear scalability is difficult.

The root cause of the uneven simulation time is that the propagation of input routes on our WAN differ significantly due to the complicated policies deployed on the WAN. For example, routes advertised from ISPs usually propagates only a few hops, while routes originated from the data centers may propagate more than 10 hops. As future work, we plan to split the subtasks to achieve more balanced run time by considering the different characteristics of input routes.

Evaluation of subtask dependency reduction. To evaluate the effectiveness of the ordering heuristic, we measured the loaded RIB files for each traffic simulation subtask. For comparison, we implemented another strategy that partitions flows into subtasks based on a random order (referred to as random). Figure 5(d) shows the CDF of loaded RIB files for our heuristic and the random strategy. The ordering heuristic significantly reduces the number of RIB files a working server needs to load. For > 80% working servers, it only needs to load no more than one third of RIB files, while the highest one loaded less than 40% of the total RIB files. In contrast, when partition subtasks in a random order, each subtask still requires all RIB files from the route simulation, same to the baseline strategy. This is because that each subtask contains a large number of flows (*i.e.*, $O(10^7)$ flows per subtask); when partitioning the subtasks in a random order, there is a high probability that some route in a route simulation subtask may be required by a flow in the traffic simulation subtask. Thus, each traffic simulation subtask depends on all route simulation subtasks with high probability.

[3]Unfortunately, we cannot compare with other tools (*e.g.*, Batfish [15]) on our WAN, since they do not support some major vendors used on the WAN.

device	vrf	prefix	communities	localPref	nexthop	...
A	global	10.0.0.0/24	100:1	**100**	2.0.0.1	...
A	vrf1	20.0.0.0/24	100:1, 200:1	10	3.0.0.1	...
B	global	10.0.0.0/24	100:1	**200**	4.0.0.1	...

device	vrf	prefix	communities	localPref	nexthop	...
A	global	10.0.0.0/24	100:1	**300**	2.0.0.1	...
A	vrf1	20.0.0.0/24	100:1, 200:1	10	3.0.0.1	...
B	global	10.0.0.0/24	100:1	**300**	4.0.0.1	...

Figure 6: Example global RIBs: (Top) base (Bottom) updated.

4 Route Change Intent Specification Language

To improve the usability of Hoyan and avoid the error-prone manual checking process, we work with our operators and design a user-friendly specification language *RCL* to express route change intents for control plane changes. Below, we first introduce *RCL* using an example, followed by its formal definition. We then present use cases to specify real-world route change intents in *RCL*. Lastly, we describe its implementation and evaluation.

4.1 *RCL* by Example

Consider a typical route attribute modification example, which changes a set of routers' route policies with the intent that on all routers: (a) routes with prefix 10.0.0.0/24 should have local preference 300 after the change, and (b) routes with other prefixes should remain unchanged.

Global RIB abstraction. As shown above, a typical route change intent needs to specify the relation between RIBs before and after the change for multiple routers (*e.g.*, all routers' RIB remain unchanged). To allow easy specification of intents covering multiple routers and also match the natural abstraction of router RIBs (as requested by our operators), we design *RCL* based on the *global RIB* abstraction, which essentially collects all routes from all routers into a single table, with the additional `device` and `vrf` fields to indicate a route's location. Figure 6 shows example global RIBs before and after the change above, where there are two routers (A and B), and A has two routes (in two VRFs) while B has one. For convenience, we refer to the global RIB before/after the change as the base/updated global RIB, respectively. For simplicity, throughout this section, we use *RIB* to mean global RIB, and *route* for a row in the global RIB.

Why not SQL? Given the global RIB abstraction, our first attempt was to use the main stream database query language SQL as the specification language. However, since SQL is not designed for network change intent specifications, our operators found it not intuitive and sometimes difficult to specify change intents. For example, to specify the relation between the base and updated RIB, one often needs to use the join operation (potentially after necessary aggregation on the RIBs). However, as non-proficient SQL users, our operators found it hard to correctly write the join condition, making it difficult to correctly specify a wide range of change intents. As another example, to use a query language for specification, one often needs to select the routes that *violate* the intent, and then asserts the selected results is empty. Our operators reported that this abstraction was non-intuitive to them, especially for specifications requiring multiple such assertions, such as some condition should hold for all prefixes. Thus, we decided to work with our operators to design a domain-specific specification language specially for route change intents. Our goal is to make it intuitive and user-friendly for our operators to specify their intents, while still supporting a wide range of intents in common change scenarios.

Comparison	$\odot$	::=	$> \mid \geq \mid = \mid \neq \mid < \mid \leq$
Field Name	χ	::=	`device` | `vrf`
		|	`prefix` | `nexthop` | ...
Route Predicate	p	::=	$\chi \odot val$
		|	χ **contains** val
		|	χ **matches** $regex$
		|	χ **in** $\{val...\}$
		|	p_1 (**and** | **or** | **imply**) p_2
		|	**not** p
RIB Transformation	r	::=	**PRE** | **POST** | $r \Vert p$
RIB Aggregate Func	f	::=	**count**()
		|	**distCnt**(χ) | **distVals**(χ)
RIB Evaluation	e	::=	$val \mid \{val...\}$
		|	$r \triangleright f$
		|	e_1 ($+ \mid - \mid \times \mid /$) e_2
Intent	g	::=	r_1 ($= \mid \neq$) r_2
		|	$e_1 \odot e_2$
		|	$p \Rightarrow g$
		|	**forall** χ: g
		|	**forall** χ **in** $\{val...\}$: g
		|	g_1 (**and** | **or**) g_2
		|	**not** g

Figure 7: *RCL* Syntax

Specification in *RCL*. We first focus on intent (a). An intuitive specification consists of two steps: (1) define the scope of target routes with prefix 10.0.0.0/24, and (2) within this scope, assert that each route in the updated RIB has local preference 300. In *RCL*, it can be easily defined as follows.

```
prefix = 10.0.0.0/24 ⇒
  POST ▷ distVals(localPref) = {300}
```

Here, `p` ⇒ `g` is a *guarded intent*. It means that intent `g` holds on the selected scope of the RIBs defined by `p`, corresponding to the two steps above. Specifically, `prefix = 10.0.0.0/24` is a *route predicate* specifying that the scope of target routes are those with prefix 10.0.0.0/24. **POST** ▷ **distVals**(localPref)= {300} asserts that in the updated RIB (in the selected scope), the set of distinct values of local preference is {300}. Here, **POST** is a keyword referring to the updated RIB and **distVals**(localPref) denotes the set of distinct values of local preference.

For the global RIBs in Figure 6, only the first routes of A and B have prefix 10.0.0.0/24 and will be examined further. Since in the updated global RIB, both routes have local preference 300, (*i.e.*, the set of distinct values is {300}), they satisfy **distVals**(localPref)= {300}; and thus the intent is satisfied.

Similarly, for the second intent, we define the scope of routes using predicate `prefix` ≠ `10.0.0.0/24`, and then specify that the global RIBs do not change for the defined scope, as follows (**PRE** is the keyword refering to the base global RIB).

```
prefix ≠ 10.0.0.0/24 ⇒ PRE = POST
```

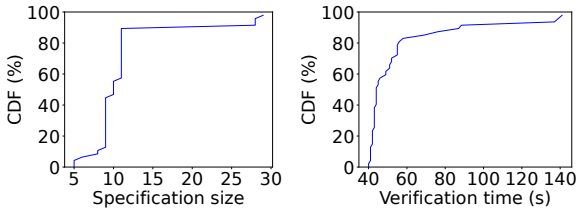


Figure 8: (Left) CDF of *RCL* specification sizes. (Right) CDF of verification time of *RCL* specifications.

In Figure 6, since only the second route of A has a prefix different from 10.0.0.0/24 and the route remains unchanged, the intent is satisfied.

4.2 Formal Definitions of *RCL*

This section provides formal definitions for *RCL*. At a high level, an *intent* in *RCL* evaluates base and updated RIBs to a Boolean value, and the RIBs satisfy the intent when the evaluation result is true. At its core, an intent either compares two entire RIBs or imposes predicates on their aggregate values. To support the former, *RCL* uses *RIB transformation* expressions (*e.g.*, filtering a RIB for targeted routes) to construct the exact RIB to compare. To support the latter, *RCL* uses *RIB evaluation* expressions to evaluate the RIBs to primitive values (*e.g.*, count the number of routes in the updated RIB), so that an intent can simply compare those values.

Figure 7 lists the syntax of *RCL*. We describe the major constructs below and leave the detailed description, semantics, and verification algorithms in Appendix A.

Route predicate: A route predicate p maps a route to a Boolean value. It allows comparison of a field to a primitive value (*e.g.*, `prefix = 10.0.0.0/24`), as well as more complex tests such as the inclusion test (*e.g.*, `communities` **`contains`** `100:1`), the membership test (*e.g.*, `device` **`in`** `{A, B}`), and regular expression matching (*e.g.*, `aspath` **`matches`** `".* 123 .*"`). Route predicates can also compose using Boolean operations.

RIB transformation: RIB transformations allow one to transform the base and updated RIBs to a single RIB. Keywords **`PRE`** and **`POST`** denote the selection of the base and updated RIB, respectively. In addition, the filter transformation ($r \parallel p$) returns a RIB containing all routes from the transformed RIB (obtained from r) satisfying predicate p.

RIB evaluation: RIB evaluation computes a primitive value from the base and updated RIBs, *e.g.*, by applying an aggregate function f. For example, **`POST`** ▷ **`distCnt`**`(nexthop)` computes the number of distinct next hops in the updated RIB. *RCL* also supports counting the number of routes (**`count`**) and collecting distinct values of a field (**`distVals`**).

Intent: an intent g evaluates the base and updated RIBs to a Boolean value. A simple intent compares two RIBs for equality (r_1 $(= \mid \neq)$ r_2), or performs arithmetic comparison of aggregate values ($e_1 \odot e_2$). Examples in §4.1 ($p \Rightarrow g$) illustrate guarded intents. In addition, the grouping intent **`forall`** $\chi : g$ specifies that g holds on all sub-RIBs grouped by each value in field χ. For example, **`forall`** `prefix:` **`POST`** ▷ **`distCnt`**`(nexthop) = 2` specifies that in the updated RIB, each prefix has exactly 2 distinct next hops. Its variant form, **`forall`** χ **`in`** $\{val...\} : g$, is similar, but limits the grouping to the given values $\{val...\}$.

4.3 Use Cases

We demonstrate real-world route change intent specifications in *RCL* below.

Validating unchanged routes. One of the most common route change intent types is that certain routes should not be changed. In a real use case, operators plan to update the exiting point for traffic from region A to a DC (with prefixes 10.0.0.0/24 and 20.0.0.0/24). It should not change the exit point for traffic from region B (suppose B has two routers, R1 and R2). This no-change intent is specified below, where for all routers in {R1, R2} and prefixes in {10.0.0.0/24, 20.0.0.0/24}, there is no change of the next hops on all their best routes.

```
forall device in {R1, R2}:
  forall prefix in {10.0.0.0/24, 20.0.0.0/24}:
    routeType = BEST ⇒
      PRE ▷ distVals(nexthop) =
        POST ▷ distVals(nexthop)
```

Validating the success of route changes. A common type of intents specifies the change effect on the updated RIB. In a real use case, operators plan to change the route policy of a border router, in order to block routes with community 100:1 from being advertised to another region. To validate the success of this change, operators wish to check that the routers R1 and R2 in that region would not have any route containing community 100:1. This can be specified as follows, which checks that for each of R1 and R2, their related routes in the updated RIB do not contain community 100:1.

```
forall device in {R1, R2}:
  POST‖(communities has 100:1) ▷ count() = 0
```

Checking conditional changes. This use case shows a common conditional change pattern that aims to change all routes satisfying some condition. In a real case, operators plan to re-route the traffic whose paths are region A-region B to a new path region A-region C. A straightforward specification is challenging because it is unrealistic to enumerate all exact prefixes with next hops to region B. Instead, using *RCL*, one can implicitly specify those prefixes and the corresponding intended changes using the logical implication and **`forall`**, as shown below (R1 and R2 are the routers in region A, 1.2.3.4 and 10.2.3.4 are the IP addresses of region B and C, respectively). Here, for every prefix, the **`imply`** rule allows the verifier to check its next hops in the updated RIB only if its next hops in the base RIB satisfy the condition (*i.e.*, with the next hop to region B).

```
forall device in {R1, R2}:
  forall prefix:
    (PRE ▷ distVals(nexthop) = {1.2.3.4}) imply
      (POST ▷ distVals(nexthop) = {10.2.3.4})
```

4.4 Implementation and Evaluation

Implementation. Our *RCL* implementation includes its parser, intent verifier, and counter-example generator. The verification of an *RCL* intent follows directly from its semantics. For unsatisfied intents, *RCL* pinpoints the exact basic predicates that are violated and outputs related routes.

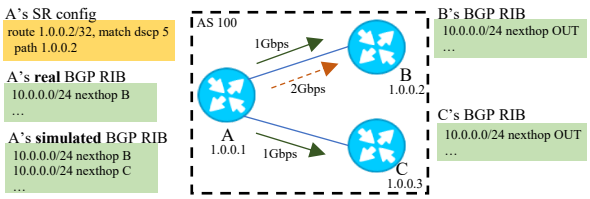


Figure 9: A real-world case demonstrating the root cause analysis workflow.

Evaluation. Our operators use *RCL* on a daily basis for the specification and verification of $O(10)$ changes each week, showing its high usability. Since its deployment in 2024, it has been able to cover most change verification requests that require route change intent specifications (the unsupported intents require concatenation of two RIBs; we plan to support it in the future). Below, we evaluate *RCL*'s performance based on a dataset with 50 specifications written by our operators in real-world change verification requests.

We quantify the size of an intent specification as the number of internal (non-leaf) nodes in its syntax tree. The left figure in Figure 8 shows the CDF of the specifications' size. *RCL*'s specifications of real-world change intents are compact; more than 90% of the specifications are smaller than 15 in its size.

The right figure in Figure 8 shows the CDF of the real-world verification time of those specifications on our WAN (run with a 96-core 2.5GHz processor and 791 GB RAM). More than 80% of them can be verified within 1 minute and all specifications can be verified within minutes, showing that the verification of *RCL* is efficient for production usage.

5 Accuracy Diagnosis Framework

In this section, we present Hoyan's accuracy diagnosis framework, including the workflows of automatic accuracy validation and root cause analysis.

5.1 Automatic Accuracy Validation

On each day, Hoyan runs the route and traffic simulation based on the network configurations, topology, and input routes/flows collected from the monitoring systems. Then Hoyan compares the simulated results with those from the monitoring systems and the live network. For traffic simulation, Hoyan simply compares the simulated traffic load on each link with the monitored traffic load and reports the difference for all links. For route simulation, Hoyan compares all simulated routes with those from the route monitoring system. However, given the fundamental principle of our route monitoring (§2.1), the monitoring system may lack certain important information, such as the ECMP routes for a prefix (only the best route for a prefix is advertised), the next hop of a route (some vendors may modify the next hop even for iBGP advertisements [2]), and attributes that do not propagate via BGP, such as the weight. Therefore, Hoyan also compares simulated routes with the live network by using the show command for selected prefixes (showing all routes is strictly prohibited in the live production network due to the large number of routes). Specifically, it compares routes for high-priority prefixes, such as /0 and /8 prefixes, and those serving key businesses. This hybrid approach not only helped us identify implementation flaws in Hoyan, but also uncovered a list of issues in our monitoring systems.

Table 4: Identified issues during 09/2022 - 03/2023

Description	Percentage
Missing routes	23.08%
Inconsistent topology	19.28%
Inaccurate traffic volume	11.54%
Input route building flaws	9.62%
Configuration parsing bugs	9.62%
Implementation flaws	7.69%
VSBs	5.77%
Unsupported features	3.85%
Nondeterministic convergence	1.92%
Unknown	7.69%

5.2 Root Cause Analysis

The original Hoyan developed a workflow for analyzing the root causes of inaccuracy in route simulation [52]; we have further enhanced it for analyzing traffic simulation results. However, analyzing the root cause of inaccurate traffic simulation results is particularly difficult due to the large scale of our WAN, the high complexity of our system, and the wide variety of root causes. As such, today we mainly rely on a hybrid approach combining Hoyan's automation with network experts' domain-specific knowledge.

This workflow consists of the following steps. (1) Identify the links with a large difference between the simulated and the real traffic load (*e.g.*, > 10% of the link's bandwidth). (2) Identify a large-volume flow traversing that link. (3) Use Hoyan to build the forwarding paths of that flow. (4) Compare each router's forwarding behavior on that flow starting from the router connected with the identified link. (5) Network experts analyze the RIBs and configurations for each router with different forwarding behavior to localize the root cause. Essentially, our workflow attempts to analyze the difference in forwarding for mis-simulated flows which may lead to inaccurate traffic simulation. Step (1) and (2) help us identify such a flow with high likelihood.

Case study. We show a real-world root cause analysis case in Figure 9. First, from automatic accuracy validation described above, Hoyan reported a link A-B where the simulated traffic load was significantly lower than the real one. Then we identified a large-volume flow f traversing A-B alone with its forwarding paths (shown in green) based on Hoyan's simulation results. Next, by comparing the rules matching f on routers B's and A's simulated and real RIBs, Hoyan identified the different forwarding behavior on router A: the simulated RIB of A has two equal-cost BGP routes with next hops B and C for f, respectively, while the real RIB of A has only one with next hop B (shown in red dashed line). To understand why the real RIB only had one route, we investigated deeper into the real RIB and found that the BGP route to C had a higher IGP cost than that of the route to B. Thus, only the one to B was used for forwarding. However, by checking the IGP configuration, we confirmed that the IGP costs of A-B and A-C were equal, so they should have established two ECMP routes as shown in the simulated RIB. Further investigation revealed that A configured

Table 5: Detected vendor-specific behaviors (VSBs)

VSB	Description
missing route policy	Whether route updates are accepted when no policy is defined.
undefined route policy	Whether route updates are accepted when an undefined policy is applied.
default route policy	Whether route updates are accepted when they match no explicit policy.
undefined policy filter	Whether an undefined filter is treated as always matching or not.
no explict permit/deny	Whether a route update is accepted when a matching policy has no explicit permit or deny action.
default BGP preference	The default route preference attribute for iBGP and eBGP.
weight after redistribution	Whether a default weight is set when routes are redistributed into BGP.
adding own ASN	Whether a device's own ASN is added after a policy overwrites the AS path.
common AS path prefix	When aggregating routes without using AS-set, whether the common prefix is added to the AS path.
VRF export policy	Whether a VRF's export policy is applied to global iBGP routes that are leaked into VPNv4.
re-leaking routes	Whether routes leaked into global VPNv4 from VRF should be re-leaked into another VRF based on RT.
redistributing /32 route[1]	Whether /32 routes produced by direct connections can be redistributed.
sending /32 route to peer	Whether /32 routes produced by direct connections can be sent to peers if redistribution is permitted.
IGP cost for SR	Whether a route's IGP cost is treated as 0 when its destination is reached via SR tunnel.
inheriting views	Which configuration options are inherited in sub-views.
device isolation	Whether devices are isolated through policies or specific configurations.

[1] Configuring a non-/32 direct route on a device's interface produces an extra /32 route, which has the following two VSBs.

an SR policy for traffic sending to B, which suggested there might be a mis-simulated behavior for the interaction between BGP, IGP and SR. After consulting A's vendor, we finally located the root cause: A's vendor would change the IGP cost to 0 for SR enabled destinations, so that A preferred the route via SR over the one via IGP. We confirmed with other vendors that they would not change IGP cost for SR policies. Thus, this was a VSB only for vendor A. We further patched our simulation and fixed this issue.

5.3 Real-World Issues

The accuracy diagnosis framework helped us identify and fix a large number of real-world issues affecting Hoyan's accuracy. Table 4 shows the issues we found within 6 months. We categorize those issues into the following three classes.

Monitoring data (row 1-3). Hoyan relies on our internal monitoring systems to provide network topology, configurations, and input routes and flows. The accuracy of the monitoring data is the key to the accurate simulation of Hoyan. However, our monitoring systems may provide inaccurate data due to various practical problems. For example, agents in the route monitoring system may fail and stop collecting routes; the traffic monitoring system may record inaccurate traffic volume for flows due to bugs in vendors' Netflow implementations; the topology management system may have topology data inconsistent with the live network due to failures in the network. We reported those issues to corresponding teams and most of them have been fixed.

Input pre-processing (row 4-5). Hoyan pre-processes the monitoring data to build the input for simulation. When the pre-processing is incorrect, the input to the simulation is also wrong. Two common flaws of the pre-processing occur in parsing routers' configurations and building the input routes. First, Hoyan builds the model of each router by parsing its configuration. However, due to the complexity and variety of different vendors' configuration formats, the implementation of parsing could be incomplete or incorrect, and thus introducing incorrect router models. Second, Hoyan's input route building service uses a list of pre-defined rules to extract the input routes injected into the network from all collected routes. Those rules could be incorrect in some corner cases. For example, one rule is to discard any route with an empty AS path. However, aggregate routes from our data centers may not carry any AS numbers [3]. As a result, those routes would not be considered as input routes mistakenly.

Simulation implementation (row 6-9). The last class of issues are related to Hoyan's simulation. First, implementation bugs/flaws are inevitable in such a large system. For example, Hoyan's early implementation of regular expression matching for AS path was flawed, leading to wrong route policy matching. Second, the wide variety of vendors implement many network features differently. We identified a large set of newly discovered VSBs and report them in Table 5. Third, due to the rapid evolution of our network infrastructure, Hoyan may not model newly introduced features, especially given the large number of VSBs we need to investigate and test. For example, the IS-IS for traffic engineering [28] feature wasn't supported by Hoyan until March 2023, leading to inaccuracy in traffic simulation. Lastly, route simulation faces a fundamental limitation on the convergence of BGP [8, 52], where Hoyan may converge to a network state that is different to the live network.

6 Deployment Experience

Hoyan is used on a daily basis for checking the correctness of changes on our WAN. For high-risk changes that require manual design and experts' review, Hoyan allows the operators to input the change plan and run the verification via our web GUI. For low-risk changes which are executed automatically, Hoyan is integrated in the automation and receives verification requests via our REST API. Each week, Hoyan supports $O(10)$ and $O(100)$ times manually and automatically triggered change verification requests, respectively.[4]

In the past years, Hoyan has identified $O(10)$ change risks each year, some of which could lead to severe incidents. Thanks to Hoyan, those risks are detected and prevented in advance; the

[4]We omit absolute numbers for confidential reasons.

Table 6: The root causes and percentages of change risks detected by Hoyan in 2024.

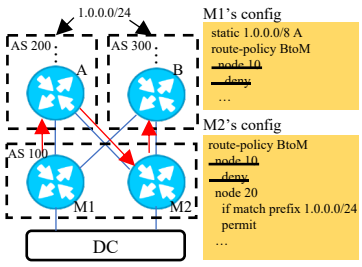
Root cause	Percentage
Incorrect commands	37.5%
Change plan design flaws	34.4%
Existing misconfiguration	15.6%
Topology issues	6.3%
Others	6.2%

misconfiguration-caused network incident has been reduced from 56% to 5% (the remaining is due to out-of-the-scope misconfigurations such as MTU.).

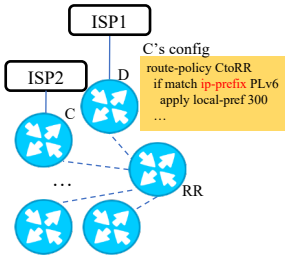
6.1 Case Study

Table 6 summarizes the root causes and the corresponding percentages of the risks identified by Hoyan in 2024. One of the major root causes were the incorrect commands used in change plans, including: (1) typos in the names of routers to be changed or wrong command formats used for a different vendor, which would cause the change to be ineffective on some routers; (2) specifying wrong prefix masks and communities, which would lead to unintended routing behavior or even incidents after the change; (3) typos in the names of community filters and prefix lists, which would trigger unexpected vendor-specific behavior due to referencing undefined definitions. The second major type of risks are caused by the design flaws in the change plans, such as setting inappropriate IS-IS costs, incorrect preference, MED values for BGP routes, which would cause unintended traffic steering. Another root cause were the existing misconfiguration on routers that were not planned to be changed, but a change on a relevant router would trigger the misconfigured behavior and cause network incidents. This type of risks are hard to be detected without Hoyan, since the design of the change plan is correct and the existing misconfiguration does not cause any unexpected behavior before the change. See below for a real case study. The final root cause were the topology issues in the network, such as failed routers and links, due to which the change would cause unexpected traffic overloading. Below, we show two real cases to highlight that Hoyan significantly helps our operators to identify complex change plan risks, which could be hard to be detected manually.

Shifting traffic to new WAN. Shifting traffic to the next-generation WAN is one of the most risky changes on our network. Our operators used Hoyan to ensure the correctness of the change and have prevented several severe change risks. Figure 10(a) shows a real risk Hoyan detected. In this case, our operators intended to shift the traffic from DC to 1.0.0.0/24 to traverse the new WAN via router B instead of router A in the old WAN. Before this change, the operators pre-installed the ingress route policy on M1 and M2 with two policy nodes (shown as node 10 and node 20), where node 10 denied all routes advertised from B and node 20 permitted the route to 1.0.0.0/24 advertised from B (denoted as route R below). In the change, the operator would delete node 10 so that route R would be permitted and used for routing given its higher preference. Our operators used Hoyan to check whether (1) route R would be installed as the best route on both M1 and M2, and (2) the traffic can be successfully shifting to B after the change. Hoyan then ran the



(a) Shifting traffic to new WAN.



(b) Changing ISP exits.

Figure 10: Real-world change risks detected by Hoyan.

route and traffic simulation for the changed network. By checking the intents against the RIBs and traffic paths, Hoyan found violations. First, only M2 installed route R while M1 did not have any route to 1.0.0.0/24. Second, the traffic from M1 to 1.0.0.0/24 would be forwarded through the path M1-A-M2-B (shown in red), causing the link A-M2 to be overloaded. The root cause of this problem was that the pre-installed route policy on M1 missed node 20. As a result, after the change, M1 would still deny route R. However, after R was installed on M2, it would be advertised from M2 to A, which then used it to forward traffic to M2. Since M1 and M2 were in the same AS, A would not further advertise the route to M1 due to AS loop prevention; M1 then used its pre-configured default route 1.0.0.0/8 to forward traffic to A. This case is hard to be identified without Hoyan, since the misconfiguration was introduced before the change, and it had no impact on the network's forwarding behavior. Fortunately, Hoyan was able to identify the risk and avoided a severe incident.

Changing ISP exits. In another scenario shown in Figure 10(b), our operator intended to change the ISP exit for a list of IPv6 prefixes from ISP1 to ISP2. This change would update the route policy on the border router C to apply higher local preferences on the target prefixes before advertising them to the router reflector RR in the region. The operator used Hoyan to check that the next hops of those prefixes should be changed from D to C on all routers in this region, and traffic to those prefixes would be steered to ISP2. Hoyan successfully verified the intent; but surprisingly, Hoyan also found that the links from C to ISP2 would be overloaded. By added another intent that routes of other prefixes should remain unchanged, Hoyan identified a violation that the next hop for other IPv6 prefixes were also changed to C mistakenly. The root cause was that the operator used the wrong command 'ip-prefix' to specify the IPv6 prefix list, instead of the correct one 'ipv6-prefix'. The behavior of this vendor would only check IPv4 prefixes after the 'ip-prefix' command, and permit all IPv6 prefixes by default. As a result, all IPv6 prefixes received by C would have the higher local

preference and resulting in unexpectedly large traffic going through C. After the command was fixed, Hoyan can verify the intents and the change was then implemented without triggering any issues.

6.2 Other Use Cases of Hoyan

Hoyan is also used in large scale for the following scenarios.

Daily configuration auditing. A large set of use cases of Hoyan is to audit the correctness of routers' live configurations. Each day, Hoyan runs the simulation based on the routers' live configuration. Then Hoyan executes dozens of auditing tasks on the simulated RIBs and traffic loads, each defining a high-level invariant that the network should hold, such as the prefixes on all routers in a router group should be the same. Hoyan has successfully detected tens of high-risk live configuration problems, such as inconsistent route policy configurations across different vendors.

Fault-tolerance checking. Our WAN is designed to tolerant a certain degree of router/link failures; Hoyan is used to check if there exists any fault-tolerance issues on the live network. The fault-tolerance checking uses Hoyan's k-failure verification capabilities [27, 52], which can efficiently check if a property holds when no more than k routers/links have failed. Through the checking, Hoyan has identified ~5 fault-tolerance problems, due to causes such as misconfiguration, topology design flaws, and unexpected link maintenance.

Post-change validation. An unexpected use of Hoyan has emerged during the deployment of our next-generation WAN. Since the next-generation WAN introduces new vendors and network features, our operators use Hoyan's simulation results as ground truth to validate if there is any hardware/software issues in the vendors' implementation. Particularly, after a network change is executed, operators run Hoyan's simulation based on the update network and input routes/flows, and compare the results with those on the live network. Since the validation happens after the change is executed, Hoyan must complete the simulation in minutes, such that our operators can roll back the change in time when there is any inconsistency. Without the high accuracy and efficiency of Hoyan, this use case would not be possible.

In addition to the use cases above, we plan to deploy Hoyan for other scenarios such as network design validation, runtime change verification (*i.e.*, verifying the correctness of a change plan during its execution), and traffic engineering validation (*i.e.*, checking if the proposed TE implementation satisfies the high-level TE objective).

7 Lessons and Opportunities

Misconfiguration localization. In the past few years, Hoyan has proven to be useful for checking the correctness and detecting intent violations of network changes. However, localizing the misconfiguration that causes the violation still relies on experts' manual analysis, which is hard and time-consuming for complex violations (*e.g.*, unexpected flow paths), sometimes resulting in delaying a planned change for days. We leave to future work on automatically localizing the misconfiguration for intent violations.

Correct specification of change intents. Our change intent languages were specifically designed to simplify the writing of formal specifications, thereby reducing the likelihood of invalid specifications. However, operators may occasionally write incorrect specifications, leading to wrong verification results. In one such case, our operator correctly specified the intended change effects of a change plan, but missed the critical "others do not change" specification. As a result, although Hoyan verified the specification, the change plan still caused unexpected route changes and had to roll back consequently. Today, we use heuristics to aid the writing of specifications, *e.g.*, by adding a default "others do not change" specification. We believe a promising approach may use more intelligent methods such as LLM to guarantee the completeness and correctness of specifications [31].

Automatic testing framework for accuracy. Accurately simulating the network's behavior is the foundation of Hoyan. Therefore, we have invested a significant amount of time and effort to ensure the accuracy of Hoyan. While our accuracy diagnosis framework facilitate the process to some extent, many key analysis steps still require manual intervention as discussed in §5. Thus, we believe an automatic testing framework that can test the difference between Hoyan and vendor's implementation [32] and analyze the root cause of the difference would be invaluable to ensure the accuracy of a large-scale network verification system like Hoyan.

8 Related Work

Control-plane verification. Many systems were proposed to verify the control plane [4, 6, 13, 15, 17, 18, 22, 37, 39, 41, 46, 48, 52, 56] Especially, Brown et al. shared lessons from Batfish's evolution [9]. Hoyan faces different challenges from Batfish, such as the scalability challenge on production WAN. Tang et al. proposed a modular way to improve the efficiency of control plane verification [44], which is different from our approach based on distributed simulation.

Data-plane and traffic load verification. There is a rich literature on data plane verification [7, 10, 12, 19–21, 23, 24, 30, 36, 38, 42, 45, 51, 53–55] and traffic load verification [26, 27, 43]. Specially, recently proposed tool Tulkun [49] runs data plane verification on routers for high efficiency, which is different from our approach based on distributed simulation.

Formal languages for networks. There were many formal languages proposed for specification of a single network snapshot [5, 7, 16, 33, 34]. Rela [50] specifies the relation of two network snapshots. Our *RCL* shares similar ideas with Rela, but focus on route change intents.

9 Conclusion

This paper presents the new evolution of Hoyan on addressing the emerging challenges on scalability, usability, and accuracy, based on our distributed simulation, the *RCL* route change intent specification language, and our accuracy diagnosis framework. Hoyan is used in Alibaba Cloud on a daily basis and prevented $O(10)$ incidents each year, helping reduce the misconfiguration-caused network incidents from 56% to 5%.

Acknowledgments

We acknowledge all teams within Alibaba Cloud that contributed to the success of Hoyan, including Network Automation, Network Operation, and Network Systems, to name a few. We thank our shepherd, Mina Tahmasbi Arashloo, and the SIGCOMM reviewers for their insightful comments. Ennan Zhai is the corresponding author.

References

[1] [n. d.]. Alibaba Cloud's Object Storage Service (OSS). https://www.alibabacloud.com/en/product/object-storage-service..

[2] [n. d.]. Understand Next Hop Set in iBGP Advertisements on Nexus NX-OS vs Cisco IOS. https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/213402-understand-next-hop-set-in-ibgp-advertis.html..

[3] [n. d.]. Understand Route Aggregation in BGP. https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/5441-aggregation.html..

[4] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 201–219. https://www.usenix.org/conference/nsdi20/presentation/abhashkumar

[5] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. In *41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[6] 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* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 155–168. doi:10.1145/3098822.3098834

[7] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 328–341. doi:10.1145/2934872.2934909

[8] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. 2023. Lessons from the evolution of the Batfish configuration analysis tool. In *Proceedings of the ACM SIGCOMM 2023 Conference* (, New York, NY, USA,) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 122–135. doi:10.1145/3603269.3604866

[9] Matt Brown, Ari Fogel, Daniel Halperin, Victor Heorhiadi, Ratul Mahajan, and Todd Millstein. 2023. Lessons from the evolution of the Batfish configuration analysis tool. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 122–135.

[10] Yiyang Chang, Sanjay Rao, and Mohit Tawarmalani. 2017. Robust Validation of Network Designs under Uncertain Demands and Failures. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 347–362. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/chang

[11] Benoît Claise. 2004. Cisco Systems NetFlow Services Export Version 9. RFC 3954. doi:10.17487/RFC3954

[12] Mihai Dobrescu and Katerina Argyraki. 2014. Software dataplane verification. In *USENIX Conference on Networked Systems Design and Implementation*.

[13] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 217–232. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/fayaz

[14] Mark Fedor, Martin Lee Schoffstall, James R. Davin, and Dr. Jeff D. Case. 1990. Simple Network Management Protocol (SNMP). RFC 1157. doi:10.17487/RFC1157

[15] 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

[16] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: a network programming language. In *16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

[17] 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* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 300–313. doi:10.1145/2934872.2934876

[18] Nick Giannarakis, Ryan Beckett, Ratul Mahajan, and David Walker. 2019. Efficient Verification of Network Fault Tolerance via Counterexample-Guided Refinement. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 305–323.

[19] 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

[20] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 200–213. doi:10.1145/3341302.3342094

[21] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. 2014. *Automated Analysis and Debugging of Network Connectivity Policies*. Technical Report MSR-TR-2014-102. Microsoft. https://www.microsoft.com/en-us/research/publication/automated-analysis-and-debugging-of-network-connectivity-policies/

[22] 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* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 310–328. doi:10.1145/3387514.3405871

[23] 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

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

[25] J Leddy, D Voyer, S Matsushima, and Z Li. 2021. Rfc 8986: Segment routing over ipv6 (srv6) network programming.

[26] Ruihan Li, Fangdan Ye, Yifei Yuan, Ruizhen Yang, Bingchuan Tian, Tianchen Guo, Hao Wu, Xiaobo Zhu, Zhongyu Guan, Qing Ma, Xianlong Zeng, Chenren Xu, Dennis Cai, and Ennan Zhai. 2024. Reasoning about Network Traffic Load Property at Production Scale. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1063–1082. https://www.usenix.org/conference/nsdi24/presentation/li-ruihan

[27] Ruihan Li, Yifei Yuan, Fangdan Ye, Mengqi Liu, Ruizhen Yang, Yang Yu, Tianchen Guo, Qing Ma, Xianlong Zeng, Chenren Xu, Dennis Cai, and Ennan Zhai. 2024. A General and Efficient Approach to Verifying Traffic Load Properties under Arbitrary k Failures. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 228–243.

[28] Tony Li and Henk Smit. 2008. IS-IS Extensions for Traffic Engineering. RFC 5305. doi:10.17487/RFC5305

[29] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. 2018. Automatic life cycle management of network configurations. In *Workshop on Self-Driving Networks (SelfDN)*.

[30] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 499–512. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes

[31] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2024. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. *arXiv preprint arXiv:2401.08807* (2024).

[32] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* (1998).

[33] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. 2012. A compiler and run-time system for network programming languages. In *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[34] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. 2013. Composing Software Defined Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[35] Sonia Panchen, Neil McKee, and Peter Phaal. 2001. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176. doi:10.17487/RFC3176

[36] Aurojit Panda, Katerina Argyraki, Mooly Sagiv, Michael Schapira, and Scott Shenker. 2015. New Directions for Network Verification. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morriset (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 209–220. doi:10.4230/LIPIcs.SNAPL.2015.209

[37] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. 2017. Verifying Reachability in Networks with Mutable Datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 699–718. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/panda-mutable-datapaths

[38] Solal Pirelli, Akvilė Valentukonytė, Katerina Argyraki, and George Candea. 2022. Automated Verification of Network Function Binaries. In *USENIX Symposium on Networked Systems Design and Implementation*.

[39] B. Quoitin and S. Uhlig. 2005. Modeling the routing of an autonomous system with C-BGP. *IEEE Network* 19, 6 (2005), 12–19. doi:10.1109/MNET.2005.1541716

[40] John Scudder, Rex Fernando, and Stephen Stuart. 2016. BGP Monitoring Protocol (BMP). RFC 7854. doi:10.17487/RFC7854

[41] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2020. Probabilistic Verification of Network 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* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 750–764. doi:10.1145/3387514.3405900

[42] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. SymNet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 314–327. doi:10.1145/2934872.2934881

[43] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. 2020. Detecting network load violations for distributed control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 974–988. doi:10.1145/3385412.3385976

[44] 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.

[45] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. 2019. Safely and automatically updating in-network ACL configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 214–226. doi:10.1145/3341302.3342088

[46] Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. 2016. Some Complexity Results for Stateful Network Verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, Marsha Chechik and Jean-François Raskin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 811–830.

[47] Daniel Walton, Alvaro Retana, Enke Chen, and John Scudder. 2016. Advertisement of Multiple Paths in BGP. RFC 7911. doi:10.17487/RFC7911

[48] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. 2012. FSR: Formal Analysis and Implementation Toolkit for Safe Interdomain Routing. *IEEE/ACM Transactions on Networking* 20, 6 (2012), 1814–1827. doi:10.1109/TNET.2012.2187924

[49] Qiao Xiang, Chenyang Huang, Ridi Wen, Yuxin Wang, Xiwen Fan, Zaoxing Liu, Linghe Kong, Dennis Duan, Franck Le, and Wei Sun. 2023. Beyond a Centralized Verifier: Scaling Data Plane Checking via Distributed, On-Device Verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 152–166.

[50] Xieyang Xu, Yifei Yuan, Zachary Kincaid, Arvind Krishnamurthy, Ratul Mahajan, David Walker, and Ennan Zhai. 2024. Relational Network Verification. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 213–227.

[51] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. 2020. Aragog: Scalable runtime verification of shardable networked systems. In *USENIX Conference on Operating Systems Design and Implementation*.

[52] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. 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* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 599–614. doi:10.1145/3387514.3406217

[53] Farnaz Yousefi, Anubhavnidhi Abhashkumar, Kausik Subramanian, Kartik Hans, Soudeh Ghorbani Khaledi, and Aditya Akella. 2020. Liveness verification of stateful network functions. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[54] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*.

[55] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. 2017. A formally verified NAT. In *ACM SIGCOMM (SIGCOMM)*.

[56] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. 2020. Check before You Change: Preventing Correlated Failures in Service Updates. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 575–589. https://www.usenix.org/conference/nsdi20/presentation/zhai

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A *RCL* Formalization

This section presents the complete formalization of *RCL* syntax (§A.1) and semantics (§A.2). Table 7 summarizes notations used in this section.

A.1 *RCL* Syntax

Figure 7 defines the syntax for our specification language, *RCL*. We explain each of its sub-language in detail below.

Route predicate: A route predicate p maps a route to a Boolean value, which is commonly used to specify the scope of targeted routes. A route contains values at each field χ.

- $\chi \odot val$ represents binary comparison, *e.g.*, `prefix = 10.0.0.0/24`.
- χ **contains** val denotes inclusion test, *e.g.*, `communities contains 100:1`. It specifies that the field *communities* is a set and that it contains the value `100:1`.
- χ **in** $\{val...\}$ denotes set membership test, *e.g.*, `device in {A, B}`. It specifies that the value of field *device* is either `A` or `B`.
- χ **matches** *regex* denotes regular expression matches of a string field. For example, `aspath matches ".* 123 .*"` specifies that the field *aspath* is a string and that it contains the substring `123`.
- *RCL* also allows Boolean compositions of predicates.

RIB transformation: RIB transformations map a pair of RIBs, *i.e.*, the base and updated RIBs, to a single RIB. The results are used for specifying RIB comparisons in the intent or for further aggregate value evaluation.

- **`PRE`** and **`POST`** are selector functions that refer to the base and updated RIB, respectively. For example, **`PRE`** `=` **`POST`** specifies that the updated RIB must be identical to the base one.
- $r \parallel p$ denotes the filter transformation. It returns a RIB containing all routes from (the returned RIB of) r satisfying the predicate p. For example, r `||device = A` allows operators to focus their intents on routes associated with device A only.

RIB aggregate func: aggregate functions map a RIB to a primitive value or a set of primitive values. They greatly expand *RCL*'s expressiveness in specifying route changes.

- **`count`** returns the number of routes in a RIB.
- **`distCnt`**(χ) returns the number of *distinct* values in field χ. This is useful, *e.g.*, when specifying the number of distinct nexthops using **`distCnt`**`(nexthop)`.
- **`distVals`**(χ) returns the set of distinct values in field χ. For example, **`distVals`**`(nexthop)` denotes the exact set of nexthops for further comparison.

RIB evaluation: Given a pair of RIBs, RIB evaluation expressions compute aggregate values by applying aggregate functions or performing arithmetic operations. They evaluate to concrete values of string type, numeric type, or a set composed of these types.

- Literal values, such as val and $\{val...\}$, can be directly used as RIB evaluations. They evaluate to values represented by themselves.

Table 7: Summary of *RCL* notations.

	Description
val, v	Concrete values of numeric or string type
$\{val...\}, \{v_1...v_n\}$	Sets of concrete values
M, N	RIBs that an intent (or sub-intent) is specified on
$\|M\|, \|\{val...\}\|$	Computes the size of a RIB (number of routes) or a set
τ	A route in a RIB
χ	A filed name in the RIB, *e.g.*, device, vrf, prefix, etc.
τ_χ	The value at field χ in route τ
$\textbf{re\_match}(s, regex)$	Matches a string s against regular expression $regex$. It returns a Boolean value.
$\textbf{filter}_p(M)$	Filters a RIB by the route predicate p. It returns a new RIB consisting of $\{\tau \in M \mid [\![p]\!](\tau) = true\}$
$[\![p]\!](\tau)$	Evaluation of a route predicate on τ. It returns a Boolean value.
$[\![r]\!](M, N)$	Evaluation of a RIB transformation. It returns a RIB.
$[\![e]\!](M, N)$	Evaluation of RIB aggregate values. It returns either a string, a number, or a set composed of these types.
$[\![g]\!](M, N)$	Evaluation of intent. It returns a Boolean.

- $r \triangleright f$ applies an aggregate function f after performing RIB transformation r. For example, `POST ▷ distCnt(nexthop)` computes the number of unique nexthops in the updated RIB.
- e_1 $(+ \mid - \mid \times \mid /)$ e_2 denotes arithmetic operations on numeric RIB evalutions.

Intent: finally, an intent g is the top-level construct for specifying route changes. It evaluates to a Boolean value.

- r_1 $(= \mid \neq)$ r_2 checks the equality (or inequality) of two entire RIBs. This is useful, *e.g.*, when comparing the base and updated RIBs.
- $e_1 \odot e_2$ compares the results of RIB evaluations. For example, `POST▷ distCnt(nexthop)` $= 8$ specifies that the number of distinct nexthops in the updated RIB must equal 8.
- $p \Rightarrow g$ denotes a guarded intent. It narrows the scope of intent g to only target routes from the base and updated RIBs satisfying predicate p. For example, `prefix = 10.0.0.0/24` $\Rightarrow g$ limits the intent's scope to routes with $prefix$ `10.0.0.0/24`.
- **forall** $\chi : g$ specifies the intent on individual sub-groups of the base and updated RIBs. Within each of the sub-group, routes contain identical values at field χ. For example, `forall prefix: POST ▷ distCnt(nexthop)` $= 2$ specifies that in the updated RIB, each $prefix$ is associated with exactly 2 distinct nexthops.
- **forall** χ **in** $\{val...\} : g$ is similar to the previous rule, but provides more flexibility. It allows operators to specify the set of values at field χ to consider. Routes containing other values at field χ are out of scope regarding intent g.
- g_1 (**and** | **or**) g_2 and **not** g_1 denote the Boolean composition of sub-intents.

A.2 *RCL* Semantics

Figure 11 illustrates evaluation rules for different *RCL* expressions. On the high-level, a route change intent either compares entire RIBs (computed by RIB transformations r), specifies a predicate over aggregate values (computed by RIB evaluations e), or composes sub-intents. We explain them in further detail below.

Route predicate. A route predicate defines a function mapping routes to Boolean values.

(a) Route Predicate p

$$[\![\chi \odot val]\!](\tau) \triangleq \tau_\chi \odot val$$
$$[\![\chi \textbf{ contains } val]\!](\tau) \triangleq val \in \tau_\chi$$
$$[\![\chi \textbf{ in } \{val...\}]\!](\tau) \triangleq \tau_\chi \in \{val...\}$$
$$[\![\chi \textbf{ matches } regex]\!](\tau) \triangleq \textbf{re\_match}(\tau_\chi, regex)$$
$$[\![p_1 \textbf{ and } p_2]\!](\tau) \triangleq [\![p_1]\!](\tau) \wedge [\![p_2]\!](\tau)$$
$$[\![p_1 \textbf{ or } p_2]\!](\tau) \triangleq [\![p_1]\!](\tau) \vee [\![p_2]\!](\tau)$$
$$[\![p_1 \textbf{ imply } p_2]\!](\tau) \triangleq \neg[\![p_1]\!](\tau) \vee [\![p_2]\!](\tau)$$
$$[\![\textbf{not } p_1]\!](\tau) \triangleq \neg[\![p_1]\!](\tau)$$

(b) RIB Transformation r

$$[\![\textbf{PRE}]\!](M, N) \triangleq M$$
$$[\![\textbf{POST}]\!](M, N) \triangleq N$$
$$[\![r \parallel p]\!](M, N) \triangleq \textbf{filter}_p([\![r]\!](M, N))$$

(c) RIB Evaluation e

$$[\![val]\!](M, N) \triangleq val$$
$$[\![\{val...\}]\!](M, N) \triangleq \{val...\}$$
$$[\![r \triangleright \textbf{count()}]\!](M, N) \triangleq |[\![r]\!](M, N)|$$
$$[\![r \triangleright \textbf{distVals}(\chi)]\!](M, N) \triangleq \{\tau_\chi \mid \tau \in [\![r]\!](M, N)\}$$
$$[\![r \triangleright \textbf{distCnt}(\chi)]\!](M, N) \triangleq |\{\tau_\chi \mid \tau \in [\![r]\!](M, N)\}|$$
$$[\![e_1 + e_2]\!](M, N) \triangleq [\![e_1]\!](M, N) + [\![e_2]\!](M, N)$$
$$[\![e_1 - e_2]\!](M, N) \triangleq [\![e_1]\!](M, N) - [\![e_2]\!](M, N)$$
$$[\![e_1 \times e_2]\!](M, N) \triangleq [\![e_1]\!](M, N) \times [\![e_2]\!](M, N)$$
$$[\![e_1 / e_2]\!](M, N) \triangleq [\![e_1]\!](M, N) / [\![e_2]\!](M, N)$$

(d) Intent g

$$[\![r_1 = r_2]\!](M, N) \triangleq [\![r_1]\!](M, N) = [\![r_2]\!](M, N)$$
$$[\![r_1 \neq r_2]\!](M, N) \triangleq [\![r_1]\!](M, N) \neq [\![r_2]\!](M, N)$$
$$[\![e_1 \odot e_2]\!](M, N) \triangleq [\![e_1]\!](M, N) \odot [\![e_2]\!](M, N)$$
$$[\![p \Rightarrow g]\!](M, N) \triangleq [\![g]\!](\textbf{filter}_p(M), \textbf{filter}_p(N))$$
$$[\![\textbf{forall } \chi \textbf{ in } V : g]\!](M, N) \triangleq \bigwedge_{i=1}^{n} [\![g]\!](\textbf{filter}_{p_i}(M), \textbf{filter}_{p_i}(N)),$$
$$where \quad p_i \equiv \chi = v_i \quad V = \{v_1...v_n\}$$
$$[\![\textbf{forall } \chi : g]\!](M, N) \triangleq \bigwedge_{i=1}^{|V|} [\![g]\!](\textbf{filter}_{p_i}(M), \textbf{filter}_{p_i}(N)),$$
$$where \quad p_i \equiv \chi = v_i, \quad v_i \in V$$
$$V = \{\tau_\chi \mid \tau \in M \vee \tau \in N\}$$
$$[\![g_1 \textbf{ and } g_2]\!](M, N) \triangleq [\![g_1]\!](M, N) \wedge [\![g_2]\!](M, N)$$
$$[\![g_1 \textbf{ or } g_2]\!](M, N) \triangleq [\![g_1]\!](M, N) \vee [\![g_2]\!](M, N)$$
$$[\![\textbf{not } g]\!](M, N) \triangleq \neg[\![g]\!](M, N)$$

Figure 11: Evaluation rules for *RCL*.

- $[\![\chi \odot val]\!](\tau)$ denotes binary comparisons between a field and a concrete value. It evaluates to $\tau_\chi \odot val$.
- $[\![\chi$ **contains** $val]\!](\tau)$ denotes membership test. It evaluates to true if and only if τ_χ is a set and contains val.
- $[\![\chi$ **in** $\{val...\}]\!](\tau)$ denotes inclusion test. It evaluates to true if and only if τ_χ is included in $\{val...\}$.
- $[\![\chi$ **matches** $regex]\!](\tau)$ performs regular expression matches on string fields. It evaluates to $\textbf{re\_match}(\tau_\chi, regex)$, which returns true if and only if the entire τ_χ matches pattern $regex$.

- Predicates can be composed of sub-predicates through Boolean operations. Their evaluation follows standard logical definitions. For example, $[\![p_1 \textbf{ and } p_2]\!](\tau)$ first evaluates individual sub-predicates on τ, then returns their conjunction as the result.

RIB transformation. They take a pair of RIBs (base and updated, respectively) and return a single transformed RIB for further specification.

- **PRE** and **POST** are keywords denoting the base and updated RIB, respectively. For example, $[\![\textbf{PRE}]\!](M, N)$ evaluates to M.
- $[\![r \parallel p]\!](M, N)$ first evaluates the tranformation r, then uses the **filter** function to select routes satisfying the predicate p and compose them as a new RIB. The final result correspond to the set of routes $\{\tau \in [\![r]\!](M, N) \mid [\![p]\!](\tau) = true\}$.

RIB evaluation. They take a pair of RIBs and compute aggregate values. The return value is either a string, a number, or a set composed of these types.

- $[\![r \triangleright \textbf{count()}]\!](M, N)$ first performs RIB transformations to obtain a single RIB, then counts the number of routes in it, *i.e.*, $|[\![r]\!](M, N)|$.
- $[\![r \triangleright \textbf{distVals}(\chi)]\!](M, N)$ returns the set of distinct values at field χ of the RIB transformation result. This is achieved by computing the set $\{\tau_\chi \mid \tau \in [\![r]\!](M, N)\}$.
- $[\![r \triangleright \textbf{distCnt}(\chi)]\!](M, N)$ returns the number of distinct values at field χ of the RIB transformation result. This is achieved by computing the size $|\{\tau_\chi \mid \tau \in [\![r]\!](M, N)\}|$.
- Arithmetic operations of numeric RIB evaluations, such as $[\![e_1 + e_2]\!](M, N)$, are evaluated following standard arithmetic definitions.

Intent checking. This top-level construct takes a pair of base and updated RIBs and returns a boolean value.

- $[\![r_1 \ (= \mid \neq) \ r_2]\!](M, N)$ denotes the equality checking between entire RIBs. It first evaluates the two transformations, *i.e.*, $[\![r_1]\!](M, N)$ and $[\![r_2]\!](M, N)$, then performs the equality checking between the two transformed results.
- $[\![e_1 \odot e_2]\!](M, N)$ denotes the binary comparison of aggregate values. It first carries out the two RIB evaluations separately, then computes $\odot$ on their results.
- $[\![p \Rightarrow g]\!](M, N)$ represents a guarded intent. It uses the **filter** function to select routes satisfying the predicate p, then evaluates the intent g on $(\textbf{filter}_p(M), \textbf{filter}_p(N))$.
- The grouping intent $[\![\textbf{forall } \chi \textbf{ in } \{v_1 ... v_n\} : g]\!](M, N)$ specifies the conjunction of n sub-intents. For each of them, *RCL* evaluates g on $(\textbf{filter}_{\chi=v_i}(M), \textbf{filter}_{\chi=v_i}(N))$. Thus, the overall intent holds if and only if g holds on every pair of base and updated RIBs filtered by $\chi = v_i$.
- The expression $[\![\textbf{forall } \chi : g]\!](M, N)$ offers a short-hand version of the previous one, when operators want to specify g on all distinct values at field χ of the original M and N, *i.e.*, $\{\tau_\chi \mid \tau \in M \vee \tau \in N\}$.
- Finally, intents compose with each other through boolean operations. For example, $[\![g_1 \textbf{ or } g_2]\!](M, N)$ evaluates to the disjunction of $[\![g_1]\!](M, N)$ and $[\![g_2]\!](M, N)$.

A.3 *RCL* Verification

The *RCL* intent verification happens after the distributed RIB simulation, as illustrated in Figure 2. It loads the entire concrete base and updated RIBs, then executes the core algorithm below. We use a data class Row to contain values for each column in a row, and we represent the whole RIB as a set of Rows. As shown in Figure 8, the overall verification fits on a single machine (with 96 cores and 791 GB RAM) and can finish within minutes.

Algorithm 1 shows function `checkIntent`, which evalutes an intent I on given base and updated RIBs, M and N. It follows the semantics in Figure 11 and adopts a syntax-guided approach to break down the checking into smaller operations. Here, `evalPredicate`(p, τ) is a boolean function checking whether the predicate p holds on values contained by row τ. `distVals`(χ, M) is an aggregation function that collects the set of distinct values at field χ in RIB M. We omit their implementations here due to their straightforwardness.

Algorithm 1 Evaluating an *RCL* intent

```
function checkIntent(I, M, N)
    match I with
    | r1 = r2 →
        r1' ← transform(r1, M, N)
        r2' ← transform(r2, M, N)
        return ribEQ(r1', r2')
    | e1 ⊙ e2 →
        v1 ← eval(e1, M, N)
        v2 ← eval(e2, M, N)
        return v1 ⊙ v2
    | p ⇒ g →
        M' ← filter(p, M)
        N' ← filter(p, N)
        return checkIntent(g, M', N')
    | forall χ : g →
        valueSet ← distVals(χ, M ⊎ N)
        return checkForAll(I, χ, valueSet, M, N)
    | forall χ in V : g →
        return checkForAll(I, χ, V, M, N)
    | g1 and g2 →
        return checkIntent(g1, M, N) ∧ checkIntent(g2, M, N)
end function

function checkForAll(I, χ, valueSet, M, N)
    for all v ∈ valueSet do
        Mv ← filter(χ = v, M)
        Nv ← filter(χ = v, N)
        if ¬checkIntent(I, Mv, Nv) then
            return False
        end if
    end for
    return True
end function

function filter(p, M)
    R ← ∅
    for all τ ∈ M do
        if evalPredicate(p, τ) then
            R.Add(τ)
        end if
    end for
    return R
end function
```

Algorithm 2 illustrates functions `transform` and `eval`. They also follow the semantics in Figure 11 and adopt a syntax-guided approach.

Algorithm 2 RIB transformations and evaluations

```
function TRANSFORM(r, M, N)
    match r with
    | PRE →
        return M
    | POST →
        return N
    | r_1 ‖ p →
        R_1 ← TRANSFORM(r_1, M, N)
        return FILTER(p, R_1)
end function

function EVAL(e, M, N)
    match e with
    | val →
        return val
    | V →
        return V
    | r ▷ f →
        R ← TRANSFORM(r, M, N)
        return F(R)
    | e_1 + e_2 →
        return eval(e_1, M, N) + eval(e_2, M, N)
    ......
end function
```