MirrorNet: High-fidelity and Scalable Network Emulation for Software-defined WAN

Congcong Miao[1]*, *Yuejie Wang*[2]*, *Jianming Wang*[1], *Xuefeng Ji*[1], *Guozhi Shan*[1], *Sirui Li*[1], *Pan Fang*[1], *Yanke Zhang*[1], *Jialin Li*[3], *Xianneng Zou*[1], *Guyue Liu*[2]

[1]*Tencent* [2]*Peking University* [3]*National University of Singapore*

Abstract

Operating a large-scale WAN reliably is becoming increasingly challenging due to the surge in traffic volumes, and the growing complexity of both software and hardware. In this paper, we introduce *MirrorNet*, our production-grade emulation framework designed to mirror a software-based WAN. Unlike traditional emulators and simulators that access only a *partial* set of network information, *MirrorNet* functions as a comprehensive *twin* of the production network, encompassing the controller, data plane, and network traffic. Our key challenge lies in striking a balance between the requirements for a fine-grained and high-fidelity emulation, scalability, and resource efficiency. To address these, we have developed a multi-faceted approach: i) we employ an incremental storage and replay method to reconstruct the historical production network at a second-by-second level; ii) we propose a network update strategy that maintains consistent alignment between the emulation and production networks; and iii) we design a custom orchestrator capable of rapidly deploying one or more large-scale emulation networks, which can operate concurrently to expedite testing. *MirrorNet* has been deployed in TWAN for over 2 years and integral in our daily WAN management tasks, aiding in troubleshooting, parameter tuning, testing, and capacity assessment.

1 Introduction

Our wide area network (WAN) connects hundreds of data centers, runs millions of applications (e.g., real-time messaging, social media, online meeting, cloud gaming), and serves billions of users. To enable flexible and centralized control, like many large service providers (Google [25,26], Microsoft [24], Meta [14]), we have adopted centralized software-defined controllers to manage our WAN. The controller is responsible for configuring $O(10)$ network parameters (e.g., reserved bandwidth limit, path utilization bound [14]) and scheduling tens of Tbps traffic.

Running such a large network *reliably* presents significant challenges. This includes handling routine planned changes like upgrades and new deployments, dealing with unexpected faults such as hardware failures and software bugs, and managing constant traffic fluctuations across $O(100)$ devices and links. Each of these scenarios has the potential to degrade performance or lead to complete unavailability, adversely affecting the user experience.

To enhance reliability, it is essential to both *proactively* assess the potential impact of planned changes and to carry out thorough *retrospective* investigations of unforeseen faults. Unfortunately, none of the existing tools can provide this *'bidirectional'* forward-planning and backward-analysis capabilities to manage production WANs.

Existing network emulation tools (e.g., CrystalNet [33], Kollaps [21]), simulation tools (e.g., Risk Simulation [46], DONS [16]), and verification tools (e.g., Batfish [12], Minesweeper [9], Hoyan [48]) can be used to validate planned changes before deployment. However, these tools only have access to *partial* production network information. For example, CrystalNet [33] is tailored for data center networks, utilizing static network topologies and router configurations as input. However, it does not incorporate real traffic data or lacks dynamic event handling. This partial view can lead to potential discrepancies between simulated outcomes and actual network behavior, resulting in unforeseen issues after deployment. Crescent [17] is an emulation tool for WAN, but it focuses on evaluating network behaviors under failure scenarios and also fail to consider TE and traffic demands. On the other side, existing monitoring and troubleshooting tools [8, 11, 36, 49] often have limited visibility into all components of a large network, and thus cannot capture the root cause of all faults. Even when faults are identified, replicating these issues and testing proposed solutions proves to be a challenging task [14].

In this paper, we present *MirrorNet*, a scalable and high-fidelity WAN emulation framework designed to tackle these challenges. Here, high fidelity means that the emulation framework accurately reconstructs the production network at any

*Both authors contributed equally to this work.

historical time. As Figure 5 shows, by mirroring the production network architecture (e.g., network devices, configurations, and network traffic), *MirrorNet* essentially acts as a *twin* of the production network. The system's orchestrator dynamically updates the emulation network in real time, reflecting *every second*'s changes in the production network. Therefore, MirrorNet can emulate the network state at any specific time. This capability enables root cause analysis of past incidents and allows for safe testing of configuration changes in a virtual environment before they are deployed to the production network. With capabilities of backward-analysis and forward-planning, *MirrorNet* has been used in three main types of daily WAN management tasks: i) tracing and reproducing unexpected events; ii) simulating configuration and architectural changes; iii) assessing and adjusting network capacity.

To achieve *high fidelity* and *scalability* in mirroring our large-scale production network in *real-time*, our framework has made the following contributions:

- **Incremental data storage and replay:** To enable precise reconstruction of past network states, it is necessary to store a vast amount of constantly changing network data, which can require substantial storage (e.g., approximately 1300 GB per day). To reduce storage resources while maintaining fine-grained replay, we employ a novel *incremental replay* method. This method involves storing changes for less frequently updated data as snapshots, while proactively capturing updates of high-frequency data as events. To accurately reconstruct the network at any historical point, we utilize the closest snapshot and replay the subsequent events (§ 3.2).
- **Consistent network updates:** As we incorporate the TE into the WAN emulation, we need to incorporate data from multiple sources that is closely related to TE. To correctly reproduce the production network, it is essential to consider the strong dependencies among various data types. For instance, to emulate a tunnel, it is important to consider factors such as the underlying network link, the traffic traversing the tunnel, and any events potentially impacting it. We propose a *safe update approach* that systematically determines an operational sequence, ensuring strong consistency between the production and emulation networks (§ 3.3).
- **Scalable emulation deployment:** To efficiently and scalably emulate large-scale networks, we adopt two key techniques. First, to enable emulating large network scales, we develop a custom orchestrator to manage virtual devices and links. It is designed to overcome the inefficiencies of existing solutions, such as OpenStack [5] and GNS3 [1], which are too slow to deploy large networks. Our orchestration solution significantly speeds up the setup process, thus enabling on-demand emulation. Second, since WAN emulator should perform various tasks in a timely manner, we propose a *multi-task* emulation approach, which is capable of concurrently running multiple network emulation tasks, significantly accelerating the network testing process. Utilizing a unique traffic translator mechanism, it also ensures efficient storage use, enabling multiple emulations to concurrently access and utilize a shared traffic dataset (§ 3.4).

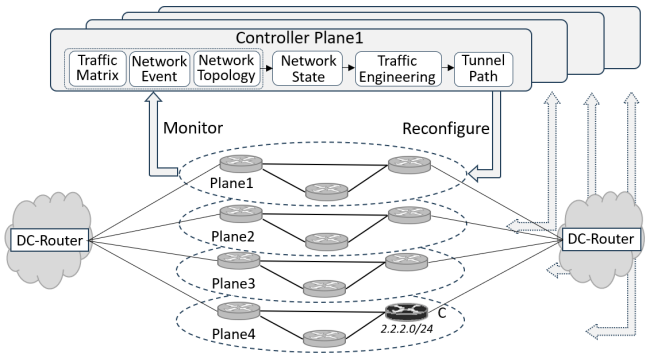


Figure 1: Multi-Plane Architecture of TWAN: An example of four planes interconnecting three regions.

MirrorNet has been deployed in our production network TWAN for over 2 years. Compared to traditional systems, *MirrorNet* enables to localize failures in minutes, orders of magnitude faster while promoting discovering previously untraceable failures, and helps to reach optimal capacity settings by satisfying 1.4× more demand for one-link failure scenarios at rush hours, which is 3.5× better. Meanwhile, *MirrorNet* can help obtain optimal parameters with minimal operations with the production network and ensure nearly 100% successful network evolutions. Furthermore, *MirrorNet* presents high-precision mirroring ability (i.e., one second) and high scalability without introducing much overhead (§ 5). Furthermore, we have learned important operational lessons throughout years of running *MirrorNet* in production (§ 6). To the best of our knowledge, we are the first to introduce the real-time emulation of the service provider WAN, related practical concerns, and our production solution to academia.

This work does not raise any ethical issues.

2 Background and Motivation

In this section, we first describe our existing WAN architecture (§ 2.1), and then show why we need the *MirrorNet* (§ 2.2).

2.1 Multi-plane based WAN Architecture

Our private WAN interconnects hundreds of data center (DC) sites across various geographical regions. Its primary role is to carry inter-DC traffic, which has been growing exponentially over the past three years. To support evolving demands and enhance resilience, we adopt a multi-plane architecture. As shown in Figure 1, the physical network topology is divided into multiple parallel topologies, called planes. Cross-DC traffic is distributed across these planes using flow-level ECMP. Each plane has a dedicated software-based central controller, runs distributed agents on each router, and manages a proportion of the overall traffic. The multi-plane architecture enables a phased rollout of upgrades and maintenance activities, reducing the risk of network-wide disruptions.

Centralized Traffic Engineering: Different from data center

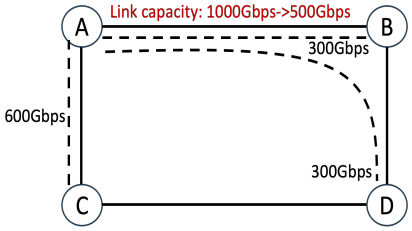


Figure 2: Link disruption causing packet loss of 100 Gbps.

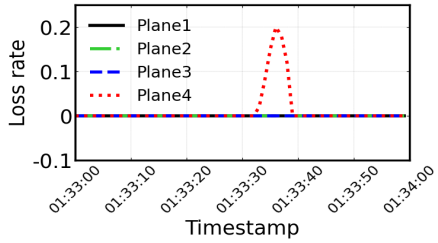


Figure 3: 0.2% of packet loss during the plane isolation.

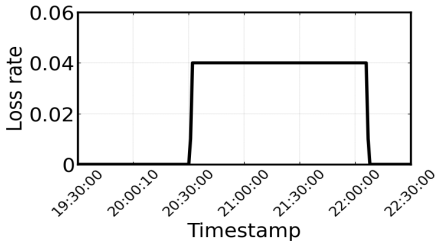


Figure 4: Persistent packet loss due to a large portion of link capacity loss.

networks where the network utilization is quite low and routing solutions like ECMP are traffic oblivious [27, 34, 45, 50], WANs involve a controller in each plane to perform centralized traffic engineering (TE) which takes real-time traffic demands into account to continuously adjust traffic scheduling decisions, so that scarce inter-DC bandwidth can be fairly shared among competing applications. The control loop of TE includes three key steps: i) *State Monitoring*: The controller polls agents on each router to discover network topology and measure bandwidth demands among sites. The topology is represented as a directed graph with each site as a node, and site-to-site connectivity as edges. The bandwidth demands between each pair of sites are represented by a traffic matrix, which is generated based on polling byte counters on each router; ii) *Run TE algorithms*: Based on the network topology and traffic matrix value, the TE algorithm allocates paths for every pair of sites to optimize specific goals, such as minimizing packet loss or link utilization. Multiple paths may exist between a pair of sites to balance load and minimize congestion; iii) *Reconfigure routers*: Finally, by comparing the paths output by the TE algorithms with the existing paths, the controller identifies changed paths and reconfigures the network using tunneling (e.g., IP encapsulation).

The TE control loop, i.e., TE action, operates periodically (e.g., every 5 minutes) or on-demand to adapt to network failures, fluctuating traffic demands, and architectural changes. Essentially, it plays a pivotal role in optimizing network resource usage, ensuring resilience under various conditions, and enhancing the overall user experience.

2.2 Why Do We Need *MirrorNet*

Since network utilization in data center networks is quite low and the routing mechanism is traffic oblivious, the primary goal of emulators like CrystalNet [33] is designed to check network connectivity. However, in WANs, the traffic engineering is used to make efficient utilization of bandwidth resource, leading to changing network states. This phenomenon makes it impossible to directly apply traditional emulators such as CrystalNet that is designed for data centers for emulation. In this section, we present four representative scenarios that highlight the issues of relying solely on a production WAN environment for network management. These scenarios underscore the necessity of developing a control plane simulation framework to address the limitations of previous emulators.

Case 1—Untraceable Network Failures: As a large scale backbone network, we typically encounter $O(10)$ failures daily, stemming from both internal devices and external service providers. These failures often trigger the TE control system to initiate traffic rescheduling. However, tracking the root causes of these rescheduling events is a significant challenge. Consider a specific incident in our network, illustrated in Figure 2, which includes a segment of our backbone topology with four sites. Each network link in this segment has a uniform bandwidth of 1000 Gbps. There are multiple tunnels on these links, the AB tunnel is assigned 300 Gbps, the ABD tunnel is also allocated 300 Gbps, and the AC tunnel is set at 600 Gbps.

An issue arose when a partial link disruption reduced the AB link bandwidth from 1000 Gbps to 500 Gbps, making it unable to handle the AB and ABD tunnels' total bandwidth. The TE control system attempted to redistribute some of the tunnel traffic from the AB link to alternative routes. Yet, it concluded that the spare bandwidth on other links was insufficient for this additional traffic, leading to a continued packet loss of 100 Gbps on the AB link. We lacked detailed runtime logs due to performance considerations. This, along with our inability to replicate the exact fault and traffic conditions, hindered a thorough root cause analysis.

Later with *MirrorNet*, we could recreate the network's state just prior to the failure, and then introduce the specific fault into our simulation. This approach allowed us to scrutinize the network's response in a controlled environment. Through multiple simulations, we discovered that the root cause of the persistent packet loss was not due to a lack of available bandwidth on alternative routes, as initially thought. Instead, it was traced back to a subtle bug in the TE algorithm [1]. This experience highlighted *MirrorNet*'s crucial role in diagnosing and solving intricate problems. With *MirrorNet*, we have fixed tens of issues, significantly enhanced our network's reliability.

Case 2—Slow Parameter Tuning: Similar to [14], our backbone network involves $O(10)$ tunable parameters (e.g., maximum link utilization, load imbalance threshold among

[1] The bug arose from an optimization feature aimed at memory efficiency. Engineers try to optimize the knapsack problem using a one-dimensional array approach rather than a two-dimensional array method to reduce memory. But, this approach is not accurately implemented, resulting in values overlapping during the calculation. As a result, the TE algorithm, despite recognizing the presence of available bandwidth on alternative links, failed to accurately identify and utilize these routes.

equivalent paths) critical for optimizing bandwidth allocation. Conversely, a too low threshold could result in frequent and unnecessary TE rescheduling actions, like switching paths for a mere 1% utilization difference between them. Here, the unnecessary TE action refers to unnecessary traffic rescheduling which fluctuates traffic distribution without much improvement in network performance. Meanwhile, these unnecessary TE actions often need new updates to be programmed into switches, which can lead to about 100ms of packet loss.

Refining these parameters to minimize unnecessary rescheduling and packet loss is challenging. The traditional process involves a series of iterative adjustments and testing. For instance, after changing parameters in version 0 (like scheduling only if link utilization differences exceed 10%), version 1 is tested in portion of the network for validation. Subsequent versions undergo similar cycles, each with different thresholds, like 20% for version 2 and 15% for version 3. This process of controlled deployment and live testing can take months. Each iteration aims to find an optimal balance but also involves risks and delays, making it a slow and challenging task to achieve the ideal parameter settings.

With *MirrorNet*, we have the capability to fine-tune parameters in a controlled emulation environment, ensuring they are optimized before deployment in the production. This system has enabled us to effectively resolve 95% unnecessary controller scheduling actions. Compared to previous methods, *MirrorNet* has substantially reduced the time (months) and resources required for adjusting parameters, marking a significant improvement in our network management efficiency.

Case 3—Shallow Single Plane Testing: Traditional testing methods for multi-plane network architectures (see Figure 1) often rely on the similarities across different planes. This approach typically involves using one plane as a testbed, assuming that successful tests can be replicated across other planes [14]. However, this approach can fail and may lead to incidents when there are differences between planes. Consider a scenario in our network where our objective was to isolate each plane to implement new updates. The process was initially trialed on the first plane, with a plan to replicate it on others if successful. For every plane, the procedure involved sequentially revoking routes for each node within the plane [2]. While this procedure worked well for the first three planes, it resulted in substantial packet loss in the fourth plane, as illustrated in Figure 3. The underlying issue was that, unlike the other router nodes using MPLS-TE to route the traffic, node C in plane-4 lacked MPLS-TE-based traffic forwarding caused by configuration bugs, and thus used the BGP to route traffic. When revoking router C in plane-4, the route update will finally spread to the DC router. At this period, the traffic is erroneously dropped during the route revocation phase.

These intricate plane-specific differences are not uncommon in our production network, stemming from varying traffic, temporary equipment maintenance, and operational variations. On the one hand, although the DC-routers in Figure 1 distribute traffic to different planes through ECMP, the traffic from the same source to the destination on different planes can differ by dozens of times, resulting in different TE policies and traffic distribution in each plane. This is always caused by elephant flow between DCs which will be routed to only one plane and this phenomenon often happens in the production network. On the other hand, temporary maintenance or operational bugs might lead to configuration discrepancies in multiple planes, which will have over days or months. As such, relying on single-plane canary testing fails to uncover these deeper, plane-specific issues, potentially leading to network disruptions [14]. With *MirrorNet*, we have shifted to a more thorough testing approach. By accurately emulating multi-plane conditions, we ensure that any network changes are comprehensively tested for safety and compatibility across all planes before implementation in production.

Case 4—Inaccurate Capacity Assessment: The lack of precise capacity estimation for failure scenarios often leads to critical incidents. These situations typically arise when traffic flow exceeds the available bandwidth during peak hours, especially when unexpected disruptions occur. Accurately predicting the network's ability to handle such fluctuations is crucial, yet it poses a significant challenge.

One of our incidents in 2022 exemplifies this issue. During peak hours, one of our 2200 Gbps bundled link between nodes A and B suddenly dropped to 400 Gbps bandwidth. This led to significant congestion and a 3% packet loss over an hour and a half, as shown in Figure 4. The TE controller couldn't effectively redistribute 1800 Gbps of tunnel traffic to other links due to their limited spare capacity.

While tools like the RSS [46] aim to predict traffic growth for capacity planning, they lack integration of TE algorithms and real configurations. This oversight limits their ability to simulate network behavior accurately. In contrast, our simulation environment, *MirrorNet*, effectively integrates these algorithms, allowing for a more realistic representation of network dynamics. This integration is crucial for understanding the network's resilience to bandwidth fluctuations under real failures and optimize TE responses.

3 *MirrorNet* Design

In this section, we present how we design *MirrorNet* to help with the above common WAN management tasks. We first give an overview of the system architecture (§ 3.1), and then describe each key component in detail (§ 3.2 ∽ § 3.4).

3.1 Architecture Overview

To enhance network reliability, it is crucial to proactively assess the impact of potential changes (as seen in case 2-4) and conduct in-depth retrospective analyses of unexpected

[2]We modify BGP routing policy configurations to achieve this. For instance, to isolate node D's traffic, a temporary rule targeting D (node address 2.2.2.0/24 deny) is added to its outgoing route policy.

Data Category	Concrete Data	Proportion	Frequency of Updates
Node information (Underlay Network)	device ID, device name, IP addresses	<0.1%	Once a year (usually only when new nodes are added)
IP Link Attributes (Underlay Network)	link ID, source device name, destination device name, IP addresses, link bandwidth, etc.	2.3%	Link bandwidth fluctuates averaging O(100) times daily and about 3 times per link.
Tunnel Information (Overlay Network)	tunnel ID, priority, source node, destination node, names, bundle groups, path link IDs, bandwidth, etc.	97.7%	Tunnel paths change up to O(1000) changes daily, averaging 1 changes per tunnel.

Table 1: Network topology changes for a typical week collected from the network event dataset.

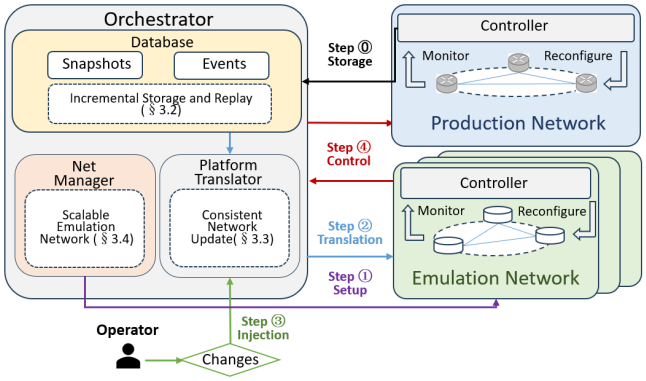


Figure 5: Workflow of *MirrorNet* which interactes with both production network and emulation network.

faults (as in case 1) in § 2.2. We refer to these as forward-planning and backward-analysis capabilities. The production network fundamentally struggles to simultaneously achieve both capabilities while maintaining high performance. This difficulty arises because such tasks invariably affect network stability and can even disrupt ongoing traffic. To address this, we have developed *MirrorNet*, an emulation system designed to replicate the production network environment without impacting its performance. While various network emulation tools exist [7, 9, 12, 16, 23, 33, 41, 46, 48], none fully satisfy our specific requirements:

• *Fine-grained and high-fidelity emulation*: Capable of emulating the production network faithfully at second-level, including detailed replication of configuration, traffic patterns, and control algorithms for any specific historical moment.

• *Scalability*: Capable of emulating large-scale networks (e.g., thousands of nodes and links) and handling diverse complexity levels of tasks efficiently.

• *Resource efficiency*: While emulation necessarily requires additional resources, it's imperative to minimize computation and storage demands and facilitate on-demand emulation.

Achieving all these requirements is a challenging task. Fine-grained emulation demands substantial storage resource, which grows with the network's scale, thus making it difficult to support large-scale networks. The requirement for high-fidelity further amplifies the resource demands. Balancing these conflicting demands requires innovative and sophisticated solutions to navigate the inherent trade-offs.

Workflow: As shown in Figure 5, our *MirrorNet* consists of two key components: a central emulation orchestrator and a set of emulation networks. The orchestrator continuously retrieves real-time network data (e.g., a snapshot of production network, network events, traffic matrix) from the production network and stores it in its database (Step 0). It takes emulation tasks from operators. Based on each task, the orchestrator deploys one or more emulation networks (Step 1), with their lifecycles managed by the Net Manager. Every emulation network, similar to a plane of the production network, comprises: i) a controller managing configurations and executing traffic engineering algorithms. Note that the controller is replicated from the production network to keep the control plane consistent to the production network; ii) a set of routers, emulated by running identical router images in virtual machines loaded with production configurations. The Translator ensures that each emulation network's state aligns with the corresponding segment in the production network (Step 2). During emulation, operators can implement changes using the same interfaces as they would in the production network (Step 3). After completing an emulation task, any changes made and validated within the emulation network can be applied to the production network (Step 4).

In the following subsections, we detail three key techniques of *MirrorNet* to achieve fine-grained and high-fidelity emulation, scalability, and resource efficiency. Firstly, *incremental storage and replay* minimizes storage needs while enabling accurate reconstruction of historical network states (§ 3.2). Secondly, *consistent network updates* ensure emulation networks closely emulate the live network (§ 3.3). Lastly, *scalable emulation network* allow efficiently emulate large-scale networks and complex tasks (§ 3.4).

3.2 Incremental Storage and Replay

To accurately emulate the production network in the *MirrorNet* and capture the network state changes, we should store the network topology including both underlay and overlay information at a fine granularity. However, the network generates massive data volumes — about 16 MB per second, totaling around 1300 GB daily and 40 TB monthly for a single plane. Fine-grained emulation (e.g., second level) introduces a big challenge since it demands substantial storage resources.

Topology data redundancy across time: Our analysis revealed that significant topology changes within the network are infrequent, offering an opportunity to optimize data storage. As shown in Table 1, we observe that node changes are rare with once a year, while most changes happen to tunnel information, with 1 change daily per tunnel on average. All of these changes are not frequent for a daily period. If we store the *changes* instead of the entire data every second, we only

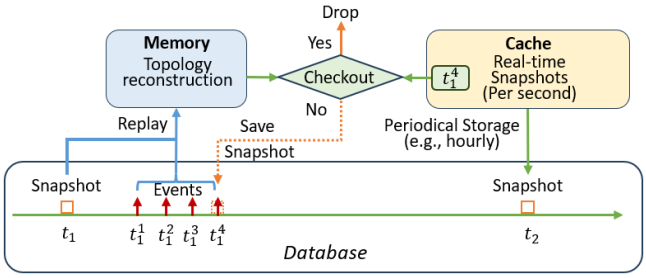


Figure 6: Reconstructing production network through replay.

need to store about 0.003% of the data volume for IP link information, and 0.001% for tunnel information, significantly reducing the storage resource [3].

Incremental storage: Based on this observation, we introduce an incremental storage mechanism to reduce the resource. Specifically, we categorized data based on its frequency of change and stored less frequently altered data as *snapshots* and more dynamic data as *events*. The snapshot information includes network topology consisting of links and nodes and a set of tunnel paths. This data is obtained directly from the TE controller and transformed into formatted data. This network snapshot is periodically stored in the database (every minute, hour, day, or year). These snapshots serve as reference points for the network's state at specific times. Here, the detailed storage frequency of the snapshot is based on the balance between the reconstruction time and the storage demand (see § 5.5). Any changes between two snapshots are recorded as an event and stored in our event database. These events' volume is 10 MB per month, a much smaller in size compared to full-state snapshots.

Reconstruction through replay: To reconstruct the network state at a particular historical moment, the method first identifies the closest preceding snapshot. It then applies the sequence of logged events that occurred after that snapshot up to the desired point in time. As shown in Figure 6, the topology reconstruction module firstly will load the closest preceding snapshot, and then change the snapshot according to the event. We mainly introduce the most common events, including link failures and tunnel path changes. If the event indicates a link failure, then the specific link will be removed in the snapshot. If the path of the tunnel changes, we will replace the previous tunnel based on the event. Note that the event should be added *sequentially* according to their timestamp. This ensures that we can obtain an accurate network state. For example, assume that the path of a tunnel is initially set to A-B-C, and two events are happening with this tunnel, i.e., the path of the tunnel is A-D-C at 1:00 and A-E-C at 1:01 respectively. If the two events are concurrently added to the network, the network may appear in an inaccurate state with A-D-C at 1:01. This process effectively *replays* the network's evolution, leading to an accurate reconstruction of its state at that specific moment.

[3]Here, the link information changes 3 times on average daily. Therefore, we only need to store 3 item instead of 86400 items for a single link over one day. This results in a percentage of ≈0.003% (3/86,400). The same method can be applied to the tunnel information (1/86,400≈0.001%).

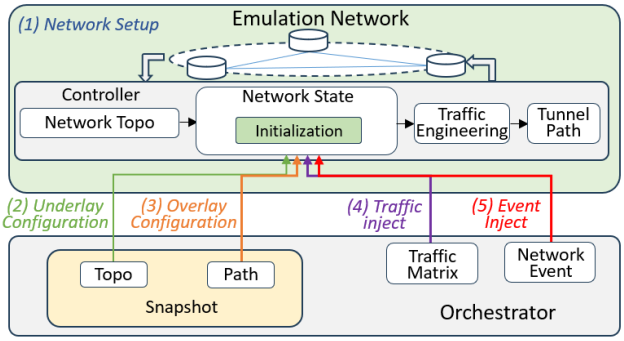


Figure 7: The process of achieving a consistent network update.

However, we observe some events may not be logged due to various issues, such as software bugs, network congestion, etc. To handle changes in the network that are not logged as events, we regularly save snapshots in a cache. We then check these snapshots against the reconstructed network topology to verify its correctness. If they match, we simply remove them from the cache. But if there are differences, we move the cached snapshot to a database as the new reference points for future reconstructions.

By combining less frequent, comprehensive snapshots with a continuous log of smaller, incremental changes, the incremental replay method achieves a balance. It minimizes the storage space needed while still maintaining the level of detail required for precise network state reconstruction.

3.3 Consistent Network Update

As WAN emulation introduces network data from multiple sources, it is challenging to ensure the emulator being consistent to the production network. The emulated TE controller is replicated from the production environment, mirroring the TE decisions performed in the real network by running the same TE algorithm. Yet not all aspects are critical to the fidelity of TE emulation. Factors such as queuing delays do not trigger TE actions; therefore, they are irrelevant in our emulation system. In fact, the order in which the network data are introduced into the emulation plays the most important role in its accuracy and reliability. Directly inputting all network data without considering their dependencies cannot ensure that the controller's scheduling behavior in the emulation environment is consistent with that in the actual network. For example, if traffic data is inputted before the tunnel construction, the traffic paths in the simulation environment can significantly differ from those in the actual network. Similarly, inputting event information before traffic data can result in an ineffective simulation of the events' impact.

Dependency-aware safe update: To develop a systematic and structured approach, we need to understand how each element interacts within the network. This understanding leads us to formulate the following processes in Figure 7.

1. *Topology Construction*: Nodes and links serves as the

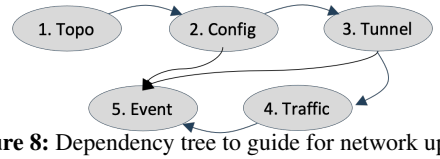


Figure 8: Dependency tree to guide for network update.

most fundamental component of the emulation environment, forming a basic platform for simulation. We use orchestrator APIs to construct it automatically.

2. *Node and Link Attribute Information (underlay network)*: Setting up node and link configurations identical to the production network to establish a faithful underlay infrastructure.
3. *Tunnel Path Configuration (overlay network)*: Mirroring the production network's tunnel paths.
4. *Traffic Information Injection*: Introducing traffic data relevant to the specific period being emulated, typically based on minute-level tunnel granularity;
5. *Event Information Injection*: Event information includes network failure event fingerprints that can influence the controller's behavior. Another type of event information involves network changes that need translation into events understandable by the simulation environment, such as changes in tunnel routing.

To ensure the safe update of the network, these processes have a strong dependency relationship. Figure 8 shows a dependency tree to guide for the network update. Depending on the specific simulation task, the above steps can be sequentially selected. For example, to verify link connectivity in the current simulation environment, steps (1), (2), (3) and (5) would suffice. To assess whether peak traffic would affect controller scheduling, steps (1)-(4) are needed. Failure to adhere to the correct order can result in an inaccurate and ineffective emulation. One particularly tricky step is (3), which requires additional processing to ensure accurate emulation of tunnels.

Tunnel mapping: As shown in Figure 9, in a production network, multiple tunnels are often used for carrying traffic between the same source and destination nodes to distribute load and provide redundancy. For instance, traffic from node A to C might be distributed across three tunnels (Tunnel1∼3) with varying bandwidth allocations, say 100, 500, and 800 Gbps, respectively. In traditional emulation setups, a direct mapping between the production and emulation tunnels isn't always established, leading to inconsistencies in traffic distribution in the simulated environment. For example, traffic that traverses Tunnel1 in production network might be *randomly* assigned to any of emulation tunnels (e.g., Tunnel5, Tunnel6). This randomness in traffic mapping can result in inaccurate traffic scheduling and ineffective simulation outcomes.

To address this issue, a more sophisticated approach is needed to establish a one-to-one mapping between production and emulation tunnels. This involves a thorough analysis of the Traffic Engineering (TE) database in the production network. The process would typically start by identifying each tunnel based on its source node and tunnel ID, then we can

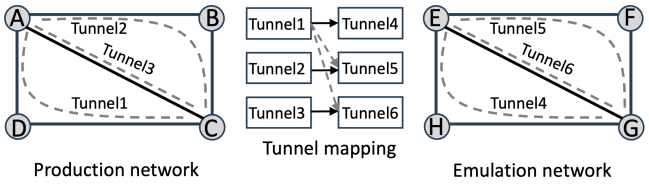


Figure 9: One-to-one tunnel mapping between the production network and emulation network.

get the tunnel's source, destination, link paths, service level (e.g., gold, silver, bronze) [14, 37], and its sequential order (sorted from lowest to highest). Similarly, in the emulation environment, tunnels with matching <source, destination, link paths, service level> criteria are identified and ordered. This leads to the establishment of a mapping between tunnels in both environments. For example, in Figure 9, the tunnel1 in the production network is correctly mapped to tunnel4 in emulation network based on the mapped source and destination nodes and mapped link paths (i.e., A-D-C and E-H-G). This mapping relationship should be established in advance, allowing for a more accurate injection (or replay) of tunnel traffic data and path variations into the emulation environment. This approach allows *MirrorNet* to mirror the production environment's tunnel structure and traffic distribution more precisely.

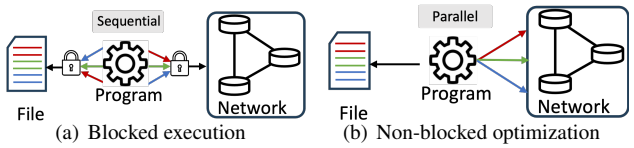
3.4 Scalable Emulation Networks

Our emulation network architecture mirrors that of the production network. In the control plane, we replicate the traffic engineering controller used in production. The data plane primarily comprises virtual devices and links. To accurately simulate the production network, we deploy VM-based virtual devices provided by our vendors[4]. To save resources, we create these VMs and establish intra-server and inter-server links *as needed*. We initially considered two popular open-source platforms for managing VMs and links: OpenStack [5] and GNS3 [1]. However, both platforms are slow in deploying large network emulation environments (§ 5.5).

OpenStack [5], known for its scalable VM management, emulates ports using TAP interfaces and links with Linux bridges or VXLAN tunnels. Designed for emulating end hosts with fewer ports and simpler connections, OpenStack proves inefficient for simulating switches and routers with *many ports* and complex connections. GNS3 [1], an open-source network software emulation platform, shows promise in running small network topologies. However, as network size increases, we notice the time required to create an emulation network grows approximately quadratically with the number of links and nodes. Creating a Kdl topology (see Table 3), for instance, takes about an hour.

Fast emulation deployment: We identify a major challenge

[4]A mixed setup of containers and VMs is also supported by GNS3. When choosing between containers and VMs for network simulation, we decide that containers have limited resource isolation, and it does not support incorporating virtual devices (e.g., Cisco routers, Juniper routers) from vendors to emulate the real network, so we adopt a VM-based approach.



(a) Blocked execution (b) Non-blocked optimization

Figure 10: (a) Blocked execution with sequential operation. (b) Non-blocked optimization with parallel execution.

when integrating vanilla GNS3 into *MirrorNet* to emulate large network scales. That is, when creating the emulation network, traditional platform adopts a blocked execution to create and launch virtual devices and links. Specifically, as shown in Figure 10(a), the traditional platform first creates the nodes in the emulation network. Once a node is created, it will launch the node, register the node information in the network management system, and store the related information in a project file. This process will be repeated until all nodes and links are created and launched. The squential launch of nodes causes the time to grow linearly with node number. Meanwhile, the frequent disk read/write operations are introduced by constant updates to a project file storing all network information. Each node and link creation expanded the file size, greatly increasing disk I/O pressure.

We address this performance decline by enabling *MirrorNet* to run in a non-blocking manner. As shown in Figure 10(b), to achieve a non-blocking operation, we firstly decouple the node creation from the node launch. Rather than starting nodes immediately upon creation, we will maintain a thread pool to create nodes and links asynchronously. Once a node/link is created, we will store the node and link information in the cache. After the last one is created, we store the information into the file. This would only require to read/write disk once and greatly reduce I/O operations. Meanwhile, we initiate all nodes simultaneously to reduce the launch time for these nodes. The creation of nodes and links and store of information in parallel can greatly reduce the time of emulation deployment even if the network is large-scale.

Concurrent emulation with traffic sharing: With more network data introduced into the emulation network, the WAN emulator can perform various tasks. This requires the rapid establishment of multiple emulation networks for parallel simulation. With a single emulation environment, tasks have to be executed *serially*. Assessing a link capacity's tolerance to link failures under current traffic volumes would involve sequentially constructing and testing each failure scenario, injecting the period-specific traffic demands into each emulation. This serial approach is tedious and can take 21 hours for a simple 64-failure scenario on only one link.

To enhance efficiency, instead of the traditional single-plane emulation, we create multiple emulation planes and execute tasks *in parallel*. However, duplicating emulation planes for every task is resource-intensive and can result in incorrect results, due to each emulation environment's unique device models, naming conventions, and link names. A mapping established with one emulation platform cannot be replicated

Algorithm 1: Traffic Translator

```
1 Function TrafficTranslator(T, ProNet, EmuNet):
2     T_e ← ∅
3     foreach t ∈ T do
4         θ ← mapping between ProNet and EmuNet of traffic t
5         t_e ← translate traffic t based on the mapping θ
6     end
7     T_e.add(t_e)
8     return {T_e}
9 End Function
```

or transferred to others due to each emulation environment's unique characteristics and configurations [5]. Therefore, the tunnel traffic collected from the production can not directly adapt to the emulation network. To address this, we introduced an intermediate *traffic translator* to translate the traffic from the production network to adapt to multiple emulation planes simultaneously. Note that to reduce memory footprints, only one copy of the global and full traffic record is stored in the orchestrator. Each plane has a distinct tunnel mapping. For specific emulation tasks running in the emulation planes, the translator is responsible for extracting and translating the related traffic data records. As shown in Algorithm 1, we first initialize the traffic set in the emulation network to $\emptyset$ (lines 2). For each tunnel traffic $t \in T$ in the production network, we find the tunnel mapping between the production network and emulation networks based on Section 3.3, and then assign the volume of traffic to t_e in the emulation network (line 3-5). The traffic information is added to the set T_e (line 7). This translator enables multiple emulation planes to share traffic efficiently while concurrently emulating tasks.

4 Implementation

In this section, we elaborate some important implementation details of *MirrorNet*. *MirrorNet* consists of two main parts, an emulation network that faithfully mocks up the physical network and an orchestrator that interacts with the production network and emulation network. *MirrorNet* consists of $\sim$10k lines of Python and Go code. The emulation network implementation has been discussed in Section 3.4. Since our production network consists of $O(10)$ links and nodes, we use multiple servers with 96 Core Processor and 512 GB memory to run the multi-host GNS3 emulated network, each server simulating $\sim$ 20 devices.

The orchestrator is deployed in the cloud using a 2-core 4GB virtual machine (VM). It contains several APIs to interact with our internal services, such as extracting inter-region traffic demands from the data warehouse, network topology information from redis, etc. For each emulation task, we utilize GoCron [2], a job scheduling platform that enables us to run Go functions. The platform provides on-demand task exe-

[5] For example, the node in the production network named *SF-BR-Route-B1* will have different names with *SF-BR-Route-B1-Simu-01* and *SF-BR-Route-B1-Simu-02* in the two emulation networks, respectively.

Events	Frequency	Examples	Tradition	*MirrorNet*
Software bugs	Low	TE algorithm issues, code block logic issues	NA	minutes
Configuration errors	Low	wrong TE policies, wrong route	days	minutes
Link failures	High	fiber cuts, link jitter	NA/hours	minutes
Traffic issues	Low	traffic burst	Hours	minutes
Device failures	Low	hardware component failures, power down	NA/days	minutes
Complex failures	Low	a mixture of traffic issues, software bugs, link failures, configuration bugs, human errors, etc.	NA	min∼hours

Table 2: The time spent to correlate network events and traffic engineering behavior with/without *MirrorNet*. **NA** means the TE behaviors can not be related to specific network events.

cution or periodic task execution at pre-determined intervals. The on-demand task includes TE behavior localization and network evolution validation, while the periodic task includes capacity assessment and parameter training. The data such as traffic demands, network events, etc., used for emulation is stored as different topics with detailed timestamps in the Apache Kafka [3]. The orchestrator reads from these topics according to the timestamp to execute tasks.

5 Evaluation

MirrorNet has been running in TWAN 's large-scale WAN for two years. In this section, we investigate how the proposed system enables associating controller TE behaviors with network events in a timely manner, especially for these previously untraceable behaviors (§ 5.1). We then use a typical scenario to show how the proposed system greatly reduces the converge time of setting these controller parameters to the optimality (§ 5.2). We demonstrate how *MirrorNet* helps evaluate network tolerance to link failures (§ 5.3) and how the digital twin network reduces the rate of failures during network infrastructure evolution (§ 5.4). Finally, we present the performance and scalability of *MirrorNet* (§ 5.5).

5.1 Localizing TE Behaviors

We firstly study whether each TE decision by the controller is properly made and network traffic is accurately dispatched. Some of TE decisions such as traffic shaping [18] are not disruptive and planned by network operators. We focus on more disruptive and unpredictable traffic engineering behaviors. Traditional systems only use static network topologies (e.g., a snapshot of the network) to validate network configurations and routes. They are not able to perform a thorough root cause analysis due to their inability to replicate the exact fault and traffic conditions. However, *MirrorNet* can recreate the network's state just prior to the failure and enables to correlate the TE behaviors and network events, and thus identify if the existing behavior is expected.

Table 2 depicts the time of correlating TE behaviors with network events with/without *MirrorNet*. Firstly, we observe that most of the TE behaviors are introduced by link failures. Traditional approaches take experienced network operators hours to correlate TE actions with a fiber cut event. However, they are not able to correlate part of the TE actions caused by the lost of part of the link capacity, i.e., link jitters. In contrast,

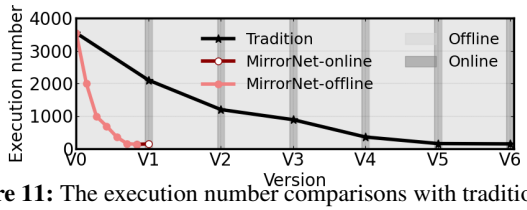


Figure 11: The execution number comparisons with traditional approaches and *MirrorNet*.

MirrorNet only takes minutes to correlate all the events related to the links. Secondly, a small part of TE actions are caused by complex failures such as combinations of link failures and software bugs, link failures, and traffic issues. Traditional systems are hard to replay these TE actions because they can not replicate the exact fault and traffic conditions. However, *MirrorNet* can put these events together to generate a network state and then only takes minutes to replay the TE actions. Thirdly, our proposed *MirrorNet* correlates TE actions with network events with minutes except for these complex failures taking hours, which is orders of magnitude faster than the existing systems. With *MirrorNet*, we have fixed tens of issues, significantly enhanced our network's reliability.

5.2 Parameter Training

There are various traffic engineering (TE) algorithms to address complex network environments [14] and $O(10)$ TE parameters are configured to adapt to network environment changes. For example, each link is set up with a maximized link utilization α to avoid the link being overloaded. Once the link utilization exceeds α, TE is performed to dispatch network traffic on the overloaded link to other links. However, existing TE parameter settings are configured based on network operators' experiences. As the network changes dynamically, we argue existing parameter settings are not always optimal. Therefore, network operators need to gradually adjust these parameters to adapt to network dynamics. However, updating these parameters directly on the production network is risky since it may introduce unexpected network failures. Here, we use two typical cases to show the superiority of our *MirrorNet* to obtain the optimal TE parameters. Our *MirrorNet* can be extended to train all parameters in the network.

Case1 - Reducing unnecessary TE behaviors. Our measurement in production shows that there are thousands of TE executions to reschedule network traffic every day. These reschedulings often need new updates to be programmed into switches, which can lead to about 100 milliseconds of packet loss. However, most of the actions are unnecessary due to

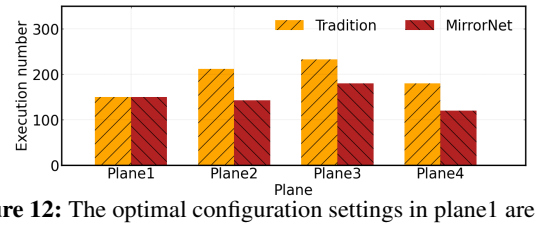
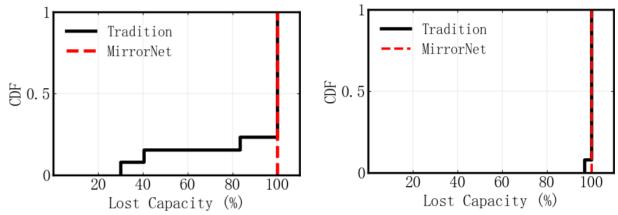


Figure 12: The optimal configuration settings in plane1 are not optimal in other planes.



(a) Rush hours (20:00∼22:00) (b) Ordinary hours (8:00∼10:00)

Figure 13: The percentage of maximum capacity loss that can guarantee zero flow loss.

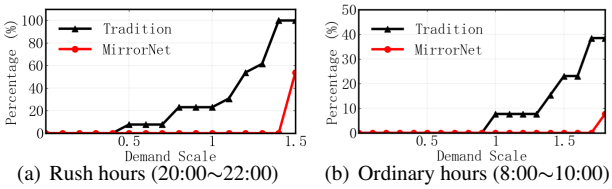
unreasonable parameter settings (see case 2 in § 2.2).

Our goal is to reduce unnecessary TE executions by optimizing parameter settings. Figure 11 shows daily TE actions with *MirrorNet* and traditional approaches. The y-axis indicates the number of TE actions; "offline" and "online" refer to validation in the emulated and production networks, respectively. Remarkably, 95% of TE actions are unnecessary compared to optimal settings. Traditional methods iteratively tune parameters directly in production, requiring six iterations to converge, increasing risks and deployment delays. In contrast, *MirrorNet* trains parameters offline with our high-fidelity network emulation without impacting on production traffic. Thus, optimal settings are reached with just one production update, minimizing both risk and deployment time.

Case2 - Parameter fine-tuning among different planes. As multi-plane architecture has been widely adopted in the WAN [14], traditional approaches always configure each plane with the same parameter settings. However, network characteristics such as TE algorithms and traffic patterns are slightly different among these planes, indicating that the optimal parameters in plane1 are not optimal in other planes. Taking the TE execution number as an example shown in Figure 12, plane1 is configured with the optimal parameter settings with the minimal TE execution number. However, these parameter settings are not optimal in other planes as the TE execution number is always larger. In contrast, *MirrorNet* establishes a faithful network emulation for each plane and then fine-tunes parameters for each plane separately. *MirrorNet* can find the optimal parameter settings for each plane although their network characteristics are different.

5.3 Capacity Assessment

Our measurement results show that part of link capacity loss will lead to persistent traffic loss, reducing network availability (§ 2.2). This observation indicates our production network



(a) Rush hours (20:00∼22:00) (b) Ordinary hours (8:00∼10:00)

Figure 14: Percentage of unsupported link failure vs. demand scales for traditional approach and *MirrorNet*.

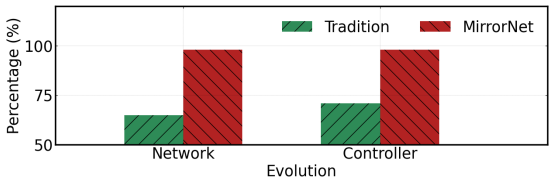


Figure 15: The successful rate when evolving network.

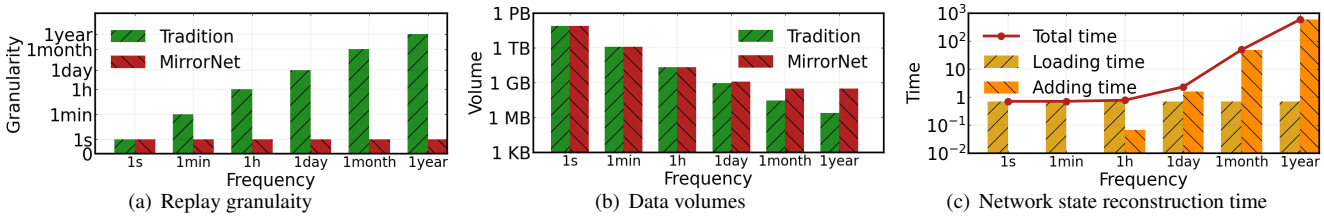
does not tolerate network failures (i.e., guaranteeing no flow loss under one IP link failure), despite various traffic engineering algorithms are deployed to address failures.

We then dig deeper to quantify how much link capacity can be lost without causing flow loss. As shown in Figure 13, we observe that traditional network capacity planning will result in traffic loss for 25% of links whose link capacity is lost, and for some links, even a small part of capacity loss (e.g., 30%) will finally lead to the traffic loss. For these ordinary hours without heavy network traffic, traditional network capacity planning still does not guarantee zero flow loss under arbitrary one-link failure scenarios. This phenomenon strongly indicates that the capacity of each link is not properly planned in a balanced utilization behavior or does not adapt quickly to changed traffic patterns. We then leverage *MirrorNet* to obtain the optimal capacity settings for each link by changing each link's capacity based on its utilization and then validate the new capacity settings. We keep the same total capacity in the whole network for fair comparisons. We observe that *MirrorNet* enables the network to reach the optimal by balancing link capacity that guarantees zero flow loss under one link failure scenario.

We then study the traffic demand scaling that influences the number of link failure scenarios. As shown in Figure 14, an important observation is that link capacity settings by *MirrorNet* can support the traffic demand scale up to 1.4× that guarantees zero flow loss under arbitrary single link failure in rush hours, while traditional link capacity settings only support the scale up to 0.4 ×. In ordinary hours, *MirrorNet* can support the traffic demand scale up to 1.7×, while traditional settings only support the scale up to 0.9 ×. *MirrorNet* can support 3.5 × and 1.9 × in the rush and ordinary hours respectively over traditional settings.

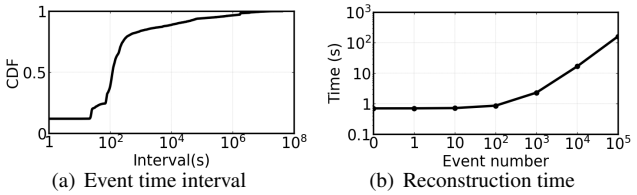
5.4 Network Evolution

Network evolution frequently happens to adapt to dynamic traffic patterns. We then study how *MirrorNet* helps improve the network evolution. We mainly focus on key factors that



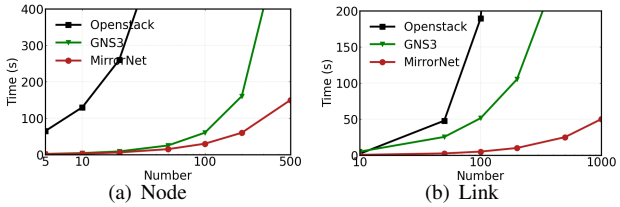
(a) Replay granularity (b) Data volumes (c) Network state reconstruction time

Figure 16: (a) Replay granularity with different snapshot frequencies; (b) Total data volumes for a year with different snapshot frequencies; (c) Reconstruction time with different snapshot frequencies.



(a) Event time interval (b) Reconstruction time

Figure 17: (a) Time interval between two events. (b) Time to reconstruct a specific state with different event numbers.



(a) Node (b) Link

Figure 18: The time to construct node and link.

are closely related to traffic engineering, including network plane (e.g., adding a link, removing a link, bundling to equal cost paths into one path) and control plane (e.g., TE parameters, new TE functionalities). Figure 15 shows the successful rate of evolving the network. A new version of the network will be gray-box testing before it is deployed in production. We consider the version successful if it passes the gray-box testing. We observe that traditional approaches based on network operators' experiences show an unsatisfying successful rate of network infrastructure evolution. The main reason is that the bugs ignored by network operators, however can be discovered in the gray-box testing. However, we utilize *MirrorNet* to evolve the network and achieves nearly 100% successful network evolutions. This is attributed to our ability to validate new version fo the network in the emulation network. Meanwhile, we can inject network failures that have happened before to learn the robustness of new networks to address these failures.

5.5 Performance & Scalability

Snapshot frequency. Network state is closely related to the underlay network (i.e., node and link information) and overlay network (i.e., tunnel information). The network state at a replying time rely heavily on the network snapshot and the events between the snapshot time and the replying time. We firstly study the time interval between two consecutive events. Here, the events refer to the changes in network topology and tunnel path. As shown in Figure 17(a), 50% of events happen within 100 seconds after the previous event, and even 10% of events happen within 1 second. The phenomenon indicates frequent events happening in the network.

We then study the time to reconstruct a specific network state with different numbers of events in Figure 17(b). We observe that the reconstruction time is about 0.7 seconds if the event number is less than 100. Most of the time is spent loading the snapshot of network topology and tunnel paths. Another observation is that the reconstruction time is approximately linear to the event number when the event number is larger than 100. The reconstruction time is more than 10 seconds when the event number is 10,000. This linear relationship is attributable to the choice of serializing the adding operations of each event to the network state to obtain an accurate network state (see § 3.2).

We study the impact of network snapshots' storage frequency. Figure 16(a) shows the replay granularity with the traditional approaches. Traditional approaches only rely on snapshots to reconstruct the network state. We observe that the replay granularity of the network state by traditional approaches relies on snapshot storage frequency. If the snapshot is stored hourly, the replay granularity is one hour and the network state within one hour cannot be reconstructed. In contrast, *MirrorNet* utilizes a combination of snapshots and events to reconstruct network state. The replay granularity is 1 second, regardless of snapshot stored frequency. To perform an accurate replay of the network state with the traditional network, an effective way is to increase the frequency of storing the snapshot. However, we observe that the data volume is hundreds of TBs for a year if we store snapshot at every second in Figure 16(b). Such a large amount of data volumes will consume huge resources and is not cost-effective. So there is a trade-off between cost-efficiency and replay granularity with the traditional approach. However, *MirrorNet* can largely reduce the data volumes without loss of replay granularity.

MirrorNet will bring a little overhead such as reconstruction time, as *MirrorNet* utilizes a combination of the snapshot and event for reconstruction. Figure 16(c) shows the network state reconstruction time consisting of loading the snapshot (i.e., loading time) and adding the events (i.e., adding time). We observe that the reconstruction time is close to the loading time if the snapshot frequency is less than an hour. In production, *MirrorNet* stores the snapshot hourly, indicating that it

	(nodes,links)	OpenStack	GNS3	*MirrorNet*
B4	(12, 38)	183s	24s	5s
IBM	(23, 85)	436s	53s	11s
TWAN	(*O(10)*, *O(50)*)	484s	45s	12s
UsCarrier	(158, 378)	4740s	320s	66s
Kdl	(754, 1790)	69912s	2735s	315s

Table 3: The time to construct different network topologies

consumes acceptable network resources for storage and does not bring much reconstruction overhead while maintaining one-second replay granularity. Our approach can be further extended to other production networks to guide network operators in configuring snapshot intervals for different networks by identifying the trade-off between the reconstruction time and network storage.

Creating emulation environment. We further study the time to construct the network topology. Here, we use two baselines for comparison: (1) *OpenStack.* an open standard cloud computing platform to provide virtual servers. (2) *GNS3.* An open-source network simulator to provide network components to design and configure virtual networks. *MirrorNet* extends open-sourced GNS3 and improves the node and link creation process to reduce the time to construct the topology. Figure 18(a) shows the time of different numbers of node construction. We observe that for OpenStack, it takes more than 100 seconds to construct 10 nodes in the emulated environment. As the number of ports in a switch is much more than that in the server, the majority of time spent by OpenStack is to create virtual ports. *MirrorNet* performs better than GNS3 because we create nodes and links asynchronously and only update the project file after the final node and link are in place to reduce the time. We only reuqire 150 seconds to construct 500 nodes, which is under 25% of the time when compared to GNS3. The performance gains are larger if the number of nodes is larger. The trend is similar to the link construction in Figure 18(b).

We further study our proposed *MirrorNet* on constructing different WAN topologies. We consider five representative WAN topologies with different sizes and their numbers of nodes and edges and the time to construct network topology are summarized in Table 3. We observe that OpenStack spends most time constructing the network topology. It takes 183 seconds to construct the B4 topology. GNS3 requires 24 seconds to construct network topology. In contrast, *MirrorNet* only spends 5 seconds, which is 36.6× and 4.8× faster than Openstak and GNS3 respectively. For the large topology such as Kdl, *MirrorNet* only takes about five minutes to construct that topology which is 221.9× and 8.7× faster than Openstack and GNS3 respectively, showing the superiority of *MirrorNet* to construct larger network topology.

6 Operational Experience

We have learned important operational lessons throughout years of running *MirrorNet* in production.

In TE emulation, fidelity comes from replaying traffic patterns in the TE algorithm for decision-making, not reproducing the raw network traffic in the testbed. In our process of running MirrorNet, we empirically discovered a fundamental constraint in the simulation environment: the inability of VM-based vendor devices to handle traffic at scales comparable to production networks. To be more specific, the measurement of backbone traffic demands in each tunnel ranges from tens of Kbps to hundreds of Gbps, far exceeding the capacity supported by emulation network (e.g., Mbps). Therefore, we only configure both links and nodes in the emulation network to maintain the network topology and utilize the traffic value from the database collected from the production network to perform the TE decisions. This simplified emulation does not compromise the fidelity of MirrorNet as our focus lies in replicating the TE behaviors of the production grade software-defined WAN. Impacts on network states by the network behaviors like microbursts and queuing delays are recorded and can be reconstructed through events, which preserves the fidelity of MirrorNet.

TE emulation duration can be shortened by speeding up traffic replay, but the maximum safe speedup is limited by TE computation and reconfiguration delays between consecutive events The running time of the emulation network is the same as the time interval we require in the production network. For example, if we want to replay the behaviors of the TE controller from 12:00 to 18:00, we need to run the emulation network for 6 hours, inputting traffic demands every 5 minutes (i.e., a TE period [10]). The emulation will take a long time to emulate a large time interval. We accelerate the emulation process by speeding up the inputting rates of traffic demands. For example, we input the traffic demands collected every 5 minutes from the production network every 30 seconds in the emulation network. In this way, the emulation speedup ratio is 10. Note that the speed-up ratio is not infinite, which varies according to factors such as TE computation time, path reconfiguration time in the router, and time interval between two successive events. For example, for two consecutive events, the speedup ratio can be maximized to the consecutive events' time interval / time of TE computation + path reconfiguration. Otherwise, if TE computation and path reconfiguration are not completed once the previous event occurs, the subsequent event will cause the controller to fail to respond in time and lead to the replay error.

Root-cause diagnosis with serial testing is too slow to meet operational demands, necessitating the need for parallel emulation. Network failures often trigger traffic reschedule in the TE control system, but identifying the root causes of these events is challenging due to the vast issue space. To accelerate diagnosis, we establish multiple emulation networks in parallel, each dedicated to a specific failure point. For example, in case 1 of § 2.2, unexpected TE behavior could stem from routing errors, configuration bugs, software faults, or TE algorithm issues. We apply the control variable method

to check each potential failure point. Traditionally, building one emulation network and testing failure points sequentially is time-consuming. Instead, we create multiple emulation networks concurrently and enable traffic-sharing (§ 3.4) for parallel emulation, significantly reducing the time to localize failures. To further streamline deployment, we developed a script using GNS3 APIs that can batch-generate emulation networks. Operators only need to provide a file specifying information of nodes and links; the script automatically constructs the corresponding emulation environments.

Emulation scopes and limitations: While *MirrorNet* faithfully emulates WAN devices using vendor-provided GNS3 virtual devices, their black-box nature imposes limitations on emulating internal failures like OS or ASIC bugs which does not trigger new TE actions [6]. However, any hardware failures that result in TE actions will be captured by MirrorNet, which will be identified in backward analysis.

7 Related Work

Traffic engineering. Centralized traffic engineering has been studied intensively in WANs to allocate traffic between data centers [6, 10, 24, 25, 29, 32, 39, 44, 47, 51]. These works formulate the problem of allocating traffic as an optimization problem and globally optimize flow allocations for desirable network performance. Recent work [14] further evolves TE algorithms to adapt to network scenarios. As these works focused on TE algorithms, *MirrorNet* is orthogonal to them. *MirrorNet* focuses on localizing traffic engineering actions with network events (e.g., software bugs, link failures, complex failures) so that we can study whether each TE decision is properly made and network traffic is accurately dispatched.

Network validation. Network verification systems [9, 12, 23, 41, 48] simulate the routing protocols to compute forwarding tables and answer reachability questions. A recent work [31] further verifies the traffic load properties. However, they could not verify software bugs, vendor protocol implementation differences, etc. Network simulators [7, 16, 46] simplify components such as network devices and controllers, and calculate their status as network events occur. However, they assume ideal behaviors and present a very different workflow from the operations of production networks. Network emulation systems such as CrystalNet [33], Kollaps [21] and Crescent [17] are closer to our work. CrystalNet [33] uses vendor's software switches loaded with production configurations to mock up physical networks. Since it is tailored for data center networks and only captures a snapshot of the network, it falls short in emulating the network state changes (e.g., traffic changes). This partial view can lead to potential discrepancies between simulated outcomes and actual network behavior, resulting in unforeseen issues after deployment. Crescent [17] is a WAN emulator that aims at evaluating networks under arbitrary changes. Kollaps [21] is a distributed network topology simulator. They do not consider network dynamics introduced by TE or real traffic data. Sibyl [11] proposes a black box testing framework for fat-tree topologies, targeting routing protocol implementations under various failure and recovery scenarios, typically in DCN scenarios. In contrast, *MirrorNet* is a mirror of the production network, which not only supports validating network configurations but also accurately captures network state changes during the emulation. *MirrorNet* integrates GNS3 [1] to emulate the underlay network. Several other network device emulators use container-based approaches, offering faster boot-up times and lower resource consumption. However, we chose GNS3 over alternatives because it provides full OS emulation through virtual machines (VMs), ensuring high fidelity and broad compatibility with various network devices. Mininet [4] uses threads to virtualize network devices, but does not emulate the full network OS and packet processing behavior. Kathará [42] relies on Docker containers that provide less OS isolation compared to VMs. Sharing the underlying kernel could have unexpected impacts on the emulation fidelity or performance, so we eventually opted for GNS3 for managing VMs.

WAN failures. There are a lot of works focusing on real-world measurements to understand WAN failures. Some works [19, 20, 36, 40, 43] focus on optical layer failures, while others focus on IP layer failures [8, 13, 15, 28, 30, 33, 35, 38, 49]. Our failure analysis confirms the observation in previous papers that a good proportion of failures are human-related configuration bugs [22, 33, 35]. Furthermore, *MirrorNet* observes some of failures such as control plane software bugs that closely related to traffic engineering.

8 Conclusion

This paper presents our experience in designing *MirrorNet*, a framework that fully emulates our production WAN. *MirrorNet* has become an indispensable tool in our WAN operations, aiding in troubleshooting, parameter tuning, testing, and capacity assessment, and significantly contributing to the network's reliability and efficiency. We describe our end-to-end design that manages the trade-offs between fine-grained, high-fidelity emulation, scalability, and resource efficiency. We hope to bring a fresh angle to WAN management research, paving the way for future innovations in the field.

Acknowledgments

We sincerely thank our shepherd Wei Bai and all NSDI anonymous reviewers for their constructive comments and suggestions. This work is supported by National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), and Peking University startup fund. Guyue Liu and Congcong Miao are corresponding authors.

[6] With a small number (i.e., O(10)) of routers, there are almost no ASIC chip bugs in production WAN.

References

[1] Gns3. https://www.gns3.com/.

[2] Gocron. https://pkg.go.dev/github.com/go-co-op/gocron.

[3] Kafka. https://kafka.apache.org/.

[4] Mininet: An instant virtual network on your laptop (or other pc). https://mininet.org/.

[5] Openstack. https://www.openstack.org/.

[6] F. Abuzaid, S. Kandula, B. Arzani, I. Menache, M. Zaharia, and P. Bailis. Contracting wide-area network topologies to solve flow problems quickly. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pages 175–200, 2021.

[7] A. Agrawal, V. Bhatia, and S. Prakash. Network and risk modeling for disaster survivability analysis of backbone optical communication networks. *Journal of Lightwave Technology*, 37(10):2352–2362, 2019.

[8] A. Alaraj, K. Bock, D. Levin, and E. Wustrow. A global measurement of routing loops on the internet. In *International Conference on Passive and Active Network Measurement*, pages 373–399. Springer, 2023.

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

[10] J. Bogle, N. Bhatia, M. Ghobadi, I. Menache, N. Bjørner, A. Valadarsky, and M. Schapira. Teavar: striking the right utilization-availability balance in wan traffic engineering. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 29–43. 2019.

[11] T. Caiazzi, M. Scazzariello, L. Alberro, L. Ariemma, A. Castro, E. Grampin, and G. Di Battista. Sibyl: a framework for evaluating the implementation of routing protocols in fat-trees. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7. IEEE, 2022.

[12] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A nice way to test openflow applications. In *9th USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[13] A. Dainotti, C. Squarcella, E. Aben, K. C. Claffy, M. Chiesa, M. Russo, and A. Pescapé. Analysis of country-wide internet outages caused by censorship. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 1–18, 2011.

[14] M. Denis, Y. Yao, A. Hatch, Q. Zhang, C. L. Lim, S. Zhang, K. Sugrue, H. Kwok, M. J. Fernandez, P. Lapukhov, et al. Ebb: Reliable and evolvable express backbone network in meta. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 346–359, 2023.

[15] N. Feamster, D. G. Andersen, H. Balakrishnan, and M. F. Kaashoek. Measuring the effects of internet path faults on reactive routing. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):126–137, 2003.

[16] K. Gao, L. Chen, D. Li, V. Liu, X. Wang, R. Zhang, and L. Lu. Dons: Fast and affordable discrete event network simulation with automatic parallelization. In *Proceedings of the ACM SIGCOMM*, page 167–181, 2023.

[17] Z. Gao, A. Abhashkumar, Z. Sun, W. Jiang, and Y. Wang. Crescent: Emulating heterogeneous production network at scale. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1045–1062, Santa Clara, CA, Apr. 2024. USENIX Association.

[18] L. Georgiadis, R. Guérin, V. Peris, and K. N. Sivarajan. Efficient network qos provisioning based on per node traffic shaping. *IEEE/ACM transactions on networking*, 4(4):482–501, 1996.

[19] M. Ghobadi, J. Gaudette, R. Mahajan, A. Phanishayee, B. Klinkers, and D. Kilper. Evaluation of elastic modulation gains in microsoft's optical backbone in north america. In *2016 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3. IEEE, 2016.

[20] M. Ghobadi and R. Mahajan. Optical layer failures in a large backbone. In *Proceedings of the 2016 Internet Measurement Conference*, pages 461–467, 2016.

[21] P. Gouveia, J. Neves, C. Segarra, L. Liechti, S. Issa, V. Schiavoni, and M. Matos. Kollaps: decentralized and dynamic topology emulation. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[22] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 58–72, 2016.

[23] D. Guo, S. Chen, K. Gao, Q. Xiang, Y. Zhang, and Y. R. Yang. Flash: fast, consistent data plane verification for large-scale network settings. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 314–335, 2022.

[24] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pages 15–26, 2013.

[25] C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, C. Bhagat, S. Jain, J. Kaimal, S. Liang, K. Mendelev, et al. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 74–87, 2018.

[26] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[27] X. Jia, Z. Yao, C. Peng, Z. Zhao, B. Lei, E. Liu, X. Li, Z. He, Y. Wang, X. Zou, et al. Turbo: Efficient communication framework for large-scale data processing cluster. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 540–553, 2024.

[28] E. Katz-Bassett, C. Scott, D. R. Choffnes, Í. Cunha, V. Valancius, N. Feamster, H. V. Madhyastha, T. Anderson, and A. Krishnamurthy. Lifeguard: Practical repair of persistent route failures. *ACM SIGCOMM Computer Communication Review*, 42(4):395–406, 2012.

[29] P. Kumar, Y. Yuan, C. Yu, N. Foster, R. Kleinberg, P. Lapukhov, C. L. Lim, and R. Soulé. Semi-oblivious traffic engineering: The road not taken. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 157–170, 2018.

[30] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental study of internet stability and backbone failures. In *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*, pages 278–285. IEEE, 1999.

[31] R. Li, Y. Yuan, F. Ye, M. Liu, R. Yang, Y. Yu, T. Guo, Q. Ma, X. Zeng, C. Xu, et al. A general and efficient approach to verifying traffic load properties under arbitrary k failures. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 228–243, 2024.

[32] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter. Traffic engineering with forward fault correction. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 527–538, 2014.

[33] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 599–613, 2017.

[34] Y. Liu, Y. Xiao, X. Zhang, W. Dang, H. Liu, X. Li, Z. He, J. Wang, A. Kuzmanovic, A. Chen, et al. Unlocking ecmp programmability for precise traffic control. 2025.

[35] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, and C. Diot. Characterization of failures in an ip backbone. In *IEEE INFOCOM 2004*, volume 4, pages 2307–2317. IEEE, 2004.

[36] C. Miao, M. Chen, A. Gupta, Z. Meng, L. Ye, J. Xiao, J. Chen, Z. He, X. Luo, J. Wang, et al. Detecting ephemeral optical events with {OpTel}. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 339–353, 2022.

[37] C. Miao, Z. Zhong, Y. Xiao, F. Yang, S. Zhang, Y. Jiang, Z. Bai, C. Lu, J. Geng, Z. He, et al. Megate: Extending wan traffic engineering to millions of endpoints in virtualized cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 103–116, 2024.

[38] R. Padmanabhan, A. Schulman, A. Dainotti, D. Levin, and N. Spring. How to find correlated internet failures. In *Passive and Active Measurement: 20th International Conference, PAM 2019, Puerto Varas, Chile, March 27–29, 2019, Proceedings 20*, pages 210–227. Springer, 2019.

[39] Y. Perry, F. V. Frujeri, C. Hoch, S. Kandula, I. Menache, M. Schapira, and A. Tamar. {DOTE}: Rethinking (predictive){WAN} traffic engineering. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1557–1581, 2023.

[40] R. Potharaju and N. Jain. When the network crumbles: An empirical study of cloud network failures and their impact on services. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–17, 2013.

[41] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 953–967, 2020.

[42] M. Scazzariello, L. Ariemma, and T. Caiazzi. Kathará: A lightweight network emulation system. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–2. IEEE, 2020.

[43] R. Singh, M. Ghobadi, K.-T. Foerster, M. Filer, and P. Gill. Radwan: rate adaptive wide area network. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 547–560, 2018.

[44] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford. Network architecture for joint failure recovery and traffic engineering. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):97–108, 2011.

[45] D. Wetherall, A. Kabbani, V. Jacobson, J. Winget, Y. Cheng, C. B. Morrey III, U. Moravapalle, P. Gill, S. Knight, and A. Vahdat. Improving network availability with protective reroute. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 684–695, 2023.

[46] Y. Xia, Y. Zhang, Z. Zhong, G. Yan, C. L. Lim, S. S. Ahuja, S. Bali, A. Nikolaidis, K. Ghobadi, and M. Ghobadi. A social network under social distancing:{Risk-Driven} backbone management during {COVID-19} and beyond. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 217–231, 2021.

[47] Z. Xu, F. Y. Yan, R. Singh, J. T. Chiu, A. M. Rush, and M. Yu. Teal: Learning-accelerated optimization of wan traffic engineering. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 378–393, 2023.

[48] F. Ye, D. Yu, E. Zhai, H. H. Liu, B. Tian, Q. Ye, C. Wang, X. Wu, T. Guo, C. Jin, et al. 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*, pages 599–614, 2020.

[49] M. Zhang, C. Zhang, V. S. Pai, L. L. Peterson, and R. Y. Wang. Planetseer: Internet path failure monitoring and characterization in wide-area services. In *OSDI*, volume 4, pages 12–12, 2004.

[50] Z. Zhang, H. Zheng, J. Hu, X. Yu, C. Qi, X. Shi, and G. Wang. Hashing linearity enables relative path control in data centers. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 855–862, 2021.

[51] Z. Zhong, M. Ghobadi, A. Khaddaj, J. Leach, Y. Xia, and Y. Zhang. Arrow: restoration-aware traffic engineering. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 560–579, 2021.