Risk based Planning of Network Changes in Evolving Data Centers

Omid Alipourfard
Yale University

Jiaqi Gao
Harvard University

Jeremie Koenig
Yale University

Chris Harshaw
Yale University

Amin Vahdat
Google

Minlan Yu
Harvard University

Abstract

Data center networks evolve as they serve customer traffic. When applying network changes, operators risk impacting customer traffic because the network operates at reduced capacity and is more vulnerable to failures and traffic variations. The impact on customer traffic ultimately translates to operator cost (e.g., refunds to customers). However, planning a network change while minimizing the risks is challenging as we need to adapt to a variety of traffic dynamics and cost functions while scaling to large networks and large changes. Today, operators often use plans that maximize the residual capacity (MRC), which often incurs a high cost under different traffic dynamics. Instead, we propose Janus, which searches the large planning space by leveraging the high degree of symmetry in data center networks. Our evaluation on large Clos networks and Facebook traffic traces shows that Janus generates plans in real-time only needing 33~71% of the cost of MRC planners while adapting to a variety of settings.

CCS Concepts • **Networks → Network management**; *Network reliability*; Network simulations.

Keywords network change planning, network compression, network simulations

ACM Reference Format:
Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. 2019. Risk based Planning of Network Changes in Evolving Data Centers. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3341301.3359664

1 Introduction

Data center networks are evolving fast to keep up with traffic doubling every year [37, 40] and frequent rollouts of new applications. They continuously change both hardware and software to scale out and add new features. These changes include *repairs* such as firmware security patches and *upgrades* such as addition of new features to switch hardware or software. Such changes are even more common in recent years with the adoption of software-defined networking [20, 23, 26] and programmable switches [3, 12, 24].

Changes come with an inherent risk of impacting customers and their traffic: operators have to apply network changes while upholding high availability and good performance—draining the entire data center before applying changes is too costly (typically measured through SLAs). When a change is taking place, the network operates at reduced capacity and has less headroom for handling traffic variations and failures [17, 18]. Google reports that 68% of failures occur during the network changes [17]. There are also other risks due to delayed changes and bugs in the change itself (§2.1).

A risk is the likelihood of any event impacting customer traffic. These events result in a violation of service-level objectives (SLOs) and hurt operator income. For example, Amazon refunds 30% of credits to customers experiencing less than 90% uptime. Thus, reducing risk is critical for all operators, but it also requires investment and is not cheap. Operators can reduce risk by overprovisioning the network [18]: with enough capacity, the network has headroom to absorb traffic variations and failures during network changes seamlessly, but this comes at a high CAPEX and OPEX cost.

There is a fundamental tradeoff between risk tolerance and cost: operators can choose to pay more, upfront, by overprovisioning to keep network utilization and risk low; or run the network at high utilization and accept a moderate risk of impacting customer traffic. Each data center operator can choose its operating point based on their budget for network resources and penalties associated with SLO violations.

Given an operating point (i.e., the level of capacity overprovisioning), operators have to make decisions on when and how to apply changes in a way that minimizes the *expected cost* of risks. However, planning a network change is challenging because it has to meet two goals:

Adaptivity: The best change plan depends on (1) *Temporal and spatial traffic dynamics* influence the expected risk of a plan. A safe plan *now* could be unsafe one hour later when traffic volumes are high (temporal dynamics). Similarly, whether we can apply a change to a core switch depends on the intra-pod and inter-pod traffic (spatial dynamics). (2) *Cost functions* which are the penalties operators incur when

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359664

customers' traffic is affected in the network. The penalty depends on the customer/cloud agreements, and it is often defined based on the type of service [4, 5] (see examples in §2.1). (3) *Other factors* also need to be taken into account, e.g., failures, the topology, and routing (see §2.2).

Today, most operators use capacity-based planning. For example, Google [17, 40] divides the switches involved in a change into equal-sized sets and applies the change sequentially to maximize the residual capacity during the change. This approach is simple and scalable, but it does not adapt to traffic dynamics or failures, and it often results in higher penalties such as increased cost (see §2.2). Therefore, new solutions are needed that can adapt to such changes.

Scalability: Finding a plan for a change is not easy: the space of possible plans is super-exponentially large (§2.1). For example, there are 3.4×10^{1213} plans for upgrading 500 switches. Brute force search of the entire space is not scalable. We often need to plan changes in real-time (as plans become obsolete after long durations due to changes traffic variations and failures) and therefore, there is a need for a system that can search the space of possible plans efficiently to find the best possible plan. We build Janus to do exactly that.

Janus is a change planner that leverages the inherent symmetry of data center networks to search for the best plan in a large planning space. Janus has the following key ideas:

Find blocks of equivalent switches: Given topology and routing, Janus identifies blocks of switches that connect to the same set of other switches (i.e., switches in one block are interchangeable). Within a block, we do not need to decide which individual switches to change at any given time, but rather how many switches to change (§3.1).

Find equivalent subplans: Some subplans include switches in different blocks but have the same impact on customer traffic under all traffic settings (§3.2). We leverage graph automorphism to identify these equivalent subplans.

Scale cost estimation: We run flow-level Monte-Carlo simulations to estimate the impact of each subplan on customer traffic (for various risk factors) and compute its cost. To speed up simulations, we build *quotiented* network graphs, a compressed representation of a data center network while ensuring its estimation accuracy (§3.3).

Account for failures: Data centers have frequent failures that lower network capacity and impact customers. It is challenging to estimate the impact of a change due to the sheer number of failure scenarios that need to be taken into account. We introduce the notion of equivalence failure classes similar to equivalent subplans (§3.4).

We evaluate Janus on large-scale Clos topologies [40] and Facebook traffic traces [37]. Our evaluation shows that Janus only needs 33~71% of the cost compared to current best practice approaches and can adjust to a variety of network change policies such as different cost functions and different deadlines. Janus generates plans in real-time: it only takes 8.75 seconds on 20 cores to plan a change on 864 switches in a Jupiter-size [40] network (61K hosts and 2400 switches).

2 Challenges and key ideas

In this section, we formulate the network change planning problem. We use examples to discuss strawmen (maximum residual capacity planners) and their limitations. We then summarize Janus's design addressing these limitations.

2.1 Risk assessment for network changes

We focus on *planned network changes* (such as upgrading switch firmware or replacing faulty links and switches) where operators can reliably prepare ahead of time. Such changes are typically at a larger scale and require more time than unplanned changes—ones that are in reaction to unexpected failures (e.g., mitigating a fault).

Risk assessment is critical for planning such changes: these changes reduce the residual network capacity and leave less headroom for dealing with unexpected events—such as traffic variations, concurrent failures, and failed changes.

To plan a network change, we consider operator specified risks and probabilities and estimate their impact on customers and the corresponding penalty to operators (i.e., cost). We choose a plan that minimizes the expected cost —operators can choose to minimize other metrics such as 99th percentile to be more resilient to the worst-case events. The steps involved are as follows:

Operator specifies risks and probabilities: Janus relies on operators to provide the types of risks and their probabilities. Some risks are easier to estimate than others; for example, operators keep historical failures of devices, which makes it easy to determine the risk of failures for network devices [2, 17, 22]. Operators also keep historical traffic matrices [44], which we can use to estimate the risk of traffic variations. However, there are other risks which are harder to measure, for example, the risk of losing customers during downtimes, the impact of downtimes during high profile events such as Black Friday, or the risk of delaying a pushing a security upgrade. We posit that even though we cannot measure the impact of these risks accurately, allowing operators to express such types of risks (with estimates or best guesses) allows for better planning decisions.

We refer readers to site reliability engineering (SRE) books, blog posts, and talks [7, 10, 34] for more detail on techniques to estimate risks. Improving these techniques is a research topic in and of itself and is out of scope for this paper.

Estimating the impact on customer traffic: We next estimate the impact of these risks on customer traffic during network changes. We measure impact by counting the percentage of ToR pairs experiencing packet loss (similar to prior work [44, 45]). We consider ToR pairs (as opposed to host pairs) to reduce the traffic matrix size while preserving the traffic dynamics inside the network [44]. We use packet loss as our measure of impact as it is an important customer

experience indicator [45]. Our solution can be extended to support cost functions defined on throughput and latency.

The impact is a random variable that depends on the probabilities of traffic matrices and risks. We run Monte Carlo simulations to estimate the impact under various traffic matrices and risks. For example, we model concurrent failures by enumerating failure scenarios and their probabilities. Under each failure scenario, the network has a lower capacity (removing all the switches that fail in this scenario). We simulate and measure the impact on customer traffic.

Assessing the cost to operators: Customer impact ultimately translates to operator cost because cloud providers have to refund customers for missing any service-level agreements (SLAs). These functions are often staged: For example, Amazon uses a staged function for refunding credits for availability violations: it provides 10% refund between 99.99% and 99.0% uptime, 30% refund for anything below 99.0% uptime [4]. Similarly, Azure provides its own version: 10% refund between 99.99%-99.0% uptime, 25% refund for 95%-99%, and 100% for anything below [5]. These functions may differ depending on the type of service [4, 5] and customer settings (e.g., enterprise agreements [6]). For example, operators may want to assign a higher penalty when interrupting critical systems, such as lock services that many other systems depend on, than interrupting background jobs (e.g., log analysis systems). Similar to customer impact, the cost is also a random variable given various risk probabilities.

The change planning problem: We define a network *change* as a *set* of operations on switches or links. When applying each operation, we move traffic away from the associated switch or link (drain), apply the operation, and move traffic back (undrain). *A plan of execution* is a partitioning of changes into subsets where changes in each subset run concurrently. We refer to each subset of changes in a plan of execution as a *subplan*. Given a plan, we compute the cost as the sum of the cost of all the subplans (i.e., steps).

Janus searches for the best plan that minimizes the expected cost[1] given an operator-specified deadline. Operators set deadlines to ensure bug fixes and feature updates are done in a timely fashion. Operators may also set other planning constraints (e.g., plan cable replacement according to the technician's work hours) and tie-breaker policies for plans with equal cost (e.g., select the plan that finishes faster when multiple plans have equal cost).

Janus tunes the plan in response to traffic variations, failures, and other sources of risks. When the risk of continuing a change is too high, operators can opt to rollback the change.

2.2 Challenges

Given a deadline for applying a change, operators follow rules of thumb that guides them to devise a plan. For example, Google [17, 40] uses a capacity-based planner that at every step changes an equal number of aggregate switches in each pod and an equal number of core switches, which leaves an equal amount of residual capacity at each step. Such rules of thumb typically aim to maximize the minimum residual capacity during the change on the operator's network.

	uptime	refund	uptime	refund	uptime	refund
Staged-1	<95%	100%	<99%	25%	<99.95%	10%
Staged-2	<95%	50%	<99%	25%	<99.99%	10%
Staged-3	<95%	100%	<99%	30%	<99.99%	10%

Table 1. Example staged cost functions from cloud providers

Subplan	C1	A1	A2	C1, A1	C1, A2	A1, A2	C1, A1, A2
%ToR pairs	0.1%	0.1%	0.1%	0.2%	2%	2%	4%

Table 2. An example of different subplans impacting different percentage of ToR pairs.

Without having additional information about when, where, or how badly traffic variations and failures happen, planners that maximize the minimum residual capacity (MRC planners) are the best planners for absorbing the impact of worst-case events in the network.

However, in data centers, operators continuously monitor traffic variations and failures [37, 40]. This means we have an opportunity to do much better than MRC if we consider these factors when planning network changes. We use a few examples to discuss MRC's limitations and where there is an opportunity to improve.

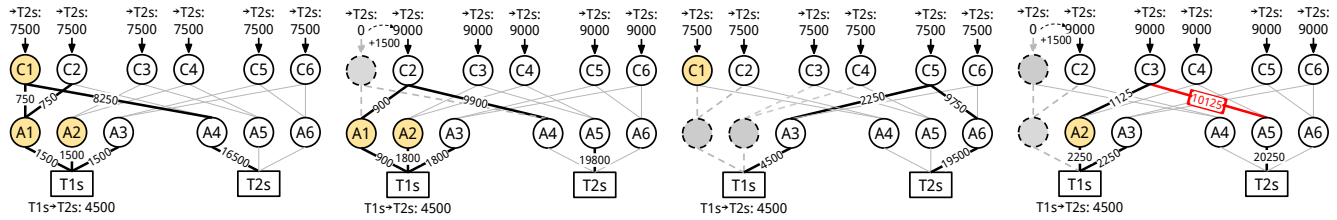
Say we want to upgrade switches A1, A2, C1 in Fig. 1. Given a deadline of 2 steps, an MRC planner may upgrade switches A1, C1 and then A2. This plan ensures the minimum ToR capacity to any other ToR is $\frac{2}{3}$ of its original capacity.[2] However, we show that this plan has more cost than alternative plans under some traffic settings and cost functions.

Plan choices depend on spatial traffic distribution. Consider that traffic from T1s and T2s is 4.5 Gbps, and traffic from the rest of the network to the T2s is 45 Gbps. The MRC plan of upgrading A1 and C1 in the first step causes congestion at links between C3/C4/C5/C6 and A4/A5/A6 (Fig. 1(d)). Instead, if we upgrade two aggregate switches (A1, A2) and then C1, there is no congestion (Fig. 1(c)).

Plan choices depend on temporal traffic dynamics. Let us consider a different scenario where the *steady-state* traffic between the T1s to T2s is for the majority of the time around 10 Gbps and the rest of the network to T2s is on average 45 Gbps. Say when we start the upgrade task, the *current* traffic between the T1s to T2s becomes 4.5 Gbps. MRC, which upgrades A1 and C1, still causes congestion. However, if we know about the temporal traffic changes (i.e., the steady-state is 10Gbps), we can choose to upgrade C1 now and upgrade A1 and A2 after. Delaying upgrading C1 to later means we may never have the chance to upgrade it safely later because of steady-state traffic dynamics.

[1] We can also minimize other statistics such as 90^{th} percentile.

[2] Other plans leave less residual capacity: Upgrading A2 and C1 first (and then A1) reduces the capacity of ToRs in the first pod (T1s) to ToRs in the second pod (T2) by 50%. Similarly, upgrading A1 and A2 reduces the network capacity for ToRs in pod one to one-third.



a. No upgrades. **b.** Upgrading C1 (no congestion) **c.** Upgrading A1, A2 (no cong.) **d.** Upgrading A1, C1 (congestion)

Figure 1. The impact of different subplans. ToR to aggregate links are 40Gbps and aggregate to core links are 10Gbps. The traffic from T1s to T2s is 4500 Mbps; other traffic to T2s are 6*7500= 45000 Mbps. The change task is to upgrade A1, A2, and C1 (yellow circles); Grey circles are switches under changes. The network runs ECMP: numbers on each link indicates the traffic on the link.

Plan choices depend on cost functions. The cost function further complicates change planning. Suppose based on the current traffic dynamics, the probabilities of impacting traffic for different subplans are summarized in Table 2. Inspired by clouds today, we define three types of staged cost functions in Table 1. For Staged-3 function, the optimal plan choice is to upgrade A1, A2, and C1 in three steps and sequentially (cost of $10 + 10 + 10 = 30$). However, for Staged-1, the optimal plan choice is to upgrade A1, A2, C1 concurrently (cost of 25). If Staged-1 returned 35 instead of 25, then the plan choice would have been the same as Staged-3. Alternatively, if we were upgrading 4 switches (instead of 3), each upgrade incurring 10 units of cost, then the best plan would be to upgrade all switches concurrently.

Other factors that impact the plan choice. The best plan also depends on other factors: topology, failures, and routing. In a Fat-tree topology, we need to be careful about the aggregate core connectivity [30] but not in a Clos topology (where each aggregate has the same set of connections). The best plan also depends on failures: if switches in a given pod have higher failure rates than other pods (e.g., because they are from different vendors), we have to apply their changes more carefully. Finally, different data centers employ different routing algorithms which react to failures and traffic variations differently [19, 40].

Key challenge: In summary, the plan choice depends on factors such as spatiotemporal traffic dynamics, cost functions, topology, routing, and failures. Such diversity makes it challenging to find a heuristic that works for all cases.

We could search for all possible plans, but there are many possible plans for upgrading n switches: the number of possible k-step plans is the number of ways we can divide n switches into k subsets (i.e., Stirling number S(n,k)) where $1 \leqslant k \leqslant n$). Therefore, the number of plans grows super-exponentially ($\sum_{k=1}^{n} k!S(n,k) \approx O(\frac{n!}{\log_e^{n+1} 2})$). For example, for a change involving 500 switches, we have more than 3.4×10^{1213} plans. Even by exploiting the high degree of symmetry in data center topologies, the number of plans still remains prohibitively large. The same upgrade task (for 500 switches) in Jupiter topology [40] has more than 2^{120} plan realizations—this is true even after we eliminate plans that violate operator specified constraints.

The problem is exacerbated when we consider traffic dynamics and failures, forcing us to make planning decisions in real-time and in response to in-network events. The planning decisions should be faster than the operation time (or by the time we can apply the plans they are obsolete). Since many network operations, especially ones on switches, take minutes [1], the planning time itself should be in seconds.

In summary, Janus has to be *adaptive* and support a variety of constraints, *scalable* and work with the largest data centers, and *fast* so that it can select plans in real-time.

2.3 Janus's key ideas

Given a set of switches (or links) involved in a network change, the plan navigator builds a repository of candidate plans (i.e., an ordered set of subplans) based on operator specified constraints. Janus continuously monitors traffic dynamics, evaluates the cost of plans using a simulator, and selects a plan with the minimum cost. After each subplan (step) finishes, Janus adjusts the plans for the remainder of the change based on traffic changes and failures.

Our key idea is to leverage the high degree of symmetry in data centers to navigate the large planning space in real-time. We show how we use network automorphism using an example topology in Fig. 2:

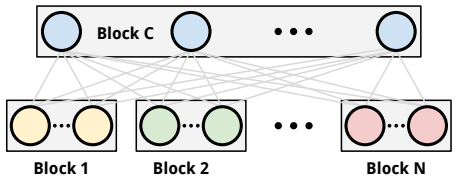


Figure 2. Janus decomposes network graphs into blocks

Identifying blocks of equivalent switches: We first identify switches that have the same connectivity and routing tables and group them into *blocks*. Switches in each block are, for all traffic purposes, indistinguishable (§3.2). Therefore, a subplan operating on a block needs to only care about the *number* of switches and not *which* switches it is changing. Fig. 2 shows several core and aggregate blocks. Given n blocks, we can describe a subplan as a tuple of n numbers $< b_1, ...b_n >$ where the i_{th} index is the number of steps for upgrading the i_{th} block. Operators can further reduce this space by taking similar actions on different blocks, i.e., merge two blocks to build superblocks (§3.1).

Identifying equivalent subplans using graph automorphism: For most data center networks, the number of blocks is large and so is the number of subplans. However, many subplans, even on different blocks, have the same impact on customers. For example, in Fig. 1, upgrading A1, C1 is equivalent to upgrading A2, C3 even though A1 and A2 are in different blocks. *Network automorphisms* can identify such equivalent subplans. Equivalent subplans speed up planning by confining risk simulations to unique subplans (§3.1).

Estimating the cost of subplans with scalable Monte Carlo simulations: We run Monte Carlo simulations on all possible traffic matrices during the network change. We discuss how we predict future possible traffic matrices and handle prediction errors in §3.3. For each traffic matrix and its probabilities, we run flow-level simulations to estimate its risk of impacting customer traffic and the corresponding costs. We then compute the expected cost under all scenarios.

Monte Carlo simulation on many different TMs take a long time, e.g., the simulation for a single TM takes minutes even for a modest size data center with 600 switches (a relatively small data center) on a single core (§5.3). To reduce simulation time, we leverage network symmetry to simulate flows on a *quotient graph* instead of the original topology but ensure the estimated risk remains the same (See §3.3).

Failure equivalence: To estimate the cost of a large number of failure scenarios, we introduce *failure equivalence classes* similar to equivalent subplans. Data centers typically use a fail-stop model to deal with failures. This makes failures similar to subplans as they both bring down a set of switches, links, or line cards. We thus can model failures as subplans taking down the failed elements for a change task.

3 Janus Design

Janus has to adapt to a variety of conditions (e.g., traffic dynamics, failures, and cost functions) and scale to large networks and large changes. For that, Janus leverages the high degree of symmetry in data center topologies to search the large planning space. Fig. 3 shows the four key components in Janus: (1) Given the topology and routing information, Janus starts by identifying blocks of equivalent switches; (2) Janus then identifies equivalent subplans across blocks; (3) Janus runs Monte-Carlo simulations using quotient networks to estimate the impact and cost of each subplan and selects plans accordingly; (4) To estimate the impact of failures, we identify equivalent classes of failures in the same way as equivalent subplans.

3.1 Identifying blocks of equivalent switches

Given topology and routing information, we group switches connecting to the same hosts and have the same routing table into *blocks*. There are many such blocks in data centers today (Fig. 2). A block is fully specified by two values: a switch and the number of such switches in that block. Operators can then granulate the number of steps it takes to upgrade switches in a block, e.g., $\frac{i}{k}$ with $0 \leqslant i \leqslant k$ of switches in each block. Blocks are a good representation because they are high level enough for operators to understand and are succinct enough for planning purposes—we only need to know a switch in that block and the number of such switches.

Operators can further make the planning space coarser by collapsing multiple blocks into one and using the same steps to upgrade them. We call these groupings *superblocks*. The intuition behind superblocks is that in large data center networks, there is enough path diversity and redundancy that many *close* plans have a similar impact on traffic. For example, for two pods with 20 aggregate switches, upgrading 3 switches in pod 1 and 4 switches in pod 2 versus upgrading 4 switches in both pods are practically similar from the residual capacity standpoint. Therefore, instead of searching in the exact planning space, we can search in a coarse-grained planning space with superblocks.

There are many ways to group blocks into superblocks. For example, they can build superblocks based on communication patterns, so that they upgrade two blocks talking with each other as one entity; or by spreading blocks with high traffic across different super blocks—so that two blocks with high traffic have the opportunity of being upgraded separately; or in its simplest form group blocks based on the type of switches, e.g., upgrade all aggregate blocks together and upgrade all core blocks together.

3.2 Finding equivalent subplans

The most computationally intensive part of planning is estimating the impact of subplans on large scale topologies. The saving grace here is that many subplans have an equal impact under all settings. If we had an automated way of identifying such subplans, we then would only need to simulate for each unique class of subplans. However, checking the equivalence of two subplans is not straightforward because of topological and routing complexities (§2.2). Here, we formalize the notion of subplan equivalence and discuss how we can efficiently find such subplans.

Definition 3.1 (Subplan Equivalence). We define two subplans s_1 and s_2 to be equivalent in a network N, when a renaming function f exists that satisfies three properties:

1. *P1: Equivalent topologies.* f maps switches in G_{N/s_1} (i.e., the topology after removing switches in the subplan s_1) and G_{N/s_2}, where for each link (A, B), for switches A and B in G_{N/s_1}, there exists a matching link, $(f(A),f(B))$, in G_{N/s_2} with the same capacity.
2. *P2: Equivalent traffic matrices.* The traffic volume between ToRs (A, B) in s_1 is the same as the traffic volume between $(f(A), f(B))$ in s_2.
3. *P3: Equivalent routing.* For a routing algorithm that makes forwarding decisions based on the topology in P1 and the traffic matrix in P2, all the forwarding tables in N/s_1 and N/s_2 are equivalent. That is, for switch $S \in N/s_1$ and $f(S) \in N/s_2$ we have: for the i_{th} rule on switch S of the form (src, dst, action) there exists an i_{th} rule $(f(\text{src}),$

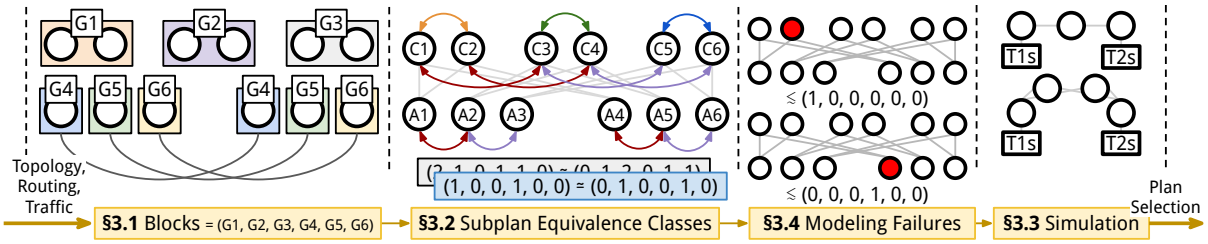


Figure 3. Janus's Design

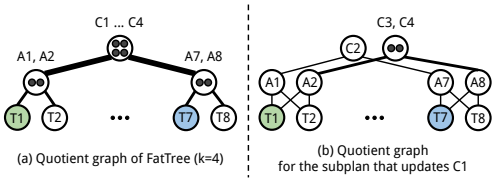


Figure 4. Example of quotient graphs for FatTree topology.

$f(\text{dst})$, $f_A(\text{action})$) on switch $f(A)$ in N/s_2, where action is a set of (nexthop, weight) tuples, and $f_A(action) = \{(f(\text{nexthop}), \text{weight}) | (\text{nexthop}, \text{weight}) \in action\}$.

For example, in Fig. 5, using this definition, a subplan, s_1, that updates C1 and a subplan, s_2, that updates $C4$ are equivalent. To show this, consider the renaming function, f, shown in the table of the same figure. Using this function, the topologies in N/s_1 after removing $C1$ and N/s_2 after removing $C4$ are equivalent (i.e., isomorphic), because we can map {C2 → C3, C3 → C1, C4 → C2, A1 → A2, A2 → A1, A7 → A8, A8 → A7, ...}. Similarly, since the traffic sources T1, T2, ..., T7, T8 map to themselves, their flows remain intact and the flow volumes between the pairs remain the same. If we use a routing algorithm that makes forwarding decisions based on P1 and P2, then P3 is also satisfied.

Many routing algorithms are equivalent, i.e., they match P3. For example, ECMP shortest path routing only uses topology information to devise multiple shortest paths between pairs of hosts. Similarly, WCMP matches P3 because its routing decisions only depend on the topology. It is possible to extend this definition to other routing algorithms that rely on switch configurations, such as BGP, by defining an equivalence between the switch configurations.

Theorem 3.1. If traffic forwarding only uses the topology, traffic, and routing as defined in Definition 3.1, two equivalent subplans have the same impact under all traffic scenarios.

Proof sketch: P1, P2, and P3 guarantee that traffic between two ToRs traverses in the same exact manner throughout the network and thus sees the same impact during the execution of the two equivalent subplans: we can find a bisimulation between the two subplan networks (see §A).

Subplan equivalence with graph automorphism. A naive approach to finding equivalent subplans may enumerate all the subplans and do a pairwise equivalence check. However, this takes too much time. Instead, we focus on finding equivalence classes of subplans: if we find a renaming function for the network that preserves P1, P2, and P3 before applying a subplan, we could rename the network first. The subplan lacks enough information to tell the difference between the original and the renamed network. Thus, we can apply the subplan on the renamed network, and in the process make it change a different set of switches. For example, in Fig. 5, if we rename C1 to C4 and C4 to C1, a subplan that operated on C1 now can also operate on the renaming of C1, that is C4. Concretely:

Theorem 3.2 (Network Automorphism)**.** For a subplan, s, and a renaming function, f, that maps network N onto itself, if f preserves properties P1, P2, and P3, the two subplans s and $f \cdot s$ (the subplan after applying the renaming function to its elements) are equivalent.

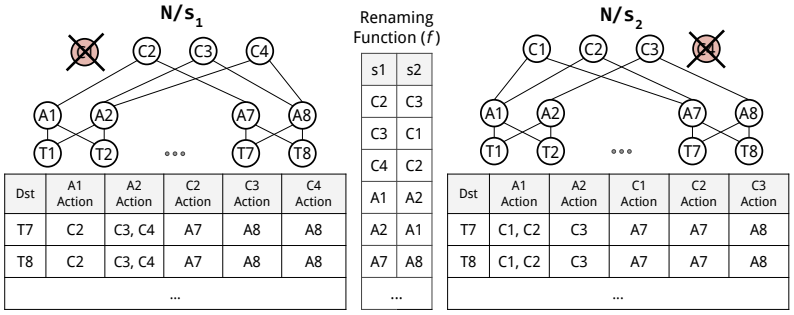
Proof sketch: By finding a renaming f that preserves P1, P2, and P3 for the network, N, we guarantee we can find a renaming function between N/s and $(f \cdot N)/s$. Similarly, we can also prove that a renaming function between $N/(f \cdot s)$ and $(f \cdot N)/s$ exists. Therefore, $N/(f \cdot s)$ and N/s are equivalent (see appendix for proof §B).

For example, consider the renaming function f in Fig. 5 where f, maps {C1 → C4, C2 → C3, C3 → C2, C4 → C1, A1 → A2, ..., A7 → A8, T1 → T1, ..., T8 → T8}. The two subplans $(f \cdot N)/s$ and N/s are equivalent under the renaming function f, because they preserve P1, P2, P3. Similarly, the two subplans $(f \cdot N)/s$ and $N/(f \cdot s)$ are equivalent under the identity function, which indeed preserves P1, P2, and P3. Therefore, $N/(f \cdot s) \equiv N/s$.

The theorem shows that using the set of renaming functions for a network N, we can generate many subplans equivalent to any other given subplan.

Given a set of renaming functions and a set of subplans, we can use the renaming functions to partition the subplans into equivalence classes. We observe the set of renaming functions forms a permutation group (it has identity, inverse, associativity, and closure properties). Using this group, we define a group action on our subplans: $G \cdot s = \{\{f \cdot v | v \in s\} | f \in G\}$ where G is the group of renaming functions, s is a subplan, v is a switch in the subplan, and f is a renaming function. This action preserves the basic properties of group actions: compatibility and identity. Group actions partition the set they act on—by using the group action, we can partition the subplan set to find equivalence classes of subplans.

For example, Fig. 6 shows three renaming functions for a k=4 FatTree. The three functions are: {f1: (C1 C2), f2: (C3 C4), f3: (C1 C3)(C2 C4)(A1 A2)(A3 A4)}. We can use the three renaming functions f1, f2, and f3, subplan {C1} is equivalent



Dst	A1 Action	A2 Action	C2 Action	C3 Action	C4 Action
T7	C2	C3, C4	A7	A8	A8
T8	C2	C3, C4	A7	A8	A8
...					

Dst	A1 Action	A2 Action	C1 Action	C2 Action	C3 Action
T7	C1, C2	C3	A7	A7	A8
T8	C1, C2	C3	A7	A7	A8
...					

Figure 5. Example of equivalent subplans. The actions show forwarding decisions at each switch for a rule matching destination (Dst).

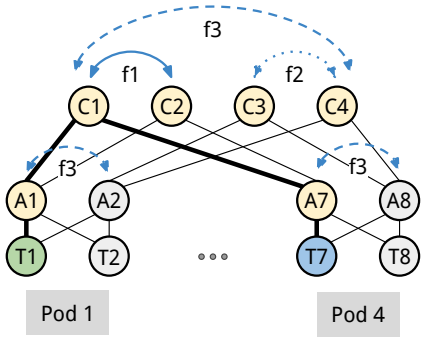


Figure 6. Renaming functions for finding equivalent subplans

to subplan f1·{C1} = {f1·C1} = {(C1 C2)·C1} = {C2}, f3·{C1} = {C3}, and f3f2·{C1} = {C4}. Similarly, a subplan s2 = {A1, C2} is equivalent to f1·s2 = {A1, C1}, f3·s2 = {A2, C3} and f3f2·s2 = {A2, C4} but not to {A2, C1}. This is because no possible combination of generators renames A1 to A2 only.

Encoding for graph automorphism engines. We can use a graph automorphism engine to find the renaming group that preserves P1, P2, and P3. Graph automorphism engines typically find automorphism groups of vertex-colored graphs—a vertex-colored graph is a graph where a coloring function, C, assigns colors to nodes. The automorphism engine guarantees the permutation of the nodes respects the coloring: we can only permute nodes that have the same color. We can define colors in a way that two nodes have the same color when they satisfy properties P1, P2, P3.

We define a label tuple for each node with one label per property in Theorem 3.1. Two nodes are permutable, if their labels exactly match, i.e., all the properties of Theorem 3.1 hold. To build the labels:

For P1, take the topology as an input to the graph automorphism engine. To encode each links' bandwidth, we assign a unique label per unique link capacity to each edge, e.g., if the data center topology uses 40G and 100G links, we use two unique labels to describe each link.

For P2, we assign a unique label to each traffic source. This coloring ensures that for every pair of traffic source, (A, B), there exists a pair, (f(A), f(B)), in the renamed network—the number of unique colored pairs matches the number of cells in the traffic matrix. If two traffic sources see similar traffic, we can allow the coloring to rename them by using the same labels. This ensures that each pair in the network has a unique traffic label assigned to it.

As P3 depends on P1 and P2, and we already label those properties, the same labels can be used for P3.

After labeling, we assign a unique color to each unique label. The number of unique colors is equal to the number of unique label tuples in the network. It is true: no polynomial algorithms are known for the general case of graph automorphism, but many polynomial-time algorithms exist for special cases of this problem [31]. In particular, we found Nauty [32] can find the automorphism groups of a large data center with 2,400 switches in 6.25 seconds (§5.3), which matches the real-time requirements of planning—we observed similar computation times for expanders [39], fat-tree [19], and bCube [13] topologies.

3.3 Estimating cost with Monte Carlo simulations

We measure the number of ToR pairs experiencing packet losses using flow-level Monte-Carlo simulations under various traffic matrices and translate the number based on cost functions. Since we search the entire planning space, we can support various cost functions. We can also extend Janus to support multiple tenants, each with their cost function.

We have to model congestion in the network, that is, how competing ToRs divide (the scarce) bandwidth among themselves. For that, we run max-min fairness to decide how much bandwidth each ToR gets (similar to [23]). This objective matches that of TCP. We also consider the network's routing tables, which is important as the network reacts to failures and traffic variations through routing changes. This is in contrast to previous work that used multi-commodity flow (MCF) for simulating data center traffic [38, 41]: while MCF is a reasonable estimation of the bisection bandwidth, it ignores routing algorithms and fairness objectives.

Our simulation relies on knowing possible traffic matrices during the change. Today, data center operators continuously collect traffic matrices (TMs). We use the current TM to represent what happens in the next planning interval and use the past TMs to predict the TMs for the remainder of the change. Previous work [44] use a similar approach and find that the current TM is a good estimation of the future (e.g., the next step of the plan)—the intuition is that the TM does not (typically) change dramatically in such a short time. When traffic is unpredictable, and our predictions are not a good representative sample of future TMs, Janus may lose some of its temporal benefits—because Janus's view of the future was incorrect. However, Janus still gains spatial benefits due to more accurate short-term predictions.

With max-min fairness as our objective, we have to find ways to speed up simulations. This is especially important for larger data centers where simulations may take much

longer to complete. We use the inherent symmetries in the data center network to achieve faster simulations: we build a quotient graph per subplan by merging switches with the same forwarding rules (e.g., all the ECMP paths). Fig. 4(a) shows the quotient graph for a k=4 FatTree. Fig. 4(b) shows the quotient graph for a subplan upgrading C1.

We build quotient graphs by using network automorphism (see §3) to identify equivalent sets of switches under P1, P2, P3. We run the group action G on individual switches (instead of subplans) and build an equivalence relation on the switches. We can merge equivalent switches because they have similar, per-link, traffic patterns. For all switches in the same equivalence class, we build a super-switch and have one virtual forwarding table across all original switches—we can merge the forwarding tables if they are the same, e.g., all the core switches have similar forwarding tables in a Fat-tree network. To add links between super switches, we only have to ensure the link capacity between super-switches is the same as the original network. Since the new topology has far fewer links/paths, we can simulate the network much faster.

3.4 Handling failures

Failures are a common risk source when planning network changes. Google reports that nearly 68% of failures occur when a change is in progress [17]. Janus models failures as capacity reductions—a failure on a set of switches remove these switches from the network graph (fail-stop), which increases the risks of impacting customer traffic.

Operators can input failure scenarios and probabilities based on their logging of historical failure events for each vendor [17, 22]. Given failure scenarios and probabilities, we can run simulations to measure impact and estimate the expected cost for each network change plan.

However, the size of failure space is exponential in the number of switches, e.g., to model independent switch failures for 2400 switches, we have 2^{2400} possible scenarios. Instead, we model the most likely failure scenarios that cover P (e.g., 99%) of the most probable failures, i.e., $\Pr[\text{Failures}] \geqslant P$. For example, if switches have 0.1% failure rate in a topology of 2400 switches, we only need to simulate up to 7 concurrent failures (binomial distribution) to cover 99% of failures.

To further reduce the number of failure scenarios, we introduce *failure equivalence classes*, i.e., failures that result in isomorphic network graphs. We can view a failure scenario as a subplan bringing down switches in the failure set (or links/line-cards). Thus, to simulate failures during a change, Janus considers a bigger change task involving both failed switches and change switches. We can then apply the same techniques above to estimate cost under failures.

4 Implementation

Janus has 7.2k lines of C code. It operates in three steps:

Operators specify the change, the cost function, and the risks. Operators can input arbitrary change requests into Janus. For each change, operators specify the length of its operations and a deadline for the change. Operators then define a cost function where the input is the percentage of ToR pairs impacted during a change interval, and the output is the associated cost. Operators can also specify time-based cost functions—to model time constraints during planning, e.g., to emphasize the risk of delaying a critical bug fix. In its current state, Janus can model concurrent failure of switches in the data center where failures are independent. We chose to implement this failure model following the example of previous work [28]. For more complex failure models, e.g., correlated failures, we rely on previous work and use their proposed sampling techniques [15] to cover the failure space.

Simulation. We assume the data center network upholding Max-min fairness for the traffic it routes through its network. Max-min fairness is also commonly used [14, 23, 27] to model how TCP flows affect each other during congestion. To model Max-min fairness, we simulate the network while respecting the routing, topology, and link constraints. Since our simulation uses P1, P2, and P3 (in Definition 3.1), it satisfies the conditions of Theorem 3.1. Therefore, we can rely on Theorem 3.1 to reduce the subplan search space. For each setting, we convert the network into a quotient network, then run our network simulator on the quotient network.

Janus uses the current TM as a prediction of the traffic for the upcoming subplan and the 10 previously observed TMs as a prediction of traffic for the rest of the plan. In practice, data center operators may have better traffic predictors and are free to use their own.

Estimate cost in real-time and adjust the plan. At runtime, Janus goes through all the subplans, applies the failure model on each subplan, and uses the TM predictions to estimate the impact of each choice. It then measures the impact of each plan and chooses a plan with the lowest expected cost. If there are multiple candidate plans, Janus picks the plan according to operator-specified tiebreakers. Other termination conditions are also possible; for example, ones that return the best plan within a deadline.

Scalability. Monte-Carlo simulations are easily parallelizable: we can run each scenario (i.e., subplan and traffic matrix) independently from others and on different machines/cores. We can then merge the results of all scenarios to build the cost random variable of each subplan.

Plan ports and links changes. Janus supports port and link changes by modeling them as *virtual switches*. To plan changes for links, we replace each link in the network graph with a passthrough virtual switch that sends the incoming traffic on each of its port to its other port. Any operation on links can thus be modeled as an operation on virtual switches. The virtual switch abstraction allows us to use the previous theorems for scaling. Similarly, to handle ports, we model each as a passthrough virtual switch similar to links.

Janus supports line card changes (e.g., replacements). A line card is a collection of N ports. We can substitute a virtual

switch with $N + N$ ports in place of a line card. We connect the first N virtual switch ports to the links and the second N ports to the switch where the line card belongs. The routing table of the virtual switch is, again, a passthrough table where the first port is directly connected to port $N + 1$, the second port to port $N + 2$ and so on.

Rollbacks. It is possible that due to unexpected events, a change task becomes costly, e.g., because there are no suitable plans or simply because the change is faulty. In that case, operators would want to rollback the upgrade. Janus generates rollback plans instantly: A rollback plan is a change plan for a subset of original change tasks.

Failed instructions. Operators may fail to follow Janus's instructions accurately. In such cases, operators can accommodate by adding these failed instructions back into the change. For example, if during the execution of a change, Janus issues an impossible instruction, e.g., because switches are physically too far apart, operators can mark these instructions as incomplete so that Janus schedules them in the upcoming intervals.

Janus offline. There are cases where operators cannot spare the computational cost of real-time planning, e.g., if they lack good traffic predictors (so they have to model many TMs) or when using complex failure models or simulators that prohibit real-time planning. Under such circumstances, operators can use Janus in what we call the offline-mode.

In offline-mode, operators feed a large number of traffic matrices (possibly from previous days) and historical failures into Janus. For example, operators could use historical traffic of recent days to predict future days [9, 33, 43, 44]. Janus then finds a *static* plan for the change that will highly likely minimize the expected cost under the provided traffic and failure settings. Operators may also want to change the objective of Janus to, for example, minimizing the 99th percentile of the risk, so that the plans that Janus suggests are resilient to worst-case scenarios. This mode is very similar to MRC, as both planners find static plans. However, Janus still enjoys the spatial benefits, and it also respects operators planning constraints such as deadlines and cost functions.

5 Evaluation

Here, we demonstrate the cost reduction, scalability, and generality of Janus using large-scale data center topologies, network change tasks, and realistic traffic traces. Our evaluation shows that Janus only needs 33~71% of MRC cost and can adjust to a variety of network change policies such as different cost functions and different deadlines. Janus generates plans in real-time: it only takes 8.75 seconds on 20 cores to plan a change on 864 switches in a Jupiter-size [40] network (61K hosts and 2400 switches).

5.1 Evaluation settings

Topology. We evaluate Janus on Clos topologies (Table 3). We use four different scales ranging from the default *Scale-1* which updates 8 pods (3.8K hosts and 168 switches) to a scale comparable to the size of Google's Jupiter topology [40] (61K hosts and 2400 switches).

Topology	# switches in the DC			# hosts	# upgrades (cores, aggs)
	# pods	# cores, aggs, ToRs	# switches		
Scale-1	8	8, 64, 96	168	3840	72 (8, 64)
Scale-4	16	24, 192, 384	600	15360	216 (24, 192)
Scale-9	24	54, 432, 864	1350	34560	486 (54, 432)
Scale-16	32	96, 768, 1536	2400	61440	864 (96,768)

Table 3. Configurations and change task for each topology. We upgrade all core and aggregate switches in all the pods.

Traffic. We generate a cloud-like trace using Google job traces to model the size and arrivals of tenants [42] and Facebook traffic traces [37] to model the traffic for each tenant. Specifically, for each tenant, we decide its arrival and leaving times and the number of ToRs it runs on based on the Google job trace. We then randomly select its traffic type: either Hadoop or web traffic, and select the corresponding trace from Facebook. We generate 400 such traffic matrices at a 5-minute interval—Minute-level TMs map to the granularity that operators use to measure SLOs in data centers today. By default, our traffic has an average maximum link utilization (MLU) of 80% (the median link utilization is 17%). We use average MLUs ranging from 65% to 95%.

Network change tasks. We evaluate Janus on a large change so that it has to explore a large planning space. Concretely, we upgrade all core and aggregate switches in the data center. Table 3 shows the details for each upgrade task. We assume each upgrade takes one timeslot (5 minutes), i.e., one traffic matrix, matching the length of firmware upgrades of today's switches [1]. Each upgrade is repeated 50 times across different hours. We report the average and standard deviation of this cost. We set deadlines of 2, 4, or 8 steps for finishing the change and choose 4 as default—this means that MRC leaves 50%, 75%, 87.5% of residual capacity in the network at each step.

Cost functions. We define three types of staged cost functions following the shape of the refund functions of major cloud providers such as Azure, Amazon, and GCloud (Table 1)[3]. To test the generality of Janus under various cost functions, we also evaluate a range of synthetic functions, namely, logarithmic, linear, quadratic, and exponential, where the input is the number of ToR pairs experiencing packet loss and the output is a cost value between 0 and 100. The details of these functions are in §C.

We use the Staged-1 function by default. One should only interpret the relative cost differences across approaches and settings, not the absolute values because despite using cloud cost functions (that operators use today in practice), it is difficult to gauge whether the combination of our choices of

[3]Even though we use these functions differently than the clouds today, we suspect that the shape and nature of cost functions will be the same.

cost functions, topology, and traffic matrices represent what operators experience in practice.

Planners. We evaluate two planners: (1) Janus which uses the last 10 and the current traffic matrices to plan the change; Janus adjusts the plan based on traffic changes (§4). (2) Janus *Offline* which uses history traffic to choose a fixed plan that does not change during execution. (3) *MRC*: a planner that maximizes the residual capacity at each step of the plan §2.2, similar to the state-of-the-art solutions used in data centers today [40].

Evaluation metrics. We report the expected cost of applying a network change while meeting each change's deadline. For each data point, we run 50 experiments and take the average.

5.2 Cost savings over MRC

Spatial benefits: We start our evaluation with a simple scenario of static traffic (using a randomly chosen TM). Because the traffic does not change, Janus online is the same as Janus offline. Janus achieves lower or equal cost to MRC under all MLU settings (Fig. 7a). At 85% MLU, Janus takes only 25% of the cost of MRC (2.5 units of cost vs. 10 units). When MLU is low (e.g., $\leqslant$ 75%), there is enough capacity in the network so both Janus and MRC can pick plans that apply the change with zero cost. Janus picks plans that upgrade more switches initially and fewer switches later on and only for busy pods. In contrast, MRC equally allocates the switches at each step. When MLU is high (e.g., $\geqslant$ 80%), every step of the plan is likely to impact ToR pairs. Janus automatically changes its goal to choose plans with a fewer number of steps to minimize the duration of traffic disruption.

Temporal benefits: Next, we evaluate Janus with tenant and traffic dynamics as discussed in our evaluation settings. Fig. 7b shows that both Janus and Janus offline have a lower cost than MRC under all MLUs. On average, Janus has 33~71% of the cost of MRC. At 85% MLU, Janus takes only 52% of the cost comparing to MRC. This is because Janus can change more switches under a low traffic load and fewer switches under a higher load.

Janus offline does not consider traffic dynamics and thus performs worse than Janus, but still better than MRC. At 85% MLU, Janus offline takes 90% of the cost comparing to MRC. This is because of the spatial benefits mentioned above. In our setting, the spatial benefit is smaller than the temporal benefit because, with tenant dynamics, the traffic shifts across ToRs fast, so there is not as much spatial skewness.

MRC also has higher variance than Janus because it chooses a fixed plan which sometimes performs very poorly. Such lack of predictability makes it difficult to understand the potential impacts of MRC plans on customers. In contrast, both Janus and Janus offline identify the best plan based on operators' policies (including plan deadlines, other constraints, and tiebreakers).

Predictability of traffic. Janus lowers the planning cost even when the traffic is hard to predict. Here, we try 5 different traffic traces where we change the ratio of Hadoop (unpredictable: all to all communication patterns that exhibit on-off chatters) to Web servers (predictable: spatially stable and constant chatter of Web servers to cache servers) users in our trace while keeping the MLU fixed. Fig. 8d shows that Janus saves cost under all settings. As we increase the proportion of Web server to Hadoop users, Janus costs decrease from 74% of MRC-plan to 51%. As traffic becomes less predictable, Janus's temporal benefits disappear, but Janus gains benefit because of spatial patterns.

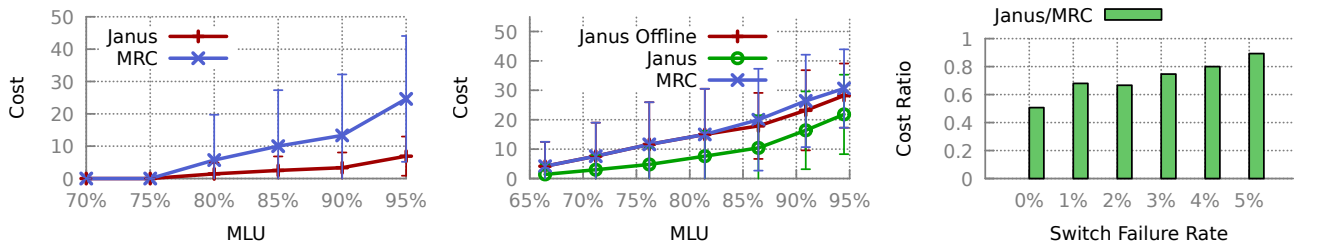
Concurrent failures. Janus also considers the probabilities of failures when it plans network changes. Here we mode independent switch failures using Bernoulli random variables, that is a switch either fails or does not with 1-5% failure rate at every step (i.e., every 5 minutes). Typically, failure rates are lower in data centers, e.g., Gill et al. [16] report 2.7% failure rate for aggregate switches over a year. However, we choose high failure rates to ensure there is a non-zero chance of concurrent switch failures in Scale-1: at 5% failure rate, we expect 4 concurrent switch failures in the space of 80 switches. High failure rate stress tests Janus as it requires the simulation of a much larger failure space: to model 99% of possible failures at 5% failure rate across 80 switches (binomial distribution), we have to consider more than 2.6 trillion failure scenarios: $\sum_{x=1}^{10} \binom{80}{x} \geqslant 2.6\text{trillion}$.

Fig. 7c shows that Janus online has 52% to 85% of the cost of MRC. As we increase the failure rates, Janus becomes more conservative in preparing for potential failures and thus requires a higher cost. MRC does not consider failure rates, and its cost remains the same for all failure rates. With a higher failure rate, Janus gets closer to MRC. This is because, as discussed in §2.2, MRC is a good option when we have little information about failures. Interestingly, as we increase failure rates, we are indirectly reducing our knowledge of failures by increasing the number of failure scenarios that we have to consider. More concretely, to cover 99% of probable failures for 80 switches, we only need to simulate 85k different scenarios at 1% failure rate, whereas that number explodes to 2.6 trillion at 5% failure rate.

5.3 Scalability

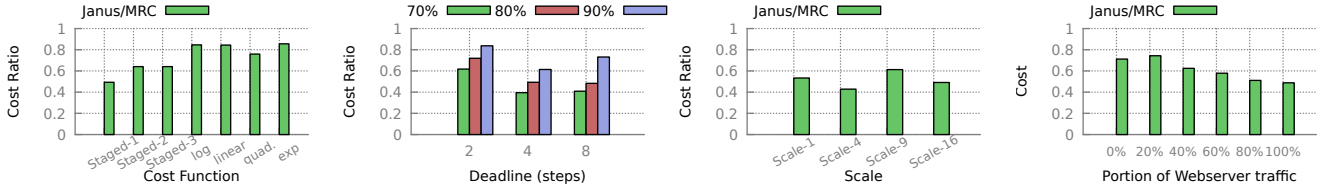
Janus finds plans in real-time even for large topologies. We evaluate on Scale-1 to Scale-4 (61k hosts) topologies. The details of these topologies are shown in Table 3). Janus online plans cost 42% to 61% of MRC plans (Fig. 8c).

Janus spends the majority of its time (>99%) estimating the impact of subplans at every step, which depends on the number of subplans and the simulation time to estimate the impact of each subplan. In §3.2, we discussed how network automorphism allows Janus to reduce both the number of subplans using subplan equivalence and the simulation time through quotient network graphs. Another added benefit is



(a) Avg./std. of plan costs (static traffic) **(b)** Avg./std. of plan cost (dynamic traffic) **(c)** Avg. plan cost ratio under switch failure

Figure 7. Comparing Janus with MRC under various settings.



(a) Janus has benefits for all cost functions. **(b)** Different deadlines. Each color is a different MLU setting. **(c)** Janus has benefits across all data center scales. **(d)** Impact of Hadoop/Webserver user split on Janus plans.

Figure 8. Janus adjusts to operators constraints and cost functions and has universal benefits across all settings. The bars show the average cost of the plans by Janus compared to the MRC planner.

that as subplans are completely independent of each other, we can parallelize Janus very easily by computing the impact of each subplan (or TM) on a different core/machine.

We measure the total running time of Janus across all the steps on one core and report it in Table 4. We also interpolate the time to 20 cores~[4] to show that Janus can plan changes in real-time even for the largest data centers. With 1 core, it takes Janus 175 seconds to plan a change for upgrading 864 switches for Scale-16. With 20 cores, it takes 8.75 seconds.

We also compare the simulation time per traffic matrix for a four-step plan with and without the quotient graph optimization: The running time on one core of Scale-1 improves from 2.9s to 0.01s, a reduction of 290x. Similarly, the running time of Scale-4 improves from 184 seconds to 0.045, a reduction of 4100x–at Scale-4 topology finding a plan could take upwards of 12 hours on a single core. We could not run the flow simulations at Scale-9 and Scale-16 without quotient graphs because we ran out of memory.

5.4 Adaptivity

Janus is adaptive in selecting plans that have low expected cost for various planning constraints and metrics.

Different cost functions: Fig. 8a shows that Janus online and offline are consistently better than MRC under a variety of cost functions. Janus online's plans cost is 64% of MRC under Staged-2 and Staged-3 cost functions and Janus offline's plans cost is 86% of the MRC cost. The results are similar for the Staged-2 and Staged-3 functions as their cost functions are similar when the packet loss rate is low (10% credit for 99.99% ToR pair connectivity). The benefits under Staged-1's cost function is larger (49% of cost compared to MRC) because Azure's cost function has more room for losses (10% credit for 99.95% availability).

Janus online uses 75~85% cost compared to MRC for logarithmic, linear, quadratic, and exponential cost functions. Janus is uniformly better than MRC regardless of cost function as Janus exhaustively searches the entire plan space.

Different deadlines: Fig. 8b shows that Janus has a lower cost than MRC for all deadlines. The cost ratio of Janus follows a U-shape for all MLUs: when the deadline is small, there are fewer candidate plans and thus less room for Janus to reduce cost compared to MRC. When the deadline is far away, MRC touches fewer switches per step and incurs less cost. For deadlines in the middle (where the majority of settings are), Janus has the most gains over MRC. The actual deadline with the best gain depends on the MLU.

Rollback: We show a scenario where the cost estimates provided by Janus helps operators to make rollback decisions. As before, the change involves upgrading all the core and aggregate switches in the Scale-1 topology (72 switches). Janus initially selects an eight-step plan but continuously estimates the cost of other plans and rollback plans, as shown in Fig. 9. At step 5, Janus reports that the expected cost of the remainder of the plan (42 switches left) is 9.901 units (red curve) and the cost of rollback of the initial bit of the plan (30 switches) is 3.354 (green curve). If operators consider the cost of 9.901 to be too high (e.g., because their budget is only 5 units), they may choose the rollback plan. After issuing the rollback, Janus can immediately select a plan for it. For example, Fig. 9 shows two rollback plans provided by Janus: Plan 1 upgrades 17, 12, 1 switches in 3 steps, and

[4]This is an artifact of the code running single-threaded.

Topology (Change size)	Planning time		simulation time per TM	
	1 core	20 cores	Without quotient	With quotient
Scale-1 (72)	2.5 s	0.125 s	2.9 s	0.01 s
Scale-4 (216)	10.06 s	0.503 s	184 s	0.045 s
Scale-9 (486)	35.9 s	1.795 s	Out of mem.	0.149 s
Scale-16 (864)	175.0 s	8.75 s	Out of mem.	0.8 s

Table 4. Janus planning time.

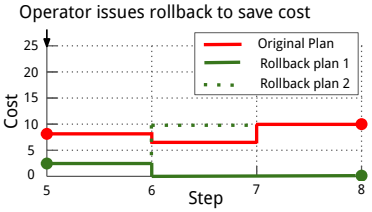


Figure 9. Janus suggests a rollback plan (Green line) that safely revert an ongoing change.

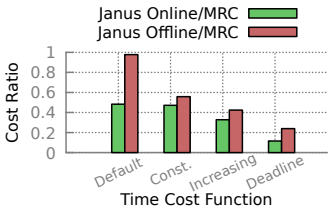


Figure 10. Different cost functions for delayed changes. MRC fails to factor time and incurs heavy cost.

Plan 2 upgrades 17, 13 switches in 2 steps. At step 6, Janus picks Plan 1 as Plan 2 is too risky (cost of 10) due to traffic dynamics.

Delaying changes: In practice, operators may not have a strict deadline but instead, have to pay for a cost if a change takes a longer time. Janus can plan for such cases. We introduce three types of cost for delayed changes: (1) *Constant cost (labeled as Constant):* Each step of the plan has a fixed cost (4 units). For example, applying a change may require a fixed amount of engineering effort in each step. (2) *Increasing cost (labeled as Increasing):* We use a linear cost function where the n_{th} step of the plan costs n units. This happens if, for example, we need to fix critical bugs quickly and the longer we wait, the more network remains vulnerable (i.e., more cost to operators). (3) *Cost after a deadline (labeled as Deadline):* We model this as a fixed cost of 30 units after the 6th step. This happens, for example, when an engineer relays the rest of the change to another engineer at the end of his shift (and increases the risk of making errors). (4) The *Default* bar is the original function with only customer impact cost. We minimize the total expected cost of customer impact and delayed changes.

Fig. 10 shows that Janus online only takes 11%-47% of the MRC cost; similarly, Janus offline has 24%-55% of the MRC cost. Janus adjusts the plan based on the cost function. However, MRC can only use a fixed-step plan (e.g., 8 steps in this case) independent of the cost function.

For *Constant* and *Increasing*, Janus selects a shorter plan (on average 2.82 and 2.84 steps) to reduce the cost of delayed changes at the expense of increasing the customer impact cost (from 8.6 in *default* to 12.4 and 11.96). In this way, Janus identifies the best tradeoff between the two types of cost. For *Deadline*, because there is a significant cost beyond 6 steps, Janus fits the plan within 6 steps to reduce the overall cost with the expense of slightly increasing the customer impact cost (from 8.6 to 9).

6 Related Work

Scheduling network updates. A few prior efforts focus on planning network updates (i.e., forwarding plane changes). Reitblatt et al. [36] introduce consistent switch rule updates to avoid loops or black-holes. zUpdate [29] plans traffic migrations (caused by network updates) with no packet loss during the worst-case traffic matrices. SWAN [20] and Dionysus [25] schedule forwarding plane updates for WAN by breaking the updates into stages with barriers in-between. While these low-level tools are useful in updating individual switch configurations, Janus plans upgrades for a (large) group of switches or links in data centers. Moreover, Janus adjusts plans based on traffic changes in real-time.

Failure mitigation. Autopilot [21] manages end-host updates and remedies failures at the end-hosts through reimaging or rebooting. Bodik et al. [11] discuss an optimization framework for increasing the resiliency of end-host applications to faults. Janus deals with the general problem of network upgrades and can provide scheduling support for these failure mitigation solutions.

Network symmetry. Beckett et al. [8] compress the control plane of large networks to test data plane properties, e.g., reachability and loop freedom. Plotkin et al. [35] scale up network verification for reachability properties by using symmetry. It is unclear how such techniques apply to network change planning under traffic dynamics. Janus builds a compressed data-plane to speed up simulations and uses subplan equivalence to prune the plan search space.

7 Conclusion

Fast network changes are critical for enabling quick evolutions of data centers today. Janus applies network changes by estimating the impact of various plans and dynamically adjusting the plans based on traffic variation and failures. Janus uses network automorphism to scale to a large number of plans. Janus plans in real-time even for the largest of data-centers and finishes upgrades with 33% to 71% of the cost of MRC planners.

8 Acknowledgments

We like to thank our shepherd, Rebecca Isaacs, for her extensive comments. Special thanks to Behnaz Arzani and Jeff Mogul for significantly improving the quality of this paper through their comments. Additional thanks to Ran Ben Bassat, Irene Zhang, Dina Papagiannaki, Srikanth Kandula, and the SOSP reviewers for their feedback on earlier drafts of this paper. This research was supported by CNS-1834263, CNS-1413978, and Facebook Fellowship program.

References

[1] 2012. Arista Warrior. http://bit.ly/2DxBYoI. (2012).

[2] 2012. Why The Drain in the Bathtub Curve Matters. (2012). http://bit.ly/2W9PWWI

[3] 2016. Software for Open Netwroking in the Cloud. https://bit.ly/2WbFspS. (2016).

[4] 2018. Amazon Compute Service Level Agreement. https://amzn.to/2NMCHuJ. (2018).

[5] 2018. Azure Service Level Agreements. https://bit.ly/1TnZwOn. (2018).

[6] 2019. Enterprise Cloud Computing on AWS. (2019). https://aws.amazon.com/enterprise/

[7] 2019. Large Installation System Administration Conference. (2019). https://www.usenix.org/conferences/byname/5

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control Plane Compression *(SIGCOMM)*.

[9] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. 2011. MicroTE: Fine Grained Traffic Engineering for Data Centers *(CoNEXT '11)*.

[10] B. Beyer, C. Jones, J. Petoff, and N.R. Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems*. http://bit.ly/2GBiuSa

[11] Peter Bodik, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A. Maltz, and Ion Stoica. 2012. Surviving Failures in Bandwidth-constrained Datacenters *(SIGCOMM)*.

[12] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. 2018. FBOSS: Building Switch Software at Scale. In *SIGCOMM*.

[13] Chuanxiong Guo and Guohan Lu and Dan Li and Haitao Wu and Xuan Zhang and Yunfeng Shi and Chen Tian and Yongguang Zhang and Songwu Lu and Guohan Lv. 2009. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers *(SIGCOMM)*.

[14] Emilie Danna, Subhasree Mandal, and Arjun Singh. 2012. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering *(INFOCOM)*.

[15] Ennan Zhai and Ruichuan Chen and David Isaac Wolinsky and Bryan Ford. 2014. Heading Off Correlated Failures through Independence-as-a-Service *(OSDI)*.

[16] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications *(SIGCOMM)*.

[17] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *SIGCOMM*.

[18] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. 2008. The Cost of a Cloud: Research Problems in Data Center Networks. *SIGCOMM Computer Communication Review* (2008).

[19] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, , Parantap Lahiri, Dave Maltz, and and. 2009. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*.

[20] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-driven WAN *(SIGCOMM)*.

[21] Michael Isard. 2007. Autopilot: Automatic Data Center Management *(SIGOPS)*.

[22] Navendu Jain and Rahul Potharaju. 2013. When the Network Crumbles: An Empirical Study of Cloud Network Failures and their Impact on Services. In *SOCC*.

[23] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally-deployed Software Defined Wan *(SIGCOMM)*.

[24] Amin Vahdat Jim Wanderer. 2018. Google Cloud using P4Runtime to build smart networks. http://bit.ly/2HG2jG4. (2018).

[25] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. 2014. Dynamic Scheduling of Network Updates *(SIGCOMM)*.

[26] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. 2010. Onix: A distributed control platform for large-scale production networks.. In *OSDI*.

[27] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. 2015. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing *(SIGCOMM '15)*.

[28] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. 2014. Traffic Engineering with Forward Fault Correction *(SIGCOMM)*.

[29] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. 2013. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM*.

[30] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. 2013. F10: A Fault-tolerant Engineered Network *(NSDI)*.

[31] Eugene M Luks. 1982. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of computer and system sciences* (1982).

[32] Brendan D. McKay and Adolfo Piperno. 2014. Practical graph isomorphism, {II}. *Journal of Symbolic Computation* (2014).

[33] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. 2013. Scalable Rule Management for Data Centers *(NSDI)*.

[34] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz AndrÃĬ Barroso. 2007. Failure Trends in a Large Disk Drive Population. In *FAST*.

[35] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. 2016. Scaling Network Verification Using Symmetry and Surgery *(POPL '16)*.

[36] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. 2012. Abstractions for Network Update *(SIGCOMM)*.

[37] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *SIGCOMM*.

[38] Brandon Schlinker, Radhika Niranjan Mysore, Sean Smith, Jeffrey C. Mogul, Amin Vahdat, Minlan Yu, Ethan Katz-Bassett, and Michael Rubin. 2015. Condor: Better Topologies Through Declarative Design *(SIGCOMM)*.

[39] Simon Kassing and Asaf Valadarsky and Gal Shahaf and Michael Schapira and Ankit Singla. 2017. Beyond Fat-Trees without Antennae, Mirrors, and Disco-Balls *(SIGCOMM)*.

[40] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network *(SIGCOMM)*.

[41] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. 2012. Jellyfish: Networking Data Centers Randomly *(NSDI'12)*.

[42] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *EuroSys*.

[43] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. 2009. Sandpiper: Black-box and Gray-box Resource Management for Virtual Machines. *Comput. Netw.: The International Journal of Computer and Telecommunications Networking* (2009).

[44] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. 2012. NetPilot: Automating Datacenter Network Failure Mitigation *(SIGCOMM '12)*.

[45] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. 2017. Understanding and Mitigating Packet Corruption in Data Center Networks. In *SIGCOMM*.

A Proof of theorem 3.1

We define a model to capture the states for a flow level simulation of the network. These states together with an operation semantics specify flows traverse the network. However, we do not care how such semantics are defined. Rather, we focus on encoding the network state in a precise manner. This allows us to define different operational semantics on top of the network state for various purposes. For example, the semantics could use proportional fairness or max-min fairness as we did.

Definition *Network*: We define a network as a tuple (G, R, S) where:

P1) $G = (V, E)$ is a graph specifying the network topology. V is the set of nodes and $E : V \times V \rightarrow \mathbb{R}$ is a function specifying which nodes are connected together and what is the capacity of the edge.

P2) R is a function assigning rules to nodes:

$$R : V \rightarrow \mathbb{R}^* \text{ where } \mathbb{R} = \{(src, dst, t, action) \mid in, out \in V, \\ t \text{ is packet specific test condition}\}$$

We refer to the i_{th} rule as $R_{v,i}$; t describes packet testing conditions not captured in the form of source or destination nodes, e.g., protocol or port; and *action* is one of **drop** or **fwd** P where $P \subset (V \times \mathbb{R})$. P specifies the portion of traffic that goes through a specific port.

P3) S is a partial function specifying the traffic sent from the end-hosts: $S : V \times V \rightharpoonup \mathbb{T}$. Where u actively generates traffic towards v, $S(u, v)$ describes that traffic in terms of a model-specific encoding $\mathbb{T}$.

Definition *Network Isomorphism*: We say two networks are equivalent up to isomorphism if there is a vertex renaming function (bijection) that permutes the nodes between the two networks while preserving the G, R, S relations. More concretely, two networks, $N \simeq N'$ are isomorphic if $\exists \pi_V : V \longleftrightarrow V'$ where $G \simeq_\pi G', R \simeq_\pi R', S \simeq_\pi S'$. For a renaming function π_V:

1) G and G' are isomorphic when:

$$E(v1, v2) = E'(\pi_V(v1), \pi_V(v2))$$

2) R and R' are isomorphic when:

$$R_i = (v, t) \Leftrightarrow R'_i = (\pi_V(v), \pi_T(t)) \text{where:}$$
$$\pi_T(t) = (\pi_V(v1), \pi_V(v2), t, \pi_A(a))$$
$$\pi_A(a) = \begin{cases} \textbf{drop}, & \text{if } a = \textbf{drop} \\ \textbf{fwd } \{\pi_V(v) | \forall v \in \text{ports}\}, & \text{if } a = \textbf{fwd ports} \end{cases}$$

3) S and S' are isomorphic when: $\forall u, v \in V : S(u, v) = \pi_T(S'(\pi_V(u), \pi_V(v)))$ where π_T permutes the nodes encoded in the traffic using π_V.

Definition *Isomorphic network function*: A network-isomorphic-invariant function $F : N \rightarrow T$ is a function that does not use identifying information for the nodes. That is, F is invariant under network isomorphisms if $N \simeq N' \Rightarrow F(N) = F(N')$ for all networks N, N'.

Theorem A.1: A network-isomorphic-invariant function, F, outputs the same value for two isomorphic networks, N, N', that is: $F(N) = F(N')$.

Proof: The proof is given by the definition of F.

Theorem A.2: Max-min fairness is agnostic under network isomorphism.

Proof: Max-min fairness is solving the following equation:

$$\begin{array}{lll} \text{maximize} & \sum_i U(x_i) & \\ \text{s.t.} & \sum_i R_{li} x_i \leqslant c_l & \text{variables} x_i \geqslant 0 \end{array}$$

Where, x_i is the rate allocation between two nodes. c_l is the capacity of the link l. And R_{li} is the routing on the links. R_{li} is one when flow i goes through link l and zero otherwise.

By using P1 and P2, we guarantee that the set of equations that we write for max-min fairness are the same between the two networks. We know that the output of max-min fairness is unique. Therefore, an arbitrary renaming of the variables names does not impact the optimization result. Therefore, since the two sets of equations between the two networks are only different in the name of the variables and since the result is unique, we can conclude that max-min fairness is network-isomorphic invariant.

B Network Automorphism

Theorem B.1 (Network Automorphism)**.** For a subplan, s, and a renaming function, f, that maps network N onto itself, if f preserves properties P1, P2, and P3, the two subplans s and $f \cdot s$ (the subplan after applying the renaming function to its elements) are equivalent.

Proof: To prove this it is enough to show that a renaming function between the two networks exist. We prove this in two parts: First, we prove that $(f \cdot N)/s$ is equivalent to $N/(f \cdot s)$. Second, We then prove that $(f \cdot N)/s$ is equivalent to N/s. Finally, we conclude that $N/(f \cdot s) \equiv N/s$.

To prove the first part, we use the identity function as the renaming function. First, it is easy to verify that the two graphs are isomorphic, that is, for every switch $A \in (f \cdot N)/s$ there exists a switch with the same name $A \in N/(f \cdot s)$. Similarly, for every link, between two switches in one network, we can find a similar link in the other network. Therefore, P1 is true.

For P2, since a subplan does not impact traffic sources, we know that for every traffic source in one graph, there exists a traffic source in the other graph. And therefore, the traffic between the two has not changes.

P3 is true given that we have proved P1 and P2.

Cost function	Formula	Cost at 99.95%	99.90%	99.75%
log	$C(100 \ln(637x + 1))$	20	40	90
linear	$C(50000x))$	20	50	100
quad	$C((4200x)^2))$	0	10	100
exp	$C(100(e^{277x} - 1))$	10	30	100

Table 5. Cost functions for purely mathematical functions.

To prove the second part, we already know by definition that $f \cdot N$ and N are equivalent, therefore, we can replace N in place of $f \cdot N$.

C Cost functions

Cost functions are shown in Table 5. C is a clamping function that bounds the output of each function between 0 and 100 and packs the values in bins of size 10:

$$C = \max(0, \min(100, \lfloor 10\frac{x}{10} \rfloor))$$