Change Management in Physical Network Lifecycle Automation

Mohammad Al-Fares, Virginia Beauregard, Kevin Grant, Angus Griffith, Jahangir Hasan, Chen Huang, Quan Leng, Jiayao Li, and Alexander Lin, *Google;* Zhuotao Liu, *Tsinghua University;* Ahmed Mansy, *Google;* Bill Martinusen, *Formerly at Google;* Nikil Mehta, Jeffrey C. Mogul, Andrew Narver, and Anshul Nigham, *Google;* Melanie Obenberger, *Formerly at Google;* Sean Smith, *Databricks;* Kurt Steinkraus, Sheng Sun, Edward Thiele, and Amin Vahdat, *Google*

https://www.usenix.org/conference/atc23/presentation/al-fares

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Change Management in Physical Network Lifecycle Automation

Mohammad Al-Fares[*], Virginia Beauregard[*], Kevin Grant[*], Angus Griffith[*], Jahangir Hasan[*], Chen Huang[*], Quan Leng[*], Jiayao Li[*], Alexander Lin[*], Zhuotao Liu[‡], Ahmed Mansy[*], Bill Martinusen[†], Nikil Mehta[*], Jeffrey C. Mogul[*], Andrew Narver[*], Anshul Nigham[*], Melanie Obenberger[†], Sean Smith[§], Kurt Steinkraus[*], Sheng Sun[*], Edward Thiele[*], and Amin Vahdat[*]

[*]Google
[†]Formerly at Google
[‡]Tsinghua University
[§]Databricks

Abstract

Automated management of a physical network's lifecycle is critical for large networks. At Google, we manage network design, construction, evolution, and management via multiple automated systems. In our experience, one of the primary challenges is to reliably and efficiently manage change in this domain – additions of new hardware and connectivity, planning and sequencing of topology mutations, introduction of new architectures, new software systems and fixes to old ones, etc.

We especially have learned the importance of supporting multiple kinds of change in parallel without conflicts or mistakes (which cause outages) while also maintaining parallelism between different teams and between different processes. We now know that this requires automated support.

This paper describes some of our network lifecycle goals, the automation we have developed to meet those goals, and the change-management challenges we encountered. We then discuss in detail our approaches to several specific kinds of change management: (1) managing conflicts between multiple operations on the same network; (2) managing conflicts between operations spanning the boundaries between networks; (3) managing representational changes in the models that drive our automated systems. These approaches combine both novel software systems and software-engineering practices.

1 Introduction

In large production networks, changes happen all the time. Lots of research and development has delivered a wide range of designs and products for managing changes to network data planes and control planes. Requirements for scalability, reliability, security, low cost, and rapid flexibility together have made it essential to automate many aspects of network management, but this work has mostly focused on managing networks *after* the physical components have been deployed. For example, Software Defined Networking (SDN) methods do not directly address designing the physical wiring of a network, or ensuring that the right switches and cables are ordered from vendors, or connected to effect a design, or how to sequence and schedule this physical work.

At Google, we have also found it necessary to automate many *abstract and physical aspects of a network's full lifecycle*, including network planning (what networks do we need to build and when, given capacity forecasts?), network design (what specific switches and links do we need?), materials ordering (what specific part numbers do we need to order and when, what cables need to be constructed?), network construction (where do data center operators need to place equipment and cables?), firmware installation, physical validation (are the links correctly connected and not suffering high error rates?), network repair processes (which links/switches can we safely drain before doing repairs?), etc. We must also provide our automated control planes with accurate, detailed "schematics" for the networks that they manage.

While initially we could perform *Network Lifecycle Management* (NLM) manually, in practice this was slow, error-prone, and inflexible. Those problems worsened with increasing scale, driving us to automate as much of this work as possible. For example, designing optimal inter-block cabling for a Jupiter fat-tree network is NP-complete, and a good approximation requires significant computation [31, 38]. Even at much smaller scales, processes like correctly rolling out router configuration changes are safest when they are carefully automated [23].

As we introduced systems to automate NLM, we discovered that we had not sufficiently understood or appreciated the difficulty of *change management* in this specific domain. Certainly, change management has always been a problem for network designers and operators, and much useful work has been done on ways to manage changes to device configurations (or SDN controller configurations) related to routing, access control lists, and other post-deployment issues.

However, several other aspects of change management began to delay our progress and undermine the value of our automation. These include managing conflicts between multiple operations on the same network; managing conflicts between operations spanning the boundaries between networks; managing representational changes in the models that drive our automated systems; and introducing major changes in our software infrastructure.

In this paper, we focus on these change-management challenges, the solutions we developed for them, and some of the experience we gained. In particular, we address several distinct (but interacting) aspects of change management:

Managing conflicts between operations: deciding what order to do things in, and what process steps can be done in parallel without conflicts. The physical lifecycle of a network involves many steps with complex dependencies. In small networks, changes are *sometimes* sufficiently rare and rapid that they can be serialized without loss of efficiency. In a large, frequently-changing network, multiple changes, sometimes with extended execution time, *must* overlap, or else capacity delivery and upgrades becomes unacceptably sluggish (e.g., see §10.1). We must prevent operational conflicts that lead to outages, or even the risk of outages, because we want to do these operations on "live" networks.

Planners must also be able to analyze potential sequences of future changes to decide the best partial order, choose the least costly option, or detect if a sequence would lead to an infeasible or invalid state (we call this "what-if analysis"). Planners also sometimes need to *modify* the order of existing plans, as constraints or requirements change.

Therefore, a key contribution of this paper is the design of a plan-management service (§5.2), and the abstractions that allow us to explicitly represent how various future lifecycle states depend on each other or can be done in parallel.

Representational change: Automation depends on machine-readable data. Foundational to the work in this paper is the MALT network-model representation [27], which we use to represent the current, desired, and potential future states of network topologies at many levels of abstraction. Planning and design processes form a pipeline of successive refinements of these models, and the generation of derived data, such as instructions for creation and placement of cables, from these models. Likewise, operational data, such as device and SDN controller configurations, are primarily derived from these models.

A notable challenge in modeling is that our continued innovation in network designs and components requires us to rapidly evolve the MALT representation (e.g., Fig. 8). Previously, we found it hard to do this safely (without production outages) and without constantly and tediously updating lots of model-generating and model-consuming software; this seriously slowed our innovation.

We describe how we allow a wide variety of model-consuming systems, built and maintained by many different partner teams, to cope with evolution in our MALT representation and how we use it to encode specific designs, without requiring unsustainable engineering efforts on the part of those teams. (§7, §8)

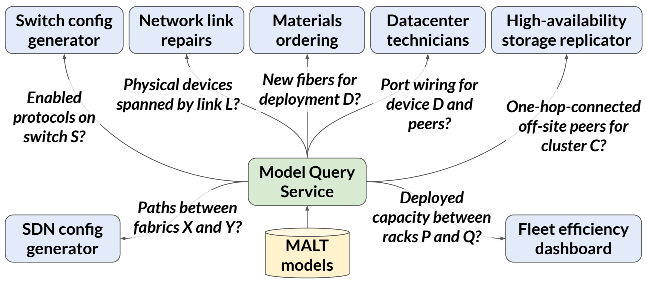

Switch config generator
Network link repairs
Materials ordering
Datacenter technicians
High-availability storage replicator
Enabled protocols on switch S?
Physical devices spanned by link L?
New fibers for deployment D?
Port wiring for device D and peers?
One-hop-connected off-site peers for cluster C?
Model Query Service
Paths between fabrics X and Y?
Deployed capacity between racks P and Q?
SDN config generator
MALT models
Fleet efficiency dashboard

Figure 1: Some use cases for MALT models.

While this paper reports on our experience with large-scale datacenter network infrastructures, we are also applying the same tools and practices in several adjacent domains, such as the management of WAN systems, of machines, and of datacenter physical designs. Our approaches are useful at smaller scales, too.

Ethical concerns: The systems described in this paper do not handle or store any Personal Identifiable Information (PII), and do not raise any ethical issues.

2 Context

This paper focuses on our *Jupiter* datacenter network fabrics [30, 31], and our *B4* WAN fabrics [16, 18]. By "fabric" in this paper, we generally mean a Clos network consisting of many switches and links, with SDN controllers.[1]

We drive our network automation using machine-readable representations, primarily the MALT representation (*Multi-Abstraction Layer Topology*) representation, described in [27], and summarized below in §2.1. We organize these representations as *models* of the network fabric's topology and many other static details.

Our automation covers many aspects of a network's entire lifecycle, including (as depicted in Fig. 1 and Fig. 2):

- **Fabric planning**: Our infrastructure planning team decides when we need to build, expand, or decommission a fabric. They express their high-level goals (where should the fabric be built? Using what abstract architecture? Of what size?) via MALT *fabric intent* models. These models have relatively few details. Planners might explore several possible options for a fabric before committing to a specific choice.

- **Fabric design**: Before we can construct a network, we convert the fabric intent into *concrete* MALT models, via a fully-automated process somewhat like compilation of a program. The concrete models, after several levels of refinement, fully define the structure of the network, down to individual switches, fibers, racks, etc.

[1]We have other networks that use a more traditional design consisting of large non-SDN routers; our management processes for those networks are somewhat different.

- **Materials ordering**: When we commit to a decision to build or expand a fabric, we take the first (nearly) irreversible step of ordering materials (switches, fibers, etc.). Since these are expensive, after this point we generally avoid changing our intentions. The choice of exactly what materials to order is an automated process driven by MALT models and additional inputs.
- **Fabric deployment**: With the materials on hand, we can build the network (place racks, switches, and SDN controller machines, connect them with fibers, and test that it all works). This step is carried out by humans, using automatically-generated detailed human-readable plans based on the MALT models. We also install the necessary on-switch software using automated processes.
- **Controller and switch configuration**: Once the network is built, we automatically generate SDN controller and switch configurations from the MALT models, with additional configuration inputs.
- **Operations**: Our operators manage the run-time behavior of a network via various tools and services. For example, they might need to "drain" part of a fabric to carry out repairs or upgrades, and then "undrain" it later. The systems that carry out these operational changes are also driven, in part, by MALT models. We also have automated control planes (e.g., [22]) that depend on structural information expressed using MALT.
- **Health and repairs**: We have automated systems to decide whether the network is unhealthy, to diagnose the causes of problems, and to help us understand what systems would be affected by a faulty (or drained) component. These all use MALT as some of their input data.

It should be obvious that we rely on *accurate* and *up-to-date* MALT models for virtually all of our network management automation. The challenges discussed in this paper are all related to preserving that accuracy when our systems are constantly evolving.

2.1 Background on MALT

We briefly summarize the aspects of MALT [27] necessary to understand this paper.

MALT is an *entity-relationship model* representation, not a database. In an entity-relationship model, entities represent "things," which can be abstract (e.g., an entire fat-tree network) or quite detailed (e.g., a specific strand of fiber), or in-between (e.g., an IP adjacency aggregating multiple fibers). Entities have "kinds" (types), names, and attributes, Entities are connected via relationships, which have kinds, but neither names nor attributes.

A collection of entities and relationships forms a MALT *model*. We divide models into *shards*, with some shard-specific metadata. Model shards are typically (but not always) aligned with physical infrastructure boundaries at the city, region, or building scale. Some shards have millions of entities. Please refer to [27, Fig 3.] and [27, Appendix A] for example MALT models. Detailed, machine-readable versions of these examples are available for download [13].

We normally store shards in MALTShop, a purpose-built system that enables easy sharing of models between systems, which depends on naming, access control, and consistency. Shards in MALTShop have names (similar to UNIX pathnames) and access-control lists (ACLs). Every update to a shard creates a new, immutable version, with an immutable version number. MALTShop uses a copy-on-write mechanism to efficiently store many versions of a shard that is being incrementally updated.

MALTShop supports a generic query language, which walks an entity-relationship graph to extract a chosen subset model. It allows one query to span a set of multiple shards.

To manage evolution, each shard can assert compliance with one or more *profiles*. A profile is essentially a contract between a shard's producer and consumers that the shard conforms to a set of predicates. Profiles are versioned; when we need to change how we represent a network, we signal that by creating a new version of the corresponding profile.

Our planners, designers, and operators usually want to think in terms of the high-level abstract designs of these networks (e.g., a Jupiter network is a collection of blocks connected by spine blocks), rather than in terms of specific switches and fibers; the support for multiple abstraction layers in a single MALT model enables this separation of concerns.

2.2 Why automate?

Our work was motivated by our large, heterogeneous networks, but we believe this kind of automated approach would be valuable for a broad range of network operators (although we realize that the market for full-lifecycle automation software may be small).

Automation enables design flexibility and experimentation. The research community has generated a wonderful range of scalable network structures, including Fat-Trees [1], expander graphs [32], F10 [25], etc. These designs exploit path redundancy to support high bandwidth and availability at relatively low cost.

However, most non-hyperscaler enterprises appear not to be using modern multipath networks. In fact, the dominant provider of network hardware recommends a simple three-layer design with large "core" switches at the top, relatively large "aggregation" switches in the middle, and smaller "access" switches (e.g., top-of-rack or ToR switches) at the bottom [8] and other vendors recommend two layers [3].[2] Why do most enterprises avoid multipath networks? Our (admittedly anecdotal) understanding from several experts is that

[2]These are old citations; the age of these documents may reflect an inherent stasis.

most enterprises lack the design tools that they would need to construct and maintain fat-tree networks, let alone less-regular designs such as expander graphs. We speculate that the low adoption rate for research-generated network designs is at least partially due to a lack of tooling, especially to support safe and frequent changes. (There are, of course, other reasons.)

Risk management. Hyperscalers use multipath network topologies [2, 31, 34, 38] also because they support incremental expansion of, and upgrades to, live networks. The need for zero-downtime changes to the structure of these networks is driven by Service Level Objectives (SLOs) targeting 99.99% or better availability, which allows at most a minute of downtime per week. Some providers hope to achieve 99.999% availability, allowing just 5 minutes of downtime per year.

Most enterprise networks are more static than ours; every change creates the risk of a large-scale outage, so operators are extremely change-averse. Their use of non-multipath designs leads to less structural redundancy, which we and other hyperscalers exploit to allow frequent changes at relatively low risk. Our automated tooling both indirectly supports low-risk changes by enabling the use of fat-trees, and directly supports it by allowing us to validate all low-level changes against higher-level intent. Smaller enterprises would benefit from using such automation to mitigate change-management risks; e.g., assigning the same IP address to two different endpoints (a mistake we have made in manual workflows).

Addressable markets. Our work focuses on automated operations on large and frequently-changing networks. Many networks are too small or static to require such automation; how many enterprises actually have large networks? Data on this topic is difficult to find, often because it is only available via high-cost market-research reports. While the number of hyperscalers is small, when we include software-as-a-service and other forms of cloud, there are at least dozens of such providers [29]. The number of hyperscaler data centers is also growing consistently [9]. However, there are many other large non-hyperscaler data centers. There are also thousands of smaller "Points of Presence" (POPs); a typical pattern for both large and small enterprises is to build and maintain small fabrics in many POPs, motivating *frequent* use of design and turnup automation.

Takeaway: Many – perhaps most – network outages result from human error, often associated with physical-network changes [15, 21]; automation with a specific focus on change management can make these changes faster and more reliable. It also enables increased agility and innovation.

2.3 Related work

Prior work on network management has mostly focused on network device configuration management, such as configuration language design [7] and configuration generation for existing networks [5, 24, 34]. Although our networks' configurations are derived from network models, our focus here is on planning for topology designs, and for the physical construction, modification, and eventual decommissioning of networks, as well as for their day-to-day operation.

Most prior work (both academic and commercial) has also typically focused on managing the network as it is now, or on verifying near-term intent (i.e., to be implemented as soon as possible). This includes verification of data-plane [19, 20, 36] and control-plane [4] properties.

In contrast, this paper addresses the challenges of planning for future physical states of a network, especially on how to manage sequences of dependent changes in the face of confounding factors, and on how to validate both individual states and sequences of changes. We note that validated, accurate topology models can enable verification of control-plane and data-plane layers.

Some prior work (e.g., [32, 37]) discussed how topology design affects the complexity of lifecycle management, but did not address how to automate the management processes.

Network operators often expand network topologies to augment capacity [37]. Prior work [38] described how we expand live data center networks, through a layer of patch panels; it uses an integer linear programming algorithm to minimize the number of wires to move, while also maintaining sufficient bandwidth through multiple stages (so as to avoid packet loss). In this paper, we address how those multiple stages are planned and coordinated.

3 Change management challenges

We start by describing some of the many specific challenges in change management.

3.1 Orchestration of physical changes

In traditional, non-automated network management, changes to the physical infrastructure of the network (e.g., adding/moving/removing a switch or link) are typically treated as risky operations. Network operators often limit these to maintenance windows, during which some or all of the network becomes unavailable – and because these windows are disruptive, they can only be scheduled rarely, and then must be carried out as quickly as possible. This approach supports neither rapid evolution nor high availability.

More modern, scalable network designs such as Jellyfish [32] and other expander graphs, Facebook's Fabric [2], and Jupiter [31] use "multipath" designs that exploit path redundancy to support high bandwidth and availability at relatively low cost. Multipath topologies also support incremental expansion of, and other upgrades to, live networks, because their redundancy allows "draining" parts of the network during these operations.

For example, the Jupiter architecture consists of several types of blocks, each of which is a Clos fabric. Some of these blocks ("pods") provide connectivity to racks of machines via Top-of-Rack (ToR) switches; some ("fabric border routers", or FBRs) provide connectivity to WANs and other Jupiters;

some provide connectivity between other blocks in the same Jupiter.[3] This modular architecture allows us to build a large fabric (dozens of blocks) incrementally, rather than paying the capital costs and energy of building an entire fabric before we need all of it. Modularity also allows us to add blocks built from newer (faster/cheaper) switches and links to an existing fabric. In order to manage the evolving connectivity between blocks without having to completely rewire everything, we use a layer of patch panels [38], but even so, adding or removing a block requires significant human effort to reconfigure the patch panels.[4]

When we first started automating operations on Jupiter networks, we ran into the issue that each change to a fabric typically depends on the previous changes. However, our original topology-model representation only allowed us to represent one "intended" view of the network, so we had to serialize changes to a fabric: generate an intended model for one change, then carry out that change, and only then could we generate a model for the next change; the resulting delays become nearly intolerable. Our use of MALT solved that problem, because we could create multiple independent versions of a fabric's model in advance.

However, this still left us with the problem of managing the dependencies between a sequence of models – for example, ensuring that two different changes did not use the same switch port for conflicting purposes. Our solution to this kind of conflict avoidance serializes changes by means of a fabric-level lock, essentially a mutex.

Takeaway: To help us discover and avoid such conflicts as quickly as possible, we have now formalized the relationships between multiple plans for a given fabric using the concept of *PlanPoints*, described in §5, and managed by a service, *TopoPlan* (§5.2).

3.2 Representational change

Our network management systems are highly automated and thus heavily dependent on machine-readable data. This data is primarily represented in MALT, on which we focus in this paper, but we use several other standardized representations, such as OpenConfig [28] for telemetry. These representations must evolve over time, to support novel network designs, new hardware, new management concepts (e.g., failure-independent "zones" for high availability), etc.

For example, initially we did not model connectivity between machines and top-of-rack (ToR) switches, so we did not model machines. However, newer policies for machine-specific security and rate-limiting required authoritative intent for these connections, so we added machines to MALT models.

[3]The original Jupiter design incorporated "spine blocks" to form a folded Clos connecting the pods and FBRs. More recently, we connect those blocks directly without using spine blocks, but sometimes the pods themselves provide transit routing between other blocks [30].

[4]Our more recent deployments replace patch panels with optical circuit switches, which avoid much but not all of the human effort for reconfiguration [30].

Table 1: Acronyms and terms used in this paper.

Acronym/ term	Definition
DCNI	Data Center Network Interconnect
MALT	Multi-Abstraction-Layer Topology representation
MALTShop	A storage system for MALT
MSID	Model-Set ID
MBS	Model-building service
MDS	User-facing design service
MQS	Model-query service
NPI	New-product introduction
Block	Modular unit of fabric design
TopoPlan	Change-management service
UIM	Unified Intent Model
PP	Patch panel
PoR	Plan-of-Record

Consequently, a consumer querying a model for "all devices connected to ToR T" now receives not just the connected fabric switches, but also the connected machine entities. In practice, we've found that the complexity and level of detail in our models tends to increase over time.

While we attempt to make most representational changes backward-compatible, this is not always possible; sometimes our best guesses about what matters are wrong. We have learned that seemingly-innocuous changes lead to outages, because of the many clever ways in which programmers accidentally build fragile assumptions into their code.

Since our overall system-of-systems cannot have any significant (multi-minute) downtime, when we need to introduce a new MALT profile (see §2.1) to signal a representational change, we cannot insist that all producers and consumers cut over at the same instant. There are many such systems (especially model consumers) with their own release cycles and constraints on engineering resources. Beyond that, any such change is risky; we would not even want to switch to a new profile without carefully-monitored "canaried" rollouts.

Takeaway: For these velocity and safety reasons, we found it necessary to decouple profile feature *introduction* from model-reader *adoption* of such features. The previous paper on MALT [27] briefly discussed our approach to profile evolution. In §7 and §8.2 we expand that discussion, showing how we use a layer of abstraction to decouple many consumers from the details of profiles.

4 Model-generation systems overview

To help readers understand the implications of managing conflicts between, and changes to, our network plans, we first describe how we generate detailed network models. Appendices §A and §B describe model generation in more detail.

Fig. 2 summarizes the model-generation process, and Table 1 summarizes acronyms and terms used in this paper. First, a model writer (e.g., the network planner) initiates a model change by sending an RPC request ① to a "design service," MDS. MDS insulates humans and external systems from a need to understand the details of UIM or TopoPlan (§5.2).

MDS is responsible for (i) translating an imperative user-level request (e.g., "add a new block to a Jupiter fabric") into a sequence of declarative high-level intent changes, and (ii) managing request concurrency (for instance, we preclude certain types of requests from simultaneously modifying the same fabric, and thus sequence those requests with a lock). MDS also collects additional information (e.g., the available IP prefixes) necessary to create the UIM changes.

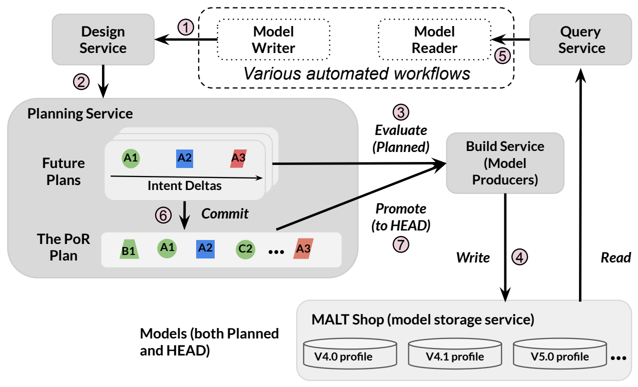


Figure 2: Model-generation systems.

We express model-generation intent in a *Unified Intent Model* (UIM), a form of MALT that abstractly represents the high-level graph of the global network at a given point in time. MDS represents intent changes as deltas to UIM.

This use of abstraction is an improvement on our prior system, in which all network changes consisted of precise orderings of low-level imperative mutations. Those were framed in terms of low-level details, such as the exact type and number of switches and their link-level connectivity; that approach bound decisions too early, making it hard to re-order a sequence of plans. Abstraction helps avoid this early binding.

Once MDS has mapped a request to a sequence of intent changes, it conveys ② this sequence to the TopoPlan plan management service (§5.2). TopoPlan supports parallelism between high-level requests by allowing interleaving of intent changes when they do not conflict. TopoPlan also supports "what-if" analysis, by maintaining multiple (sometimes thousands of) branches of possible future states.

Whenever we need to build concrete MALT models from a UIM plan (e.g., for physical installation), TopoPlan invokes ③ MBS, a "build service" that compiles the high-level intent to MALT models, which are stored ④ in MALTShop. Fig. 2 shows that we simultaneously generate semantically-equivalent concrete models in multiple profiles (see §2.1) – e.g., "V4.0," "V4.1," "V5.0" – to support profile evolution (see §7). For more details on MBS, and many more details on the model-generation process, including examples of UIM, see §A and subsequent appendices.

Models represent the intended network, so mismatches against actual state represent deployment errors (e.g., miscabling, etc.), and we correct reality to match the plans. We use various mechanisms to detect these mismatches, such as neighbor discovery via LLDP (IEEE 802.1AB [17]).

Model readers can query ⑤ these generated models via MQS, a semantic Query Service that also helps support evolution (see §8).

A single high-level operation may invoke these processes multiple times over the course of weeks or months. For instance, when we expand a live Jupiter fabric, we need to do this in several stages, to ensure the network always has sufficient residual capacity during each expansion step. This means we need to generate MALT models for each intermediate stage.

Network model mutations are not real-time. We have safety checkers to block planned changes if they would violate consistency checks, or capacity thresholds designed to leave room for switch or link failures. Systems with real-time goals, such as our SDN controller [11], maintain internal representations of network links, initialized from MALT models.

5 Physical-change plan management

To illustrate the problems of managing concurrent future changes, consider a simple example network with two switches A and B connected via four links (Fig. 3(a)). This network is currently carrying live traffic; hence we call the corresponding MALT representation of the network the "live" model. We would like to expand the network capacity by adding a third switch C (Fig. 3(b)). We call the MALT representation of this future network the "planned" model.

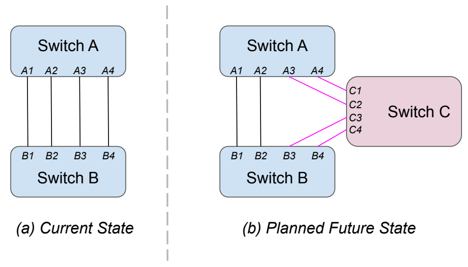


Figure 3: Models for current and planned network.

For several reasons, we want to generate a planned model well in advance. First, it allows us to accurately itemize the physical resources (switches, racks, fiber optic cabling, etc.) required to support switch C. Many of the resources have high costs and lead times: purchasing too much in advance wastes money, while purchasing too little or too late slows our ability to deliver network and compute capacity.

Second, modeling in advance allows us to simulate the future network, and validate it against reliability requirements. *E.g.*, if we must maintain $\geq 75\%$ of normal capacity during expansion, we must change the live network in stages, as shown in Fig. 4. We cannot directly switch between the initial and planned states, which would move two links from switch A (and B) simultaneously, causing 50% capacity loss.

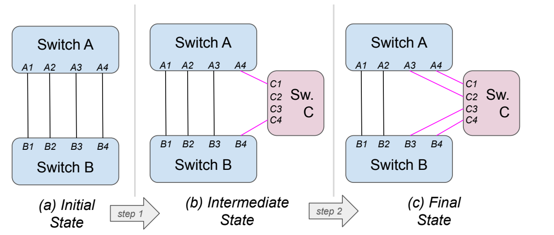


Figure 4: Two-stage migration from initial to final state.

We can also analyze or validate multiple options for a future plan, on metrics such as total cost of ownership (TCO).

5.1 Challenges with concurrent plans

Concurrent management of multiple plans creates several challenges: scaling issues, and (worse) the risks of incorrect planning decisions made because of stale models.

Scaling concurrent plans across a large and changing network: While one could use *ad hoc* methods to manage a set of concurrent plans, such as creating copies of future models in temporary storage, that quickly runs into scaling issues. Because our network is large and frequently changing, we have to manage a large set of concurrent plans. This creates two main problems: sequencing and validation.

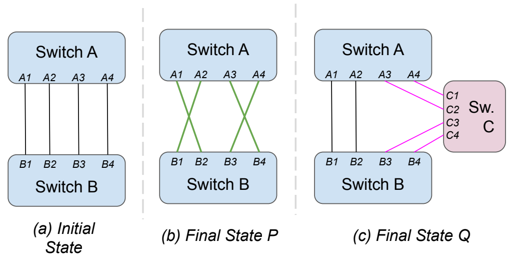


Figure 5: Two conflicting updates, P and Q.

For example, consider a planned change P that rewires the connectivity between switches A and B (Fig. 5(b)). This change conflicts with another planned change Q (Fig. 5(c)) to add switch C on ports (A3, A4, B3, B4), and thus we need to decide whether P or Q should come first.

However, different teams may be making the concurrent changes without coordinating, and the resultant sequence order can affect materials-ordering, instructions for technicians, etc. Worse, the live model is continually changing, and a fabric-level change that is applied to the "live" model (e.g., upgrading one or both switches) may invalidate the preconditions for P and Q, such as port reservations.

When each fabric has dozens of blocks and hundreds of switches, and must evolve rapidly to meet business needs (via expansions, upgrades, etc.), enforcing serialization on operations such as P and Q creates painful drag. While manual management of operational concurrency might be practical for just a few fabrics, an enterprise with dozens or hundreds of fabrics would find that unsustainable. Therefore, we need automation-friendly support: for tracking plans, and how they are ordered and sequenced; for rapid conflict-detection and plan-validation; and for speculative analysis of multiple future options (e.g., to cope with supply-chain issues).

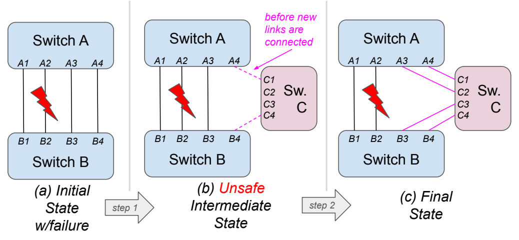

Switch A
A1 A2 A3 A4
B1 B2 B3 B4
Switch B
before new links are connected
Sw. C
C1 C2 C3 C4
(a) Initial State w/failure
step 1
(b) Unsafe Intermediate State
step 2
(c) Final State

Figure 6: Stale models might lead to an unsafe state.

Stale models: The live network can change unexpectedly, rendering plans stale, along with any derived actions we might have taken. For instance, if a link is removed for repairs (Fig. 6(a)), the initial and final states are both "safe" because they meet our 75% capacity threshold. However, the transition to the intermediate state (Fig. 6(b)) creates an unsafe state (only 50% capacity) while we are changing links. Unless we update and revalidate the models for all intermediate states, we would not detect this risk.

Advance planning can also lead to confusion if, during the long period between plan creation and implementation, constraints change – e.g., supply-chain issues force the use of different hardware.

Thus, given hundreds of potentially-conflicting changes that need to be sequenced, and validated for safety or TCO, manual management of multiple models, or managing placeholder elements in a single "live" model, quickly becomes intractable.

5.2 Plan management service: TopoPlan

To address the many challenges of plan management, we developed TopoPlan, analogous to a software-development version control system (VCS). A network change, specified as patches to high-level intent, may be directly applied to the live (i.e., HEAD) intent. However, the TopoPlan service also allows network changes to be sequenced in branches.

As in a VCS, changes within a branch can be added, removed, reordered, or merged, while branches can be rebased or merged. Branches can represent highly speculative changes (e.g., hypothetical "what-if" scenarios), authoritative changes (e.g., scenarios which we have financially committed to), or changes that are somewhere in between.[5]

Concrete MALT models can be compiled for any change in any branch, allowing us to analyze the network-capacity and TCO implications of any hypothetical future network state. TopoPlan also allows us to detect *conflicts* over limited resources – e.g., two plans trying to use the same switch port for different purposes.

[5]In contrast to traditional VCSes, which are geared toward human users, are text-based, and have change-rate and branching limits, TopoPlan was built with automation in mind, with arbitrary branching, and focuses on sequencing changes to high-level intent, using Protobuf-based intent-patches with special merge semantics [14].

We show concrete examples in §10 of how using TopoPlan greatly improves our deployement and operational efficiency through project pipelining, stacking planned changes, and enabling accurate material orders for future projects.

Plans and PlanPoints: TopoPlan maintains multiple branches of possible future state. A single future change to the network is represented by a *PlanPoint*, which consists of (i) deltas to the UIM, and (ii) a best-estimate timestamp at which the changes will be realized. A series of PlanPoints is called a *Plan*, which groups together a set of network changes into a timeline. A Plan consists specifically of (i) a sequence of PlanPoints, and (ii) a baseline snapshot of MALT models to which the PlanPoint deltas are to be applied. MALT models can be compiled for every PlanPoint in the Plan simply by starting with the UIM of the baseline, and, for each PlanPoint, applying its UIM deltas and compiling concrete MALT models: we term this process *evaluating* a Plan. The compiling process internally invokes the model generation service, as discussed in §4, and attaches the generated concrete models to the PlanPoint.

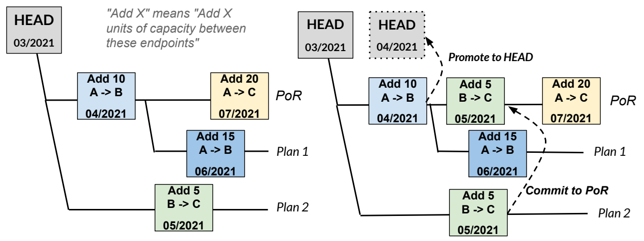


Figure 7: TopoPlan plans (left) and operations (right).

The *Plan-of-Record*, or PoR, is a canonical branch, which stores changes that we have committed to with a high degree of certainty (i.e., we have ordered materials that we really prefer not to waste). In the example of Fig. 7 (left), the PoR contains two PlanPoints that add capacity between B4 sites A -> B and A -> C. Plan 1 proposes increasing the capacity augment between A -> C from 10 to 15 units, while Plan 2 proposes an additional augment between B -> C. Note that, in this example, Plan 1 is "baselined" from a PlanPoint on the PoR, rather than from HEAD.

Committing and promoting plans: Changes on a non-PoR branch can be *committed* (Fig. 2 ⑥) to the PoR (Fig. 7 (right)) if the change does not conflict with other changes on the PoR, and when infrastructure planners sign off on the financial readiness of the change.

Only entire Plans can be committed to the PoR. When a Plan is committed, all its PlanPoints are validated against those on the PoR, and on success they are copied to the PoR. Conflicts are automatically detected by TopoPlan, but manually resolved by backing out, fixing a plan, and retrying. For example, in Fig. 7 (right), attempting the *commit* operation might reveal that we cannot add 5 B -> C units because we have already committed all free ports at C to the 20 A -> C units. We currently rely on human planners to make priority decisions outside TopoPlan, although automating some of these decisions is clearly worthy of future work.

Once the changes in a PlanPoint are ready to be realized in the physical network, the PlanPoint can be *promoted* (Fig. 2 ⑦) to HEAD (the intent representing the current desired state of the network). Promoting a PlanPoint applies its UIM change directly to HEAD, compiles HEAD to concrete MALT models, and removes the PlanPoint from the PoR.

Concrete MALT models can also be generated on demand for any PlanPoint on any branch (including PoR). These models can be used for what-if analyses, for example.

Changing plans: PlanPoints can be added, removed, edited, reordered by changing their timestamps, or merged by collapsing their UIM deltas. Plans can be created, deleted, or rebased by changing their baseline MALT snapshot. Both Plans and PlanPoints are versioned, and their version numbers are incremented on any of theses changes.

Every time HEAD is updated (100s of times per day), TopoPlan rebases the PoR to that new version of HEAD and does light-weight validation to ensure that no highly-certain PlanPoints are invalid. We also perform heavy-weight POR evaluation periodically, but not on every change to HEAD.

We also support backtracking and regeneration of a "known-good state." Thankfully we rarely use this; the complexity of backtracking is sometimes high, especially for plans near their deployment date, since dependencies can force us to unwind multiple changes. Undoing physical changes is expensive and risky, so we use multi-layered validations, as described later in this section, to avoid backtracking.

Lightweight operations: Network changes that are typically planned in advance are expensive capacity-related operations. Most network changes, however, are small-scale local updates (e.g., link repairs, ToR modifications); such changes are typically done immediately and are thus applied directly to HEAD. As a performance optimization, these direct-to-HEAD changes are typically not specified as PlanPoints (and thus skip the PoR), as they almost never affect future planned capacity changes.

Validations: Because compiling concrete MALT models could be slow and must be performed in sequence, we can perform a lighter-weight *UIM validation* operation on a Plan, which computes and validates the UIM for every PlanPoint without full MALT compilation. This runs intent-validation suites for each network and interconnection intent, first separately (to ensure certain properties and assumptions are met), and then globally (to ensure the UIM intents are consistent).

We also periodically run a detailed validation of the concrete models generated from the PoR PlanPoints. This validation is too expensive to run on every commit, so we made a tradeoff between speedy commits and full validation. Full validations still happen often enough to prevent costly mistakes. When our automated validations detect a conflict between plans (e.g., a missed dependency between PlanPoints that would lead to double-allocation of a port), this usually requires human intervention, to modify one or more plans. Automated resolution is an intriguing research topic.

6 Model generation challenges

Our model-generation system faces several challenges:

Design complexity: Some aspects of our network designs are complex, requiring deep domain knowledge to convert abstract intent into concrete models. In a few cases, e.g., the *Data Center Network Interconnect* (DCNI) layer of connectivity between Jupiter aggregation blocks and spine-blocks, the design is complex enough to require algorithmic support, such as the ILP solver we use to "restripe" the DCNI on changes, while minimizing unnecessary changes to wiring [38].

Heterogeneity: We often add new technologies to our networks – we call these *new product introductions* (NPIs). NPIs sometimes involve novel concepts and so require representational change (see §7). We add NPIs without retiring old products during their useful lifetimes, so our networks are heterogeneous in many design details; our management systems must cope with that. Similarly, we need to be able to rapidly evolve our model-generation system without creating an un-maintainably complex code base.

These challenges, especially our need to support NPIs, pushed us to adopt a layered, modularized design for our model-generation system.

We compose our network designs from fundamental units ("blocks", e.g., server-aggregation blocks and spine-blocks in Jupiter, and B4 blocks [16, 18]). Each block could contain hundreds of chassis and tens of thousands of ports and internal links. A complete data center fabric is composed of up to hundreds of blocks, along with a DCNI. Our fleet has several dozen distinct block types, and each Jupiter or B4 network can have several different generations of blocks.

Our model-generation system uses a modular framework to generate product-specific block-level models, plus additional modules to compose these blocks into a consistent, complete network. For each block type, we have a topology "build unit": a software component that knows how to instantiate that block from high-level intent. These block-level build units are expressed as rules in a concise topology-description language. For many NPIs, we need only create a slightly-modified version of an existing build unit. Other build units, written in traditional programming languages, create inter-block (DCNI) links, assign IP addresses, or validate that the generated models are correct, etc.

When TopoPlan invokes MBS (Fig. 2), MBS creates a dataflow graph, in which the processing steps are the appropriate build units, the input is the intent in UIM, and the outputs are detailed MALT models. MBS constructs this dataflow graph dynamically, to account for changes in our overall network design (the details are beyond this paper's scope).

Scale: Our dataflow graphs are expensive to evaluate, requiring GiBs of I/O and many minutes of CPU time. Concrete MALT models are highly detailed, since they must represent the full underlying detail of our networks. A MALT model representing a single data center network can have millions of entities and relationships, and we have many data centers. Our WAN models have similar scale.

Therefore, MBS uses caching to avoid recomputing previously-generated models. Truly global changes to the Google network are rare and most changes are highly local, so we typically see cache hit rates above 99%, reducing the graph execution costs by two orders of magnitude.

7 Representation evolution

While MALT provides a common, flexible representation, we sometimes need to change its schema, or how we use it, to support NPIs or new management processes.

NPIs generally require changes to model generators, but not always to model readers. For example, a link-speed upgrade from 100G to 200G that otherwise involves no topological changes could be represented by changing the physical_capacity_bps attributes of some EK_PORT entities; this change might not require any updates to model readers.

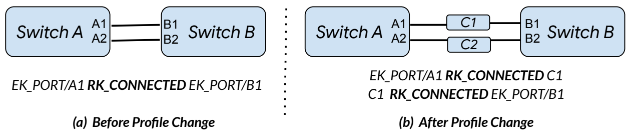

Switch A
A1
A2
B1
B2
Switch B
EK_PORT/A1 RK_CONNECTED EK_PORT/B1
(a) Before Profile Change
Switch A
A1
A2
C1
C2
B1
B2
Switch B
EK_PORT/A1 RK_CONNECTED C1
C1 RK_CONNECTED EK_PORT/B1
(b) After Profile Change

Figure 8: MALT representation of a simple network before and after a profile change.

However, many changes *do* require changes to readers. Fig. 8 illustrates this with a simplified example, where we add new devices (C1 and C2) between switches A and B. Suppose a model reader wants to query what peer switch port is connected to port A1 on switch A. With the old design (Fig. 8(a)), a query could just follow the RK_CONNECTED relationship from A1 to B1. With the new design (Fig. 8(b)), that query would only reach C1, probably not what the query-author intended; the model reader would have to be updated. Because model readers vastly outnumber model writers, such changes are disruptive. Further, a confused model reader (e.g., the configuration generator for our SDN controllers [11]) could cause outages. So, if we fail to realize that a reader needs an update, these changes are also risky.

We mitigate the risk by signalling change through the use of *profiles*. A profile [27] is effectively a versioned contract between model generators and readers, attached to each model. When a generator changes its output in a way that *might* confuse readers or require updates, we increment the profile version. Thus, a reader can detect during testing when it encounters a model with a profile it cannot understand.

MALT profiles by themselves do not avoid the need for updating model readers. Churn due to profile change was a major problem once MALT became widely adopted, which led us to two requirements: (i) Model readers should have to change their code as little as possible (ideally, not at all) in response to profile changes, (ii) We must provide strong guarantees that migrating to a new profile will not result in regressions to model readers.

Key design choices. To avoid the need to update both model generators and all consumers at the same time, our generators produce multiple semantically-equivalent models, with different profiles, from the same intent (the example in Fig. 2 shows models in versions V4.0, V4.1, and V5.0). This allows some consumers to start testing against a new profile, while most consumers read from the most recent "released" profile (a few stragglers may use older profiles).

To simplify or avoid the code-update problem for most model consumers, we developed a semantic query engine, MQS (§8).

8 Model query service

MALT supports querying models via raw traversal-based queries [27]. However, profile changes could break raw MALT queries (see §7). Generating multiple profile versions mitigates this, but migrating to a new profile version is toilsome: (i) Raw MALT queries are structural rather than semantics-based; (ii) Client code does not always include regression tests for new profiles, and when it does, it is difficult to narrow errors to specific queries; (iii) Client code typically queries for specific data (e.g., ports within a rack), but raw queries return full MALT subgraphs (e.g., all the devices, trays, ports and their relationships), making it hard to predict whether a profile change will affect a given client.

This motivated us to develop MQS, an abstraction layer above MALT query, to minimize profile evolution toil. Stonebraker *et al.* discuss a somewhat different solution to the problem they call "database decay" [33].

8.1 Semantic queries

Instead of writing code that explicitly traverses the MALT graph, as was previously done, developers now write code in a new language that captures the semantics of their query (e.g., "give me all peer ports connected to this device") and hides the mechanics of the MALT graph traversal (e.g., "follow RK_CONNECTED relationships on this device's ports until I reach other ports"). The underlying implementation of these "semantic queries" might be different for each profile; this is hidden from the caller.

Semantic queries are recorded in a registry, allowing us to automatically test that they return the same (or at least, consistent) results across profiles and data sets. This testing framework automatically detects unexpected changes in query results for any potential profile or data change at change-review time, protecting model readers well before such changes can affect production.

8.2 Canned queries

MQS offers a *canned query* API. A canned query is a named function, with defined semantics that are profile-independent. Canned queries are *registered* with MQS. When called to execute a canned query, MQS translates it to an appropriate MALT query, executes it, and processes the returned subgraph to return a set of entities (see Fig. 9).

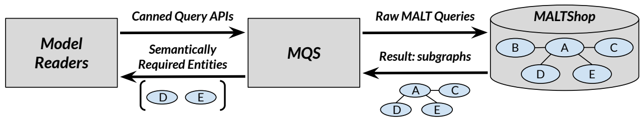


Figure 9: An MQS canned query converts a subgraph to just the entities that are semantically useful to the caller.

Several properties of MQS enable it to return consistent results across multiple profiles, without client code changes: (i) Canned queries can use different MALT queries for different profile versions. (ii) Canned query registration includes semantic tests, allowing centralized regression tests on upcoming "beta" profile versions. (iii) Canned queries return entities with attributes restricted to those that are relevant to clients. This, combined with returning entities rather than MALT subgraphs, greatly reduces the API surface of a model query, allowing easier testing and query evolution.

We run several kinds of centralized validations to ensure minimal profile evolution toil:

(a) **Profile version tests:** we test canned queries across multiple profile versions (including upcoming "beta" versions). Failing tests cause us to either fix the profile or to update the canned-query definition for the new version.

(b) **Data change tests:** When we backport bugfixes or introduce new kinds of products into our network, we introduce changes in multiple profile versions. Before committing these data changes to production models, we generate models in a test environment and compare canned query results between test and production environments.

(c) **Binary and configuration rollout tests:** Prior to releasing a new MQS binary or its associated configuration to production, we run canned queries against the canary and production versions; release automation proceeds only if the results are identical.

9 Cross-shard consistency

We do not try to represent all of Google's networks as one giant model. Not only would the scale create prohibitively expensive memory and communication costs, and complex coordination challenges, but also a single fault could put the entire infrastructure at risk. Instead, we shard the models at natural boundaries (e.g., one shard per datacenter fabric). Sharding allows us to limit the scope of most changes, which greatly simplifies conflict detection.

Many operations span shard boundaries (e.g., adding a WAN link to a datacenter). MALTShop allows queries to transparently span multiple shards, but this raises a question: since we can have multiple future plans (hence, model versions) for each shard, and these plans can be mutated by independent processes, how do we query a *consistent* set of shard versions?

We currently lack a general solution to the shard-consistency problem[6], but we have a workable solution for all datacenter and B4 shards: we (conceptually) rebuild all shards from high-level intent on any intent change (and then use aggressive caching to avoid *actually* building more than necessary). Thus, each intent-change leads to a new *model set*, which is given a unique *model set ID* (MSID) via MALTShop's model-labeling feature. An MSID thus represents a consistent view across these model shards (but unfortunately, not across adjacent shards owned by other systems). MBS uses MSIDs to support consistent queries that cross shard boundaries.

10 Operational experiences

In this section, we dig into several operational experiences and lessons learned. We found the change-management features of TopoPlan to be especially useful in speeding up deployment activities by weeks or months.

10.1 Deadzone reduction

To illustrate one benefit of the PlanPoint abstraction (§5.2), we explain how this speeds up a capacity-delivery process.

We add capacity to Jupiter fabrics, in units of blocks, in a three-step process. **The early modeling** step creates a place-holder version of the block that will exist at a future time. This yields a concrete model from which we can create an order for materials, and it reserves resources, such as patch panel (PP) ports, for this expansion. Once materials are ready for deployment on the data center floor, the **turnup/prepare step** installs and qualifies the new block, but does not connect it to the DCNI. Finally, the **restripe step** gradually folds the new block into the fabric's topology.

However, the same Jupiter fabric may have multiple capacity changes in flight, and we are not always able to overlap the substeps of these augments. Consider the case of a PP expansion followed immediately by a block expansion. We cannot generate the block-to-PP physical striping for the new block until the new PPs and their port reservations are available in a model. Therefore, *if we had only one model*, the first two steps of the block expansion would be blocked until the PP expansion is completed. This results in a *deadzone*, a period when there is work that could be done in a fabric, and is technically unblocked (i.e., we have all the software infrastructure to do the work), but we cannot start because of model-change serialization.

The PlanPoint abstraction allows us to avoid this serialization and effectively pipeline execution. We can create a PlanPoint for the block's early model that uses the post-PP-expansion PlanPoint as its "previous model." The resulting PlanPoint captures a future state when the PP expansion has fully completed and the block's early has been built from it. We can then calculate specific fiber-bundle lengths from this PlanPoint, allowing us to order those bundles long before the PP expansion starts.

[6]We suspect it is similar to distributed-replica consistency, and something like vector clocks might work.

10.2 WAN change management

Our B4 WAN [16, 18] is mutated even more rapidly than a Jupiter fabric, due to its global scale. We change B4 multiple times per day: adding new "neighborhoods," expanding an existing neighborhood, augmenting link capacity between neighborhoods, migrating a neighborhood to a new technology (which entails moving some link endpoints), or removing a neighborhood.

As with Jupiter, we change a live neighborhood or adjacency in multiple steps, to preserve enough capacity to meet SLOs. Thus, we not only generate planned-state models for the end-state topology, but for all intermediate steps as well.

Sometimes projects may be executed independently (e.g., augmenting 2 disjoint edges), but in other cases they are interdependent: e.g., if neighborhood B is port-constrained, adding capacity between neighborhoods A <-> B might first require removing links between B <-> C. Due to real-world constraints, such as supply-chain disruptions, data-center construction delays, etc., the actual execution sequence of these projects rarely follows the global order by which they are initially committed to the plan-of-record (which happens well in advance).

We augmented TopoPlan to express and track such dependencies, adding a layer above TopoPlan to prevent unsupported execution sequences. This allows us to manage WAN projects extending years into the future. Because of the long lead times for materials (fiber bundles, optics, etc.), our ability to pipeline and avoid unnecessary and rigid serialization of plans can shorten deployment timelines by weeks or months.

10.3 Early materials procurement

Supply-chain problems often make procurement of materials (switches, transceivers, etc.) the longest step in a network change. So, we try to order materials as early as possible, ideally before we know exactly how a new network block will fit when we install it (given other in-progress changes with fuzzy completion schedules). In the past, we used heuristics based on prior projects (and some simple scaling rules) to do early orders; this was not always reliable.

Instead, now we speculatively generate a detailed model of the post-change network, and from that we compute and place a precise order. TopoPlan allows us to create these speculative models and manipulate them exactly as we manipulate "real" models, using unmodified software and workflows.

10.4 Software migration

Every software system sooner or later becomes obsolete. Our modeling infrastructure has gone through major software migrations several times, first transitioning consumers from an older representation to MALT, and then changing from a set of monolithic model generators to a modular, more evolvable framework. Managing these migrations without production outages required us to carefully introduce changes in phases. We describe our approach to migration in appendix §C.

11 Summary and future work

We have described how we plan and coordinate changes to large network infrastructures, with a specific emphasis on the need to support parallelism in the face of plan-changes caused by evolving real-world constraints. Since our network-management systems are heavily automated, our plan management must therefore also be automated to keep up with the pace of change, and to avoid mistakes.

We described the TopoPlan system for change management (§5.2), and the fundamental concepts (PlanPoints and branches) it relies on. We discussed how this approach has significantly increased the velocity with which we manage changes to both datacenter networks and WANs (§10.1, §10.2).

Automated network management also depends on explicit models of current and planned topologies, and innovation often requires representational change for our modeling language. We described how we support representational change without major software-engineering disruption, by means of explicit profiles (§7), and a profile-independent query layer that supports many (but not all) model consumers (§8).

Research challenges: While we have already greatly benefited from the work described in this paper, we see many challenges that demand future improvements in change management. In particular, supporting cross-shard consistency, at scale and without funneling all changes through a logically-centralized owner, remains unsolved (see §9). There might also be ways to expand the set of properties that can be validated automatically – not just simple resource conflicts, but also higher-level goals such as fault-resilience.

Acknowledgments

The systems described in this paper represent the design and engineering efforts of many current and former Googlers, in addition to the authors. We would especially like to thank Omid Alipourfard, Drago Goricanec, Xander Lin, Bret McKee, Joon Ong, Sujithra Periasamy, Mitchell Price, Fang Ruan, Michael Rubin, Shouqian Shi, Yiran Su, and Sting Zhou for their contributions.

We thank Vince Muir and Bryan Stiekes for their helpful advice. We also thank, for their feedback on this paper and earlier versions, Aditya Akella, Dennis Fetterly, several sets of anonymous reviewers, and our shepherd, Nik Sultana.

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. SIGCOMM*, 2008.

[2] A. Andreyev. Introducing data center fabric, the next-generation Facebook data center network. https://engineering.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/, 2014.

[3] Arista Networks, Inc. Arista 7500 Scale-Out Cloud Network Designs. https://www.arista.com/assets/data/pdf/Whitepapers/Arista_7500_Scale_Out_Designs.pdf, 2016.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proc. SIGCOMM*, 2017.

[5] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations . In *Proc. SIGCOMM*, 2016.

[6] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *Proc. SIGPLAN*, 2010.

[7] Xu Chen, Yun Mao, Z Morley Mao, and Jacobus Van der Merwe. Declarative Configuration Management for Complex and Dynamic Networks. In *Proc. CoNEXT*, 2010.

[8] Cisco. Cisco Data Center Infrastructure 2.5 Design Guide. https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCI_SRND_2_5a_book/DCInfra_2a.html, 2011.

[9] Kip Compton. Cisco's Global Cloud Index Study: Acceleration of the Multicloud Era. https://blogs.cisco.com/news/acceleration-of-multicloud-era, 2018.

[10] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI*, 2004.

[11] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and

Amin Vahdat. Orion: Google's Software-Defined Networking Control Plane. In *Proc. NSDI*, 2021.

[12] Google. Blaze. http://googleengtools.blogspot.com/2011/06/testing-at-speed-and-scale-ofgoogle.html, 2011.

[13] Google. MALT Example Models. https://github.com/google/malt-example-models, 2023.

[14] Google. Protocol Buffers. https://developers.google.com/protocol-buffers, 2023.

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

[16] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-Defined WAN. In *Proc. SIGCOMM*, 2018.

[17] IEEE Standards Association. IEEE 802.1AB-2016: IEEE Standard for Local and metropolitan area networks - Station and Media Access Control Connectivity Discovery. https://standards.ieee.org/ieee/802.1AB/6047/, 2016.

[18] 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. B4: Experience with a Globally-Deployed Software Defined WAN. In *Proc. SIGCOMM*, 2013.

[19] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. NSDI*, 2012.

[20] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proc. HotSDN*, 2012.

[21] Hyojoon Kim, Theophilus Benson, Aditya Akella, and Nick Feamster. The Evolution of Network Configuration: A Tale of Two Campuses. In *Proc. IMC*, 2011.

[22] 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. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *Proc. SIGCOMM*, 2015.

[23] S. Lee, T. Wong, and H. S. Kim. To Automate or Not to Automate: On the Complexity of Network Configuration. In *Proc. IMC*, 2008.

[24] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. Automatic Life Cycle Management of Network Configurations. In *Proc. SelfDN*, 2018.

[25] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A Fault-Tolerant Engineered Network. In *Proc. NSDI*, 2013.

[26] Guillaume Maudoux and Kim Mens. Correct, Efficient, and Tailored: The Future of Build Systems. *IEEE Software*, 35(2), 2018.

[27] Jeffrey C Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with Modeling Network Topologies at Multiple Levels of Abstraction. In *Proc. NSDI*, 2020.

[28] OpenConfig Working Group. OpenConfig. https://openconfig.net/, 2014.

[29] Richard Peterson. Top 25 Cloud Computing Service Provider Companies (2021). https://www.guru99.com/cloud-computing-service-provider.html, 2021.

[30] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Connor, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yamsura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter Evolving: Transforming Google's Datacenter Network via Optical Circuit Switches and Software-Defined Networking. In *Proc. SIGCOMM*, 2022.

[31] 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. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proc. SIGCOMM*, 2015.

[32] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. In *Proc. NSDI*, 2012.

[33] Michael Stonebraker, Dong Deng, and Michael L. Brodie. Database Decay and How to Avoid It. In *Proc. Big Data*, 2016.

[34] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down Network Management at Facebook Scale. In *Proc. SIGCOMM*, 2016.

[35] Amin Vahdat. NANOG Keynote: Failing Last and Failing Least. https://www.nanog.org/news-stories/nanog-tv/keynotes/keynote-failing-fast-and-failing-least/, 2020.

[36] G.G. Xie, Jibin Zhan, D.A. Maltz, Hui Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On Static Reachability Analysis of IP Networks. In *Proc. INFOCOM*, 2005.

[37] Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. Understanding Lifecycle Management Complexity of Datacenter Topologies. In *Proc. NSDI*, 2019.

[38] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C Mogul, and Amin Vahdat. Minimal Rewiring: Efficient Live Expansion for Clos Data Center Networks. In *Proc. NSDI*, 2019.

Appendices

Whenever it is necessary to build concrete MALT models from a UIM, we invoke MBS (§A), a "build service" that compiles the high-level intent from TopoPlan to generate MALT models, using product-specific model producers. Collectively, we refer to our module-generating software as Nimble (§B).

A Generic build service: MBS

Model production is the result of the execution of a dataflow graph of *build units* (§B.1). Build units are product-specific, but the orchestration of their execution to construct MALT models is generic. MBS is an execution engine that, given a set of named inputs, (i) constructs the dataflow graph of build units, (ii) executes that graph to produce MALT models, and (iii) transactionally stores the generated models in MALTShop. A set of output models in one transaction is given a *Model Set ID* (MSID), so that model readers can see a consistent snapshot of models.

The main input to MBS is a global UIM representing specifications for the entire Google network at a given point in time; this concisely represents the desired high-level state (e.g., the number of network fabrics in a location, their topology type, interconnect capacity between sites, etc.). MBS is also stateful; build units record their low-level decisions as an input to future model builds, to reduce network churn.

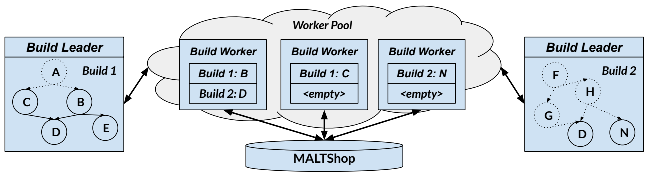


Figure 10: An illustration of our distributed build system showing two independent builds in progress.

Two-phase build. MBS is a distributed dataflow graph evaluator. A graph node can represent either a datum or a rule: data nodes are almost always MALT model fragments, while rule nodes execute build units. Rule nodes are connected to data nodes via either input edges, which specify the input models for a build unit, or output edges, which specify a build unit's output models.

Because the dataflow graph used to generate a set of concrete MALT models is highly data-dependent (e.g., a new MALT model output will be added if we're modeling a new data center location), we dynamically construct this dataflow graph in MBS, by executing a much smaller, static dataflow graph. Special build units in the static graph read the intent model and compute the full, dynamic graph, which MBS then executes to produce concrete models.

Graph execution (static or dynamic) is orchestrated by a leader and worker distributed system (Fig. 10). To execute a graph, MBS assigns it to one leader, which parses and validates the graph, then executes rules in parallel, using a pool of workers, as the rule inputs become available.

Build performance optimization. The resulting full-size dataflow graph, with 100s of thousands of rule nodes, is slow and expensive to evaluate, requiring GiBs of I/O and many minutes of CPU time. Therefore, MBS uses extensive caching to avoid recomputing previously-generated models. Caching is based on hashes of input data nodes and rule specifications, allowing MBS to skip rule evaluation if the corresponding hashes identify a cached output model.

Why we built a distinct system: While Google and others have created extensive distributed dataflow graph execution engines [6, 10] and tooling to efficiently manage and build binaries from source code [12, 26], we created MBS as a distinct system for several reasons:

(a) **Dynamic graphs:** Existing systems expect the execution graph to be provided as an input, rather than itself being dynamically generated based on the input intent models.

(b) **Stateful execution:** Updates to a fabric's concrete model should implement the new planned intent while making as few changes as possible to the existing network (to limit the costs of physical changes). Existing build systems [12] did not easily allow execution i of model generation to consume the output of execution $i-1$.

We suspect the combination of (a) and (b) is novel in the context of build systems.

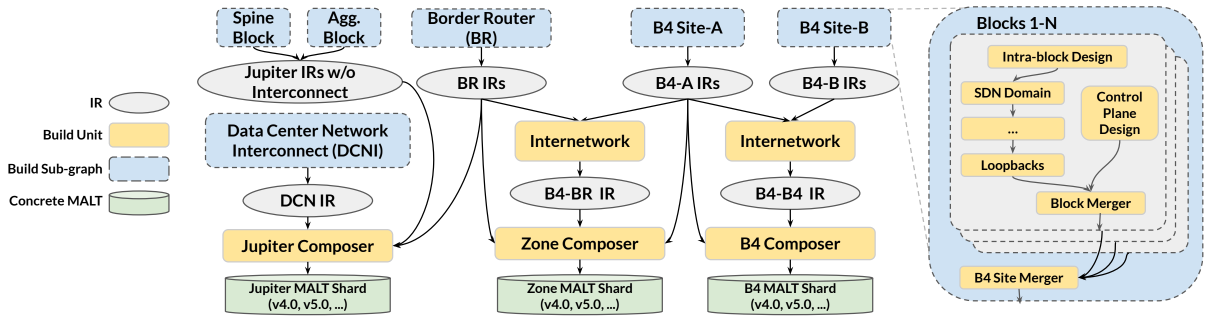


Figure 11: Simplified build graph for producing Jupiter and B4 models, with some areas abstracted for clarity. An expanded view of the B4 subgraph is shown on the right as an example (with some IRs omitted).

B Model generation: Nimble

We developed an ecosystem of re-usable *build unit* software modules, collectively called Nimble, each responsible for constructing a specific part of a model. We organize the build units into a dataflow graph that makes their dependencies explicit. Compared with a monolithic generator, using modular build units respects the knowledge domains involved in designing different aspects of a network (e.g., topology design, IP assignment, port allocation, link striping, etc.).

Modularity also provides natural boundaries to confine problem complexity. For instance, we divide topology design into deterministic descriptor-driven build units and dynamic solver-driven build units. This makes the generation of large-scale topologies with specific properties more tractable than directly solving a monolithic topology optimization problem. Modules without dependencies (or satisfied dependencies) can be executed in parallel, which helps scaling. Finally, modularity supports heterogeneity, since we can add or change just the necessary build units, and reuse others.

Each product's model-producers typically define dozens of build-unit types, to collectively perform the full spectrum of model generation tasks for each fabric and interconnect in the network, including topology generation, IP address allocation, SDN controller domain assignment, etc. We roughly categorize build units into four classes: (i) intra-block topology generators (§B.1.1), (ii) inter-block capacity generators (§B.1.2), (iii) model fragment composers (§B.1.3), and (iv) model validators (§B.1.4). While the first two categories both generate topologies, their designs are drastically different. The model fragment composers and validators are at the end of the design pipeline, and are necessary to support model sharding and ensure correctness.

Fig. 11 depicts a simplified example build graph involving the model producers for both B4 and Jupiter networks. Build units shown in the figure are abstracted, simplified representations of those we use in production.

B.1 Build unit overview

A build run creates two categories of model data: (i) the final concrete MALT model(s) adhering to a specific profile, and (ii) Intermediate Results (IRs): internal data, also represented in MALT, that facilitate communication between build units, but are not exposed for external consumption.

Some IRs are ephemeral, since they are consumed by downstream build units in the same build run. Some IRs are memoized results, required for the future build runs (e.g., the set of allocated ports). Memoization avoids the need to always build all output from scratch. Some IRs are part of the inputs to the composers (see §B.1.3), which process and stitch these together to generate the concrete, versioned model-shard outputs. Each build unit may take as input the model intent, previous IRs, and previous concrete models, and outputs IRs or concrete models.

We compose our network fabrics from fundamental units called "blocks" (e.g., server-aggregation blocks and spine-blocks in Jupiter, and B4 blocks [16, 18]). Each block could contain hundreds of chassis and tens of thousands of ports and internal links. A complete data center fabric is composed of up to hundreds of blocks, along with a "Data Center Network Interconnect" (DCNI). Our fleet can have several dozen distinct block types, as their technology evolves.

Each block type has a fixed, deterministic internal topology, but the DCNI or WAN interconnect depends on their dynamic properties (e.g., block type, uplink capacity, port availability, etc.). We generate each intra-block topology from a declarative topology description (§B.1.1) but we generate the DCNI and WAN interconnects via solvers that optimize link striping and port allocations (§B.1.2).

B.1.1 Intra-block topology generator

The intra-block topology generator is effectively a compiler that parses a *topology descriptor* that declaratively describes the desired block topology, and emits a corresponding MALT fragment of the detailed design. These descriptors are parameterized templates for each topology type. This process is deterministic, and does not require a complex solver, given the regular design of block internals.

The descriptor language expresses intent for a given block type as a hierarchy of modules, with specific entity-kinds (e.g., packet switches or ports) as leaf nodes. Descriptors can specify entity attributes, which can be parameterized (e.g., the index of a module within its parent), and naming schemes. Interconnection patterns between entities, such as ports, are selected via *mappers* such as *fullmesh* or *biject*, within a scope called a *group*. When invoking the compiler, a build unit can pass certain parameters (abstracted from the model intent) to the descriptor; e.g., to control the number of racks within the B4 block. This allows one descriptor to support a variety of topologies for the same block-type generation. (§B.2 provides more details on the descriptor-based approach.)

While the build-graph structure is flexible, for simplicity, the typical pattern in most model producers' pipelines is for the first build unit to invoke the compiler to construct the backbone IR for a block, while subsequent build units build on this IR with additional entities and relationships (e.g., allocating management IPs, SDN control domains, etc.).

B.1.2 Inter-block capacity design

The internal topology of a block is typically static throughout its lifecycle. Inter-block connectivity, however, is frequently updated as we add or decommission blocks, add capacity between blocks, or fix incorrectly-wired fibers. Updates to the topology must meet capacity and availability requirements, and also minimize change to deployed reality (i.e., not move fibers unnecessarily). We have several solver-based build units for inter-block connectivity that tackle different classes of problems. *E.g.*, the *Internetwork* build unit in Fig. 11 uses a generic interconnect design and management solver for block-level striping, port-allocation, interface IP addressing, etc. This is used to generate WAN connections between B4 sites, and to the *Border Routers* of the data center fabrics.

These build units try to maximize the path diversity between pairs of sites or blocks, which improves tolerance of physical faults (e.g., link-, chassis-, block-level failures), while adhering to physical deployment constraints (e.g., minimizing the number of wasted ports).

For pairs of B4 sites, for example, each site may span several Points of Presence (PoPs), each containing multiple blocks; the interconnect solver minimizes the maximum imbalance in block-to-block, PoP-to-PoP, and block-to-PoP allocations of links across block-pair. We formulate this as a mixed integer programming optimization problem.

The design problem for the data center network interconnect (DCNI) has a large optimization space. We discuss that solver in appendix §B.3.

B.1.3 IR composers

At the end of each model generation run, a set of *composers* is responsible for stitching together the IRs produced by the upstream built units to create the concrete model shards. The MALT models are sharded for a variety of reasons, such as domain isolation and scalability, as discussed in [27]. We have a dedicated composer for each model shard.

Within each shard, the composer processes and merges IRs, based on their tagged profiles, to generate profile-compliant models for all supported profile versions. We define our pipeline such that any profile-agnostic processing (e.g., resource allocation, etc.) is done as early as possible, while profile-dependent modeling is typically branched further downstream, at or near the composers; this helps ensure data consistency across profiles.

B.1.4 Validators

During each model generation run, we also validate attributes, and design rules in several categories: (i) *Intent validation* ensures that the UIM is internally consistent and its changes are legitimate; e.g., the UIM satisfies the properties required by model producers. (ii) *Property validation* focuses on validating network-specific invariants we expect in each model (e.g., ports do not conflict, IP addresses are not duplicated), and (iii) *intent-to-model validation*, which is designed to harden the intent-to-model translation that typically requires dynamic solvers (e.g., whether the striping between a B4 neighborhood and Jupiter delivers the intended capacity, while satisfying diversity and balance requirements). Finally, (iv) *model-change scope validation* ensures the scope of model changes matches the corresponding change in the intent. [7]

During the course of development, all these validation suites have caught some exceptions, especially for NPIs, which if left undetected would have caused network outages or costly deployment errors.

B.2 Details: intra-block topology generator

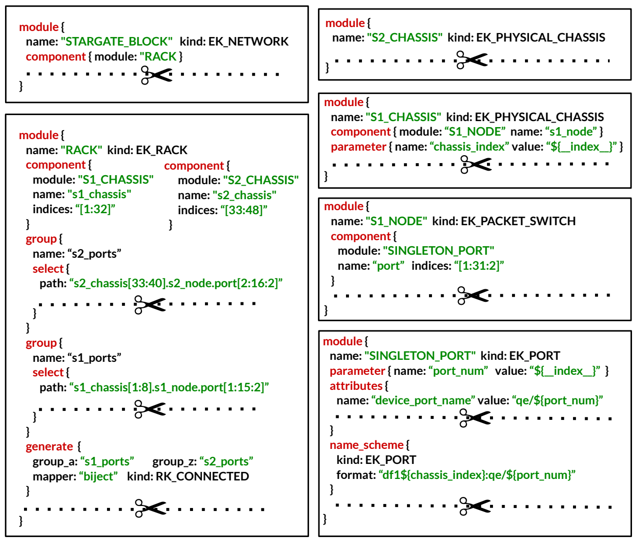
This section provides additional details on how we support a high-level approach to block-level design. The intra-block topology generator includes (i) topology descriptors that fully declare the topology and (ii) a compiler that parses the descriptors and translates them into MALT models. These descriptors deterministically declare the intended topology.

We explain several key aspects of topology descriptors using an example snippet (Fig. 12) of the descriptor of a B4 Stargate block [16]. We simplified the descriptor for clarity.

Hierarchy: A network is a hierarchy of *modules* with a single tree root and multiple branches. The root module of a block is usually an EK_NETWORK, and its name is globally unique.

Modules: A module, the basic building block in descriptors, defines one MALT entity kind and the topology within the entity. As our networks are heterogeneous, multiple modules (with distinct names) may refer to the same entity kind. For instance, we have two module definitions for EK_PHYSICAL_CHASSIS in Fig. 12. A descriptor can have multiple instances for the same module.

[7] MBS also performs more basic validations such as MALT lint checks and profile schema checks.



```
module {
  name: "STARGATE_BLOCK"  kind: EK_NETWORK
  component { module: "RACK }
}
```

```
module {
  name: "RACK"  kind: EK_RACK
  component {                     component {
    module: "S1_CHASSIS"            module: "S2_CHASSIS"
    name: "s1_chassis"              name: "s2_chassis"
    indices: "[1:32]"               indices: "[33:48]"
  }                               }
  group {
    name: "s2_ports"
    select {
      path: "s2_chassis[33:40].s2_node.port[2:16:2]"
    }
  }
  group {
    name: "s1_ports"
    select {
      path: "s1_chassis[1:8].s1_node.port[1:15:2]"
    }
  }
  generate {
    group_a: "s1_ports"     group_z: "s2_ports"
    mapper: "biject"   kind: RK_CONNECTED
  }
}
```

```
module {
  name: "S2_CHASSIS"  kind: EK_PHYSICAL_CHASSIS
}
```

```
module {
  name: "S1_CHASSIS"  kind: EK_PHYSICAL_CHASSIS
  component { module: "S1_NODE"  name: "s1_node" }
  parameter { name: "chassis_index" value: "${__index__}" }
}
```

```
module {
  name: "S1_NODE"  kind: EK_PACKET_SWITCH
  component {
    module: "SINGLETON_PORT"
    name: "port"   indices: "[1:31:2]"
  }
}
```

```
module {
  name: "SINGLETON_PORT"  kind: EK_PORT
  parameter { name: "port_num"   value: "${__index__}" }
  attributes {
    name: "device_port_name" value: "qe/${port_num}"
  }
  name_scheme {
    kind: EK_PORT
    format: "df1${chassis_index}:qe/${port_num}"
  }
}
```

Figure 12: A snippet of a topology descriptor for a Stargate Block.

Components: The topology within a module is defined by recursively including other modules as its *components*. The number of components (of the same entity kind) included in the parent module is concisely expressed using indices. For instance, in the module "S1_NODE", the singleton port component is defined with indices [1:31:2], indicating that there are 16 entities in this module, with indices from $\{1,3,...,31\}$. The default relationship between a module and its components is that the module entity RK_CONTAINS all its components. Other relationship types can be specified to override that default, at the component type granularity.

Attributes: Entities in MALT models can have attributes, such as taxonomy (e.g., chassis type) and state (e.g., link is in turnup). Attributes specified in topology descriptors are self-contained: i.e., they are either static values, or they are deterministically computable using the *parameters* defined within the upstream hierarchy (branch) of the entity.

Parameterization: The descriptor is not another topology programming language – we omitted constructs such as conditionals. However, allowing basic parameterization of modules offers useful flexibility and concision. The most common use of parameters is to pass information top-down. For instance, the chassis_index parameter of the "S1_CHASSIS" module is subsequently used by the singleton port contained by the chassis. If a module is componentized with indices, we have multiple instances of this module; the "${__index__}" provides each instance's index. Parameters defined in a module are recursively visible to all components and sub-components in the module.

Relationships: To create relationships between components or create intra-block links between ports, the descriptor introduces the *Group* operation to *Select* a set of components within a module hierarchy, and then applies a *Generate* operation to generate relationships or links between the components of the two groups. A *mapper* is used to decide how the components in *group_a* are mapped to those of *group_z*. Two common mappers are *biject* (pairwise, requiring the two groups to have the same number of items) and *fullmesh*. In Fig. 12, the "RACK" module uses group and generate to define how S1 ports and S2 ports are connected.

Naming schemes: Each MALT entity has a unique ID, combining its name and entity kind. A topology descriptor specifies a naming scheme by either constructing it ad-hoc (potentially using parameters) or simply referring to a preconstructed regular expression (in most cases).

Given a descriptor, the topology compiler is responsible for parsing the descriptor and generating the corresponding MALT model fragment. Internally, the compiler parses modules top-down, builds multiple branches based on the component indices while enforcing parameter scopes within each branch, and finally constructs entity names, attributes and relationships. The compiler also allows customized mappers in the Generate operations to compensate for topology irregularities. Because the compiler does not make any topology-specific assumptions, it is generic and reusable across all descriptors.

For most model producers, their first build unit instructs the topology compiler to construct the backbone IR for a network. Subsequent build units decorate the IR with additional entities and relationships (e.g., allocating management IPs and SDN control domains). Fig. 11 depicts these data flows. When invoking the compiler, a build unit can pass certain parameters (abstracted from the model intent) to the descriptor; for instance, to control the number of racks within the B4 block. This enables us to support a variety of topologies for the same network generation while reusing the same descriptor.

B.3 Details: inter-block topology generator

This section expands on the discussion in §B.1.1, describing how we generate models for the Data Center Network Interconnect (DCNI) in a Jupiter network.

DCNI overview. A Jupiter network uses a layer of Patch Panels (PPs) between server blocks and spine blocks. Each block (server or spine) directly connects to the front side of PPs, and the block-to-block connectivity is (indirectly) established by cross-connecting the back side of PPs. Having a PP layer removes a lot of complexity that would be introduced by directly connecting server and spine blocks, e.g., reduced fiber length and human labor. This is discussed in detail in [38].

We use the term DCNI (Data Center Network Interconnect) to describe the two collections of "physical" links, i.e., block-to-PP links and PP cross-links. The challenge for inter-block design is to produce an optimal DCNI. We call the resulting block-to-block paths "logical" links.

Block-to-PP generation. The Jupiter model producer has a dedicated DCNI build unit. This build unit first performs the block-to-PP link generation to construct the physical topology, and then generates PP cross-links to produce the desired logical block-to-block topology. For a given block, its block-to-PP fibers fan out equally across every patch panel using a predetermined pattern (i.e., agnostic to intent). Once the block is deployed, such fibers never change.

The block-to-PP links are dynamically allocated in three steps: (i) A *block-to-PP spec generator* translates the UIM into an IR specifying the number of PP ports needed for each block; (ii) A *patch panel port allocator* takes that IR as input, and dynamically assigns available ports to block-to-PP fibers in an on-demand manner. It must read the previous models in order to honor deployed reality; (iii) A *bad port swapper* reads bad-port UIM, and uses reserved ports to replace those bad ports. An external device-repair workflow automatically creates bad-port UIM to record faulty ports. Since block-to-PP link restripe does not affect traffic (because these new links have not been used to carry traffic), this restripe is accomplished in one shot, i.e., without phasing.

PP cross-link generation. Given the physical topology, obtaining the desired logical topology is a complex problem. Thus, the DCNI build unit invokes a dedicated external solver, which translates PP cross-link generation into an ILP (Integer Linear Programming) problem, as described in [38].

Although the ILP solver is able to compute the desired final state, it does not naturally support the crucial requirement that the DCNI must carry live traffic during restripe. This requirement is addressed by having multiple incremental restripe stages, where each stage only alters a small portion of topology, to ensure that the network has sufficient residual capacity in all stages. We use an automated *expansion planner* to decide the number of required stages. Given a restripe request, the expansion planner iteratively searches for the smallest number of stages C (starting from 1) that satisfies the residual capacity requirement. For each evaluated value of C, the DCNI build unit invokes MDS to generate a series of C hypothetical models that resemble each intermediate stage. The residual bandwidth is then calculated, from these models, by counting the number of added/removed links. We provide additional details for the restripe process in §B.4.

Another goal of the DCNI build unit is to ensure topology stability. By its nature, the logical topology solver is not deterministic. Invoking the solver in different model generation runs could cause the DCNI to change arbitrarily, even without any intent change, forcing us to do useless re-wiring. Thus, we use a persistent memoization layer, allowing the solver to store its calculated topology solutions persistently. This both prevents redundant calculation, and ensures that the DCNI build unit generates deterministic output.

B.4 Case study: Jupiter restripe

When we add blocks or spine blocks to a Jupiter network, we must perform a "restripe" operation, to redistribute uplinks from aggregation blocks to spine blocks. When we do a restripe that could affect in-service links, we must do it incrementally, to avoid distrupting too much capacity at once.

In this section, we use the restripe process to illustrate details about the DCNI build unit.

There are three major catogeories of Jupiter data center restripes. (i) Front-only restripe: the restripe only adds/removes the block-to-PP links that do not have fiber jumpers on the back side, so it touches only the front side of patch panels. These added/removed links do not carry live traffic, so incremental restripe is not required. (ii) Back-only restripe: the restripe only alters the PP cross-links, so the scope of change is limited to the fiber jumpers on the back side. Incremental restripe is required since it changes the logical block-to-block links. (iii) Combined restripe: the restripe alters both block-to-PP and PP cross-links at the same time. It is the most labor-intensive process compared with the other categories, and it is also required to be an incremental restripe.

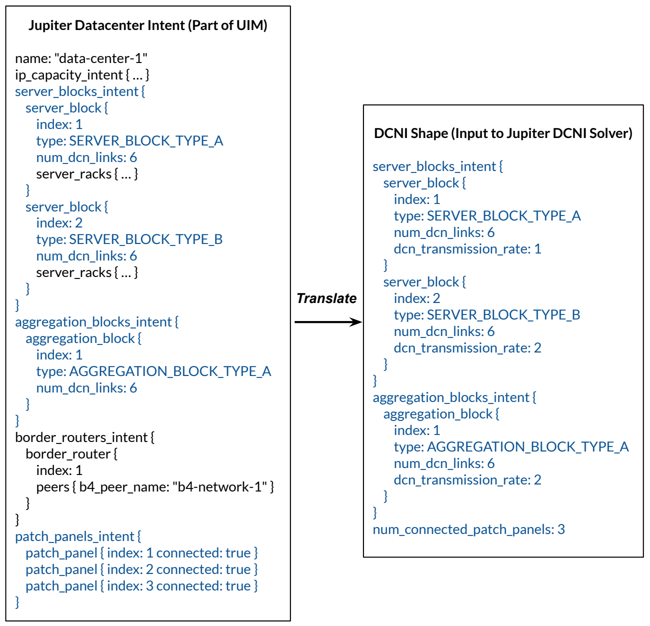
We observed that for most common restripe use cases, the model update process could be divided into one front-only restripe and multiple stages of back-only restripe. As an example, when a new block is added to a Jupiter fabric, we first add the block-to-PP links (front-only restripe), and then we use multiple incremental stages to update PP cross-links (back-only restripes). Such granularization helps reduce the sequence requirement in the workflow, and allows better parallelism among different workflows that are expanding different parts of a Jupiter fabric.

Intermediate restripe stages. Similar to most model changes, the DCNI design is also driven by intent changes. Fig. 13 shows an example of high-level Jupiter data center intent (on the left side), which is part of the global UIM. The fields that are related to DCNI are highlighted in blue (e.g., the number of PP chassis). The DCNI build unit is responsible for translating the Jupiter intent into DCNIShape, a protocol buffer defined as the input to the topology solver interfaced with the *PP-Cross-Link Gen* build unit, a subcomponent of the logical DCNI build unit.

The PP cross-link restripe process consists of small stages to gradually transform the current topology to the desired one given by the solver. We use a single PP rack as the smallest granularity of restriping stages, because it provides natural grouping of the logical block-to-block links. Thus the whole restripe process boils down to rewiring patch panels in multiple incremental stages.

Conceptually, the model for each stage could be summarized as a 4-tuple: $(\mathsf{T}_o, \mathsf{PP}_o, \mathsf{T}_n, \mathsf{PP}_n)$, where T_o and T_n denote the two sets of all physical block-to-block links in the old and new topologies; PP_o and PP_n form a partition of all PP indices, where PP_n represents PPs that are restriped to have links from T_n, and PP_o represents the rest of patch panels that still have links from T_o. Fig. 14 shows a 3-stage restripe with three PPs:

(a) Before restripe. There is no old topology. The "new" physical topology T_0 connects server block 1 and spine block 1 via PP 1, 2, and 3. Thus the tuple is



Jupiter Datacenter Intent (Part of UIM)

```
name: "data-center-1"
ip_capacity_intent { ... }
server_blocks_intent {
  server_block {
    index: 1
    type: SERVER_BLOCK_TYPE_A
    num_dcn_links: 6
    server_racks { ... }
  }
  server_block {
    index: 2
    type: SERVER_BLOCK_TYPE_B
    num_dcn_links: 6
    server_racks { ... }
  }
}
aggregation_blocks_intent {
  aggregation_block {
    index: 1
    type: AGGREGATION_BLOCK_TYPE_A
    num_dcn_links: 6
  }
}
border_routers_intent {
  border_router {
    index: 1
    peers { b4_peer_name: "b4-network-1" }
  }
}
patch_panels_intent {
  patch_panel { index: 1 connected: true }
  patch_panel { index: 2 connected: true }
  patch_panel { index: 3 connected: true }
}
```

DCNI Shape (Input to Jupiter DCNI Solver)

```
server_blocks_intent {
  server_block {
    index: 1
    type: SERVER_BLOCK_TYPE_A
    num_dcn_links: 6
    dcn_transmission_rate: 1
  }
  server_block {
    index: 2
    type: SERVER_BLOCK_TYPE_B
    num_dcn_links: 6
    dcn_transmission_rate: 2
  }
}
aggregation_blocks_intent {
  aggregation_block {
    index: 1
    type: AGGREGATION_BLOCK_TYPE_A
    num_dcn_links: 6
    dcn_transmission_rate: 2
  }
}
num_connected_patch_panels: 3
```

Figure 13: An example of DCNI intent translated from UIM.

$(\{\},\{\},\mathsf{T}_0,\{1,2,3\})$.

(b) Stage #1. The new topology physical T_1 connects server block 1, 2, and spine block 1. Only patch panel 1 has been updated. Thus the tuple is $(\mathsf{T}_0,\{2,3\},\mathsf{T}_1,\{1\})$.

(c) Stage #2. The new topology T1 remains the same. PP 2 is further folded into the logical topology. Thus, the tuple could be summarized as $(\mathsf{T}_0,\{3\},\mathsf{T}_1,\{1,2\})$.

(d) Stage #3. The new topology T1 remains the same. All PPs are updated into the logical topology. There is no more "old" topology as restripe is completed. Thus, the is tuple $(\{\},\{\},\mathsf{T}_1,\{1,2,3\})$.

This tuple is stored in an IR called MaskedDcnTopology IR. The PP-Cross-Link Gen build unit will read the previous MaskedDcnTopology IR and Jupiter intent to update the tuple, and then translate the tuple to an intermediate topology.

C Live migration to new infrastructure

Our legacy modeling infrastructure[8] had numerous problems, including scaling issues, and weak support for schema evolution and parallel operations. For production safety, we could not simply stop using the old systems and immediately migrate its many users to our new model-generation infrastructure (Nimble) until we were confident that the old and new systems were functionally equivalent. Even small discrepancies can cause serious outages. However, we could not just stop using the old systems while we tested the new ones, as that would have blocked all network operations for weeks or months.

[8]The legacy infrastructure was similar to a relational database schema.

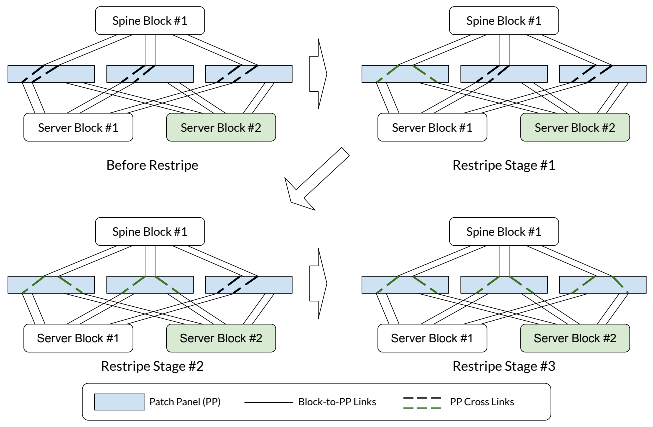


Figure 14: An example of 3-stage restripe with three PPs.

We conducted a live migration from the old systems in four phases:

Phase I: Exporter. We wrote an *exporter* pipeline that converted legacy models to equivalent MALT models. This allowed most model consumers to migrate to MALT. At this point, the legacy models were still treated as authoritative.

Phase II: Validation. To avoid production outages, we had to ensure that models produced by Nimble were functionally equivalent to the exporter-generated models. We built a pipeline that reverse-engineered UIM and relevant state from the exporter's output, yielding intent that we could feed to Nimble. We could then check that Nimble and the exporter generated identical MALT models from semantically-equivalent intent.

Phase III: Read migration. After the model equivalence checks passed consistently for several weeks, we atomically flipped the MALTShop paths where the exporter and Nimble wrote their models, automatically causing readers to consume Nimble-generated models. We staged this changeover on a per-model-shard basis, enabling read migration for some shards while we were still fixing differences for other shards.

Phase IV: Write migration. We then migrated all model writers (e.g., capacity-delivery and data center expansion workflows) to use MDS. After this, we deprecated the legacy model-design tools.

Phased migration turned out to be invaluable. Together with our high-level design principles for reliable systems [35], we managed to finish the fleet-wide migration of our critical modeling infrastructure without any production incidents.