Running BGP in Data Centers at Scale

Anubhavnidhi Abhashkumar and Kausik Subramanian, *University of Wisconsin–Madison;* Alexey Andreyev, Hyojeong Kim, Nanda Kishore Salem, Jingyi Yang, and Petr Lapukhov, *Facebook;* Aditya Akella, *University of Wisconsin–Madison;* Hongyi Zeng, *Facebook*

https://www.usenix.org/conference/nsdi21/presentation/abhashkumar

This paper is included in the Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation.

April 12–14, 2021

978-1-939133-21-2

Open access to the Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

Running BGP in Data Centers at Scale

Anubhavnidhi Abhashkumar[♯*†], Kausik Subramanian[♯*], Alexey Andreyev[◇], Hyojeong Kim[◇], Nanda Kishore Salem[◇], Jingyi Yang[◇], Petr Lapukhov[◇], Aditya Akella[♯], Hongyi Zeng[◇]
University of Wisconsin - Madison[♯], Facebook[◇]

Abstract

Border Gateway Protocol (BGP) forms the foundation for routing in the Internet. More recently, BGP has made serious inroads into data centers on account of its scalability, extensive policy control, and proven track record of running the Internet for a few decades. Data center operators are known to use BGP for routing, often in different ways. Yet, because data center requirements are very different from the Internet, it is not straightforward to use BGP to achieve effective data center routing.

In this paper, we present Facebook's BGP-based data center routing design and how it marries data center's stringent requirements with BGP's functionality. We present the design's significant artifacts, including the BGP Autonomous System Number (ASN) allocation, route summarization, and our sophisticated BGP policy set. We demonstrate how this design provides us with flexible control over routing and keeps the network reliable. We also describe our in-house BGP software implementation, and its testing and deployment pipelines. These allow us to treat BGP like any other software component, enabling fast incremental updates. Finally, we share our operational experience in running BGP and specifically shed light on critical incidents over two years across our data center fleet. We describe how those influenced our current and ongoing routing design and operation.

1 Introduction

Historically, many data center networks implemented simple tree topologies using Layer-2 spanning tree protocol [5, 11]. Such designs, albeit simple, had operational risks due to broadcast storms and provided limited scalability due to redundant port blocking. While centralized software-defined network (SDN) designs have been adopted in wide-area networks [28, 29] for enhanced routing capabilities like traffic engineering, a centralized routing controller has additional scaling challenges for modern data centers comprising thousands of switches, as a single software controller cannot react quickly to link and node failures. Thus, as data centers grew, one possible design was to evolve into a fully routed Layer-3 network, which requires a distributed routing protocol.

Border Gateway Protocol (BGP) is a Layer-3 protocol which was originally designed to interconnect autonomous Internet service providers (ISPs) in the global Internet. BGP has supported the Internet's unfettered growth for over 25 years. BGP is highly scalable, and supports large topologies and prefix scale compared to intra-domain protocols like OSPF and ISIS. BGP's support for hop-by-hop policy application based on communities makes it an ideal choice for implementing flexible routing policies. Additionally, BGP sessions run on top of TCP, a transport layer protocol that is used by many other network services. Such explicit peering sessions are easy to navigate and troubleshoot. Finally, BGP has the support of multiple mainstream vendors, and network engineers are familiar with BGP operation and configuration. Those reasons, among others, make BGP an attractive choice for data center routing.

BGP being a viable routing solution in the data center (DC) networks has been well known in the industry [11]. However, the details of a practical implementation of such a design have not been presented by any large-scale operator before. This paper presents a first-of-its-kind study that elucidates the details of the scalable design, software implementation, and operations. Based on our experience at Facebook, we show that BGP can form a robust routing substrate but it needs tight co-design across the data center topology, configuration, switch software, and DC-wide operational pipeline.

Data center network designers seek to provide reliable connectivity while supporting flexible and efficient operations. To accomplish that, we go beyond using BGP as a mere routing protocol. We start from the principles of configuration *uniformity* and operational *simplicity*, and create a baseline connectivity configuration (§2). Here, we group neighboring devices at the same level in the data center as a peer group and apply the same configurations on them. In addition, we employ a uniform AS numbering scheme that is reused across different data center fabrics, simplifying ASN management across data centers. We use hierarchical route summarization on all levels of the topology to scale to our data center sizes while ensuring forwarding tables in hardware are small. Our policy configuration (§3) is tightly integrated with our baseline connectivity configuration. Our policies ensure reliable communication using route propagation scopes and predefined backup paths for failures. They also allow us to maintain the network by seamlessly diverting traffic from problematic/faulty devices in a graceful fashion. Finally, they

[*] Work done while at Facebook. Authors contributed equally to this work.
[†] Currently works at ByteDance.

ensure services remain reachable even when an instance of the service gets added, removed, or migrated.

While BGP's capabilities make it an attractive choice for routing, past research has shown that BGP in the Internet suffers from convergence issues [33, 37], routing instabilities [32], and frequent misconfigurations [21, 36]. Since we control all routers in the data center, we have flexibility to tailor BGP to the data center which wouldn't be possible to achieve in the Internet. We show how we tackled common issues faced in the Internet by fine-tuning and optimizing BGP in the data center (§4). For instance, our routing design and predefined backup path policies ensure that under common link/switch failures, switches have alternate routing paths in the forwarding table and do not send out fabric-wide re-advertisements, thus avoiding BGP convergence issues.

To support the growing scale and evolving routing requirements, our switch-level BGP agent needs periodic updates to add new features, optimization, and bug fixes. To optimize this process, i.e., ensure fast frequent changes to the network infrastructure to support good route processing performance, we implemented an in-house BGP agent (§5). We keep the codebase simple and implement only the necessary protocol features required in our data center, but we do not deviate from the BGP RFCs [6–8]. The agent is multi-threaded to leverage multi-core CPU performance of modern switches, and leverages optimizations like batch processing and policy caches to improve policy execution performance.

To minimize impact on production traffic while achieving high release velocity for the BGP agent, we built our own testing and incremental deployment framework, consisting of unit testing, emulation and canary testing (§6.1). We use a multi-phase deployment pipeline to push changes to agent (§6.2). We find that our multi-phase BGP agent pushes ran for 52% of the time in a 12 month duration, highlighting the dynamic nature of the BGP agent in our data center.

In spite of our tight co-design, simplicity, and testing frameworks, network outages are unavoidable. On the operational side, we discuss some of the significant BGP-related network outages known as SEVs [38] that occurred over two years (§6.3)—these outages were either caused by incorrect policy configurations, bugs in the BGP agent software, or interoperability issues between different agent versions during the deployment of the new agent. Using our operational experience, we discuss current directions we are pursuing in extending policy verification and emulation testing to improve our operational framework, and changing our routing design to support weighted load-balancing to address load imbalances under maintenance/failures.

Contributions.

- We present our novel BGP routing design for the data center which leverages BGP to achieve reliable connectivity along with operational efficiency.
- We describe the routing policies used in our data center to enforce reliability, maintainability, scalability, and service reachability.
- We show how our data center routing design and policies overcome common problems faced by BGP in Internet.
- We present our BGP operational experience, including the benefits of our in-house BGP implementation and challenges of pushing BGP upgrades at high release velocity.

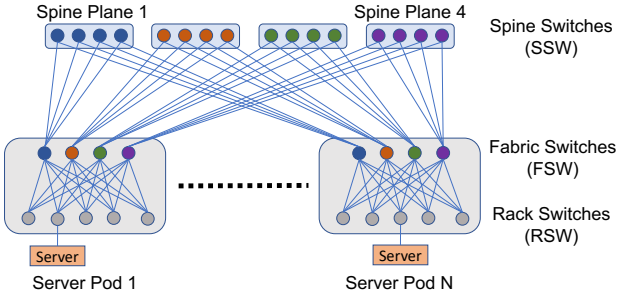

Figure 1: Data Center Fabric Architecture

2 Routing Design

Our original motivation in devising a routing design for our data center was to *build* our network quickly while keeping the routing design *scalable*. We sought to create a network that would provide high availability for our services. However, we expected failures to happen - hence, our routing design aimed to minimize the blast radius of those.

In the beginning, BGP was a better choice for our needs compared to a centralized SDN routing solution for a few reasons. First, we would have needed to build the SDN routing stack from scratch with particular consideration for scalability and reliability, thus, hindering our deployment pace. Simultaneously, BGP has been demonstrated to work well at scale; thus, we could rely on a BGP implementation running on third-party vendor devices. As our network evolved, we gradually transitioned to our custom hardware [18] and in-house BGP agent implementation. This transition would have been challenging to achieve without using a standardized routing solution. With BGP, both types of devices were able to cooperate in the same network seamlessly.

At the time, BGP was a better choice for us compared to the Interior Gateway protocols (IGP) like Open Shortest Path First (OSPF) [39] or Intermediate System to Intermediate System (ISIS) [25]. The scalability of IGPs at scale was unclear, and the IGPs did not provide the flexibility to control route propagation, making it harder to manage failure domains.

We used BGP as the sole protocol and did not pursue a hybrid BGP-IGP routing design as maintaining multiple protocols would add to the complexity of the routing solution. Our routing design builds on the eBGP (External BGP) peering model: Each switch is a BGP speaker and the neighboring BGP speakers are in different autonomous systems (AS). In this section, we provide an overview of our BGP-based routing design catered for our scalable data center fabric topology.

2.1 Topology Design

Application requirements evolve constantly, and our data center design must be capable of scaling out and handling additional demand in a seamless fashion. To this end, we adopt a modular data center fabric topology design [4], which is a collection of *server pods* interconnected by multiple parallel *spine planes*. We illustrate our topology in Figure 1.

A server pod is the smallest unit of deployment, and it has the following properties: (1) each pod can contain up to 48 server racks, and thus, up to 48 rack switches (RSWs), (2) each pod is serviced by up to 16 fabric switches (FSWs), and (3) each rack switch connects to all FSWs in a pod.

Multiple spine planes interconnect the pods. Each plane has multiple spine switches (SSW) connecting to FSWs using uniform high-bandwidth links (FSW-SSW). The number of spine planes corresponds to the number of FSWs in one pod. Each spine plane provides a set of disjoint end-to-end paths between a collection of server pods. This modular design enables us to scale server capacity and network bandwidth as needed—we can increase compute capacity by adding new server pods, while inter-pod bandwidth scales by adding new SSWs on planes.

2.2 Routing Design Principles

We employ two guiding design principles in our DC-wide BGP-based routing design: *uniformity* and *simplicity*. We realize these principles by tightly integrating routing design and configuration with the above topology design.

We strive to minimize the BGP feature set and establish repeatable configuration patterns and behaviors throughout the network. Our BGP configuration is homogeneous within each network tier (RSW, FSW, SSW). The devices serving in the same tier have the same configuration and policies, except for the originated prefixes and peer addresses.

We generate the network topology data and configuration which includes port-maps, IP addressing, BGP, and routing policy configurations for our switches irrespective of the underlying switch platforms. The abstract generic configurations are then translated into the target platform's configuration syntax by our automation software. This ensures that we can easily adapt to changing hardware capabilities in the data center. The details of our configuration management and platform-specific syntax generation can be found in Robotron [44].

2.3 BGP Peering & Load-Sharing

Peering. For uniformity and simplicity in configuration and operations, we treat the whole set of the BGP peers of the same adjacent tier (RSW/FSW/SSW) on a network switch as an *atomic group*, called peer group. Each data center switch connects to groups of devices on each adjacent tier. For example, a FSW aggregates a set of RSWs and has uplinks to multiple SSWs—this makes two distinct peer groups. All BGP peering sessions between adjoining device tiers—for example RSW↔FSW and FSW↔SSW—utilize the same protocol features, timers, and other parameters. Thus, all peers within a group operate in a uniform fashion.

We apply BGP configuration and routing policies on a peer group level. Individual BGP peer sessions belong to a peer group and do not have any additional configuration information beside the neighbor specification. This grouping helps us to simplify configuration and streamline processing of routing updates, as all peers in the same group have identical policies.

For peering, we use direct single-hop eBGP sessions with BGP NEXT_HOP attribute, set to the remote end of the point-to-point subnet. This makes the link usable for BGP routing purposes as soon as it is up. If there are multiple parallel links between the devices, we treat them as individual point-to-point Layer-3 subnets with corresponding BGP sessions. This design allows us to clearly associate BGP sessions with the corresponding network interfaces and simplifies RIB (routing information base) and FIB (forwarding information base) navigation, manipulation, and troubleshooting.

Load-Sharing. To support load-sharing of traffic along multiple paths in the data center, we use BGP with Equal Cost Multipath (ECMP) feature. Each switch forwards traffic equally among paths with equivalent attributes according to BGP best path selection and routing policy in effect. With the presence of multiple paths of equal cost, the vast majority of the switch FIB programming involves removing next hops (when failure occurs) or adding them back (when switch/link comes back up) in the existing ECMP groups. Updating ECMP groups in the FIB is a lightweight and simple operation.

We do not currently use weighted load-balancing inside our data centers for various reasons. Our fabric topology is highly symmetric with wide ECMP groups. We provision the bandwidth uniformly to maximize flexibility of dynamic service placement in the data center. Coupled with the design of our failure domains, this ensures sufficient capacity for services under most common failure scenarios. Moreover, WCMP [48] requires more hardware resources due to the replication of next-hops to perform weighted load-balancing. This does not align well with our goal of minimizing the FIB size requirements in hardware.

2.4 AS Numbering

Following the design principles of uniformity and simplicity, we design a uniform AS numbering scheme for the topology building blocks, such as server pods and spine planes. Our AS numbering scheme is canonical, i.e., the same AS numbers can be reused across data centers in the same fashion. For example, each SSW in the first spine plane in each data center would have the same AS number (e.g., AS 65001). Similarly, the RSWs and FSWs in every server pod of every data center share the same AS numbering structure. To accomplish this goal, we leverage BGP confederations [7]. A confederation

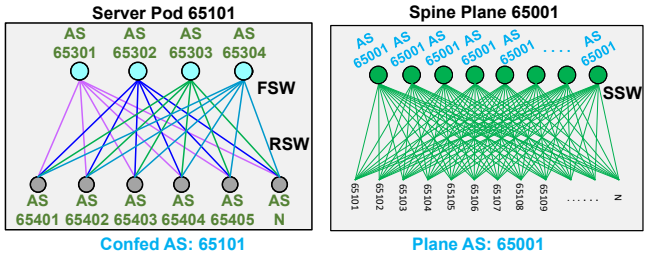

Figure 2: BGP Confederation and AS Numbering scheme for server pods and spine planes in the data center.

divides an AS into multiple sub-ASes such that the sub-ASes and internal paths between them are not visible to the BGP peers outside the confederation.

The uniformity facilitated by our use of confederations and the reusable ASNs (as opposed to a flat routing space) establishes well-structured AS_PATHs for policies and automation. This also helps operators to reason about a routing path easily by inspecting a given AS_PATH during troubleshooting. Inside the data center, we utilize the basic two-octet Private Use AS Numbers, which are sufficient for our design.

Server Pod. To create a reusable ASN structure for server pods—the most numerous building blocks inside our data center network—we implement each server pod as a BGP Confederation. Inside the pod, we allocate unique internal confederation-member ASNs for each FSW and each RSW. We then peer between the devices in a fashion similar to eBGP. The structure of these internal sub-AS numbers repeats within each pod. We assign a unique private AS number per pod (Pod ASN) within a data center as a Confederation Identifier ASN, which is how the pod presents itself to the data center spine and servers. The numbering pattern of unique pod Confederation Identifier ASNs repeats across different data centers. In Figure 2, in each pod, RSWs are numbered from ASN 65401 to N, FSWs are numbered from ASN 65301 to ASN 65304, and server pods are numbered as Confederation Identifier ASN 65101, 65102 and so on.

Spine Plane. Each spine plane in the data center fabric has its own unique (within the data center) private ASN assigned to all SSWs in it. In Figure 2, in the first spine plane, all SSWs are numbered ASN 65001. Similarly, all SSWs in the next spine plane would be numbered ASN 65002. This simplicity is possible because each SSW device operates independently from the others, serving as a member of the ECMP groups for the paths between pods. As no two SSWs directly peer with each other, they can use the same AS number. Reuse of ASNs acts as a *loop breaking mechanism*, ensuring that no route will traverse through multiple SSWs. The unique per-plane ASNs also aid us in simple identification of the operationally available planes for paths visible on rack switches.

2.5 Route Summarization

There are two principal categories of IP routes in our data centers: infrastructure and production. Infrastructure prefixes facilitate network device connectivity, management, and diagnostics. They carry relatively low traffic. In the event of a device or link failure, their reachability may be non-critical or can be supported by stretched paths. Production prefixes carry high-volume live traffic of our applications and must have continuous reachability in all partial failure scenarios, with optimal routing and sufficient capacity of all involved network paths and ECMP groups.

There are many routes in our data centers. To minimize the FIB size requirements in hardware and ensure lightweight control plane processing, we use hierarchical route summarization on all levels of the network topology. For production routes, we design IP addressing schemes which closely reflect the multi-level hierarchy. The RSWs aggregate the IP addresses of their servers and the FSWs aggregate the routes of their RSWs. For infrastructure routes, we have the following aggregates. Each device aggregates the IP addresses of all its interfaces, i.e. per-device aggregate. FSWs aggregate per-device RSW/FSW infrastructure routes into per-pod aggregates. And SSWs aggregate per-device SSW infrastructure routes into per-spine aggregates.

Depending on the route type and associated reachability criteria, switches advertise prefixes into BGP either unconditionally, or upon meeting the requirement of the minimal number of more-specific prefixes. The more-specific prefixes have a more limited propagation scope, while the coarser aggregates propagate farther on the network. For example, rack prefixes circulate only within their local pod, while pod-level aggregates propagate to the other pods and racks within the local data center fabric.

Hence, despite the sheer scale of our data center fabrics, our structured uniform route summarization ensures that the sizes of routing tables on switches are in low thousands of routes. Without route summarization, each router would have over hundred thousand routes, each route corresponding to the switches' interfaces and server racks. Our approach has many benefits: it allows us to use inexpensive commodity switch ASICs at the data center scale, enables fast and efficient transmission of routing updates, speeds up convergence (§5), and speeds up programming forwarding hardware.

3 Routing Policies

A key feature of BGP is the availability of well-defined attributes that influence the best path selection. Together with the ability to intercept route advertisements and admission at any hop and session, it allows us to control route propagation in the network with high precision. In this section, we review the use cases for routing policies in our data centers. We also describe how we configure the policies in BGP, while

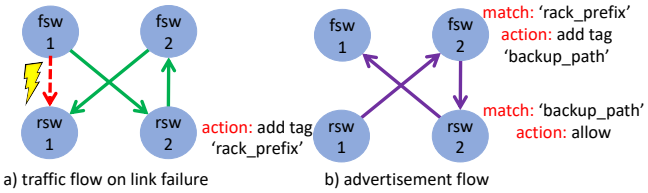

Figure 3: Example of predefined backup path policy.

realizing our principles of uniformity and simplicity.

3.1 Policy Goals

The Internet comprises multiple ASes owned by different ISPs. ISPs coordinate with each other to ensure routing objectives across the Internet. The routing policies mainly pertain to peering based on business relationships (customer-peer-provider) among different ISPs. However, since all the routers in our data centers are controlled by us, we do not have to worry about peering based on business relationships. Our data center routing design uses routing policies to ensure reliability, maintainability, scalability, and service reachability. We summarize these policy goals in Table 1.

Goal	Description
Reliability	Enforce route propagation scopes, predefine backup paths for failure
Maintainability	Isolate and remediate problematic nodes without disrupting traffic
Scalability	Enforce route summarization, avoid backup path explosion
Service reachability	Avoid service disruptions when instances of services are added, removed or migrated

Table 1: Policy goals

We use BGP Communities/tags to categorize prefixes into different types. We attach a particular *route type* community during prefix origination at the network device. This type *tag* persists with the prefix as it propagates. We perform matching on these communities to implement all our BGP policies in a uniform scalable fashion. We demonstrate how we use them with the examples in this section.

Reliability. Our routing policies allow us to safeguard the data center network stability. The BGP speakers only accept or advertise the routes they are supposed to exchange with their peers according to our overall data center routing design.

The BGP policies match on tags to enforce the intended route propagation scope. For example, in Fig. 3b, routes tagged with *rack_prefix* only propagate within the pod (i.e., not to the SSW layer).

Using BGP policies, we establish deterministic backup paths for different route types. This uniformly-applied procedure ensures the traffic will take predictable backup paths in the event of failures. We use backup path policies to protect FSW-RSW link failures. Consider the example in Fig. 3. We use tags to implement the backup policy, as shown in Fig. 3b. When *rsw*1 originates a route, it adds a *rack_prefix* tag. The $fsw2$ matches on that tag, adds another tag *backup_path*, and forwards the route to *rsw*2. *rsw*2 ensures routes tagged with *backup_path* are advertised to $fsw1$. When $fsw1$ detects the tag *backup_path*, it installs the backup route and adds the tag *completed_backup_path* (not shown in figure) which stops any unnecessary continued backup route propagation. In Fig. 3a, when the fsw1-rsw1 link fails, fsw1 will not send a new advertisement to its SSWs to signal the loss of connectivity to rsw1. Instead, BGP will reconverge to use the backup path ($fsw1 \rightarrow rsw2 \rightarrow fsw2 \rightarrow rsw1$) to reroute traffic through another RSW within the pod. And due to route summarization at the FSW (§2.5), these failures within a pod will not be visible to the SSWs and hence the routers outside the pod.

Backup paths are computed and distributed automatically as a part of BGP routing convergence. They are readily available when link failure happens. Typically, an FSW has multiple backup paths, of the same AS path length, to each RSW. When the direct fsw-rsw link fails, all of the backup paths will be used for ECMP.

In our network, each device has inbound (import) and outbound (export) match-action rules. Routes get advertised between two neighboring BGP speakers (X and Y) if they are allowed at both ends of the BGP session, i.e., they need to match an outbound rule of device X and an inbound rule of its neighboring device Y. This logic protects against routing misconfigurations on the peer. Additionally, on each device, routes that do not match on any of its rules are dropped to prevent unintended network behaviors.

Maintainability. In a data center, many events occur every hour and we expect things to fail. We see events like rack removal or addition, link flap or transceiver failure, network device reboot or software crash, software or configuration push failure, etc. Additionally, network devices are undergoing routine software upgrades and other maintenance operations. To avoid disruption of production traffic, we gracefully *drain* the device before maintenance—production traffic gets diverted from the device without incurring losses. For this, we define multiple distinct operational states for a network device. The state affects the route propagation logic through the device, as shown in Table 2. We change the routing policy configuration of a device based on its operational state. These configurations implement the logic specified in Table 2.

To gracefully take a device or a group of devices out of service (DRAINED) or put it back in service (LIVE), we apply policies corresponding to the current state on the peer groups. This initiates the new mode of operation across all affected BGP peers. Previous works [23] have used a multi-stage draining to gracefully drain traffic without disruptions. We also implement a multi-stage drain process with an interim WARM state. In the WARM state, we change the BGP policies to de-prioritize routes traversing through the device about to be drained. We also adjust the local and/or remote

State	Description
LIVE	The device is operating in active mode and carries full production traffic load.
DRAINED	The device is operating in passive mode. It doesn't carry any production traffic. Only the traffic to/from infrastructure/diagnostic prefixes may be allowed. Transiting infrastructure prefixes are lowered in priority.
WARM	The device is in process of changing states. It maintains full local RIB and FIB ready to support the production traffic, but adjusts route propagation and signals to avoid attracting live traffic.

Table 2: Operational states of a network switch

ECMP groups and ensure that network links do not become overloaded during the transition from LIVE to DRAINED state and vice-versa. Once BGP converges, all production traffic is rerouted to/from the device, and we can change the state of the network device again into the final state.

In the DRAINED state, BGP policies allow us to propagate only selected prefixes through the devices, and change route priorities. For example, this feature allows us to maintain reachability to the infrastructure (e.g., the switch's management plane) and advertise diagnostic prefixes through the devices under maintenance, while keeping the production traffic away from such devices.

Drain/undrain is a frequently used operation in data center maintenance. On average, we perform 242 drain and undrain operations daily. These operations take on average 36*s* to complete. The multi-stage state change ensures that there are no transient drops during this process.

Scalability. The routing policies allow us to implement and enforce our hierarchical route summarization design (§2.5). For example, in our network, our policy in FSW summarizes rack-level prefixes into pod-specific aggregates. They advertise these aggregates to the SSW tier. These policies also control propagation scopes for different route aggregation levels and minimize the routing table sizes in our switches.

The predefined backup paths also aid in scalability. These paths ensure our reaction to failures are deterministic and avoid triggering large-scale advertisements during failures which can cause BGP convergence problems.

To reduce policy processing overhead, we design all our policies to first apply rules which accept or deny the most number of prefixes. For example, in a drained state (Table 2), the FSW's outbound policy toward SSW first rejects routes marked to (i) avoid propagation to SSWs, or (ii) carry any production traffic. After that, it matches and lowers the priority of infrastructure routes before sending them to SSW. This design ensures we minimize the policy processing overhead on routes that will be dropped.

Service Reachability. One important goal of the data center network design is providing service reachability. A service should remain reachable even when an instance of the service gets added, removed, or migrated. As one of the mechanisms for providing service reachability in the network, we use Virtual IP addresses (VIPs). A particular service (e.g., DNS) may advertise a single VIP (serviced by multiple instances). In turn, anycast routing will provide reachability to one of the instances for traffic destined to the VIP.

To support flexible instance placement without compromising uniformity and simplicity, we create a VIP injector service in the form of a software library integrated with a service instance. The injector establishes a BGP session with the RSW and announces a route to signal the VIP reachability. When the service instance gets terminated, the injector sends a route withdrawal for the VIP. The routing policy on the RSW relays VIP routes to FSW after performing safety checks, such as ensuring that the injected VIP prefix conforms to the design intent. FSW's inbound policy from RSWs tags and sets different priorities for different VIP routes. This method allows for network-wide VIP priorities for active/backup applications.

By directly injecting VIP routes from services, we do not need to make changes to the network when creating/destroying service instances or adjusting active/backup service behaviors. That is, we do not need to change RSW configurations to start/stop advertising the VIPs or change VIP instance priorities. Our services integrate the injector library into their code (§5) and fully control when and how they want to update their VIPs.

3.2 Policy Configuration

For scalability and uniformity reasons, our policies primarily operate on BGP Communities and AS_PATH regular expression matches, and not on specific IP prefixes. To implement policy actions, we may accept or deny a route advertisement, or modify BGP attributes to manipulate the route's priority for best-path selection. We configure our routing policies on the BGP *peer group* level—therefore, any policy change is simultaneously applied to all peers in the group. Our reusable ASN design (§2.4) also allows us to use the same policies across our multiple data center fabrics.

The number of policy rules that exist between tiers of our data center network are relatively lightweight: 3-31 inbound rules (average 11 per session) and 1-23 outbound rules (average 12 per session). The majority of outbound policies tag routes to specify their propagation scope, and the majority of inbound policies perform admission control based on the route propagation tags and adjust LOCAL_PREF or AS_PATH length to influence route preference.

For the most numerous device role in our fleet, RSW, we keep the policy logic at the necessary minimum to reduce the need for periodic changes. To compensate for this, the FSWs in the pods have larger policies that offload some processing logic from the RSWs.

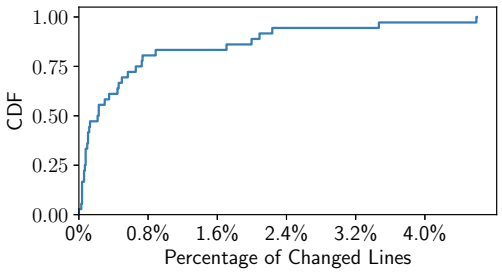

Figure 4: Network Policy Churn

For the commonly used BGP communities and other prefix attributes we maintain structured naming and numbering standards, suitable both for humans and automation tools. For the purposes of this paper, we elide the low-level details of our policy language syntax, objects, and rules.

3.3 Policy Churn

We maintain a global set of abstract policy templates and use them to generate individual switch configurations via an automated pipeline [44]. The routing policy used in our data center is fairly stable—we have made 40 commits to the routing policy templates over a period of three years. We show the cumulative distribution function (CDF) of the number of lines of changes made to the routing policy templates in Figure 4. We observe that most changes to the policy are incremental—80% of commits change less than 2% of policy lines. However, small changes to policy can have drastic service impacts, therefore they are always peer-reviewed and tested before production deployment (§6.2).

4 BGP in DCs versus the Internet

Multiple papers have studied issues with BGP convergence [33, 37], routing instabilities [32] and misconfigurations [21, 36], in the context of the Internet. This section summarizes these issues and describes how we address them in the data center context.

4.1 BGP Convergence

BGP convergence at the Internet-scale is a well-studied problem. Empirically, BGP can take minutes to converge. Labovitz et al. [33] proposed an upper bound on BGP convergence. During convergence, BGP attempts to explore all possible paths in a monotonically increasing order (in terms of AS_PATH length)—a behavior known as the path-hunting problem [2]. In the worst case, BGP convergence can require O(n!) messages, where n is the number of routers in the network. Using *MinRouteAdvertisementInterval* (MRAI) timer—minimum time between advertisements from a BGP peer for a particular prefix—BGP convergence can take O(n) x MRAI seconds. As mentioned in §3.1, our data centers experience many drain/undrain operations daily. These operations will cause BGP to reconverge, and this makes convergence a frequent event in our data centers.

To alleviate the BGP path-hunting problem, we define route propagation scopes and limit the set of backup paths that a BGP process needs to explore. For example, rack prefixes circulate only within a fabric pod; thus, an announcement or withdrawal of a rack prefix should only trigger a pod's reconvergence. To prevent slow convergence during network failures, we employ BGP policies that limit the AS_PATH that a prefix may carry, thus curbing the path-hunting problem.

Our topology design with broad path diversity (§2) and our predefined backup path policies (§3.1) ensure we only trigger fabric-wide re-advertisements when a particular router has lost all connections to its peers. Such events require tens to hundreds of links to fail, which is very unlikely. Thus, BGP convergence delays are infrequent in our data center. Since we want the network to converge as quickly as possible, we set the MRAI timer to 0. This could lead to increased advertisements (as each router would advertise any changes immediately), but our route propagation scopes ensure these advertisements do not affect the entire network.

4.2 Routing Instability

Routing instability is the rapid change of network reachability and topology information caused by pathological BGP updates. These pathological BGP updates lead to increasing CPU and memory utilization on routers, which can result in processing delays for legitimate updates, or router crashes; these can lead to delay in convergence or packet drops. Labovitz et al. [32] show that a significant fraction of routing updates on the Internet was pathological and do not reflect real network changes. With fine-grained control over the routing design, BGP configuration, and software implementation, we ensure that these pathological cases do not manifest in the data center. We describe the common pathological cases of routing instabilities and the solution in our data center to mitigate these cases in Table 3.

The most frequent pathological BGP message pattern reported by Labovitz et al. was WWDup. WWDup is a repeated transmission of BGP withdrawals for a prefix, which is unreachable. The cause of WWDup was stateless BGP implementation: a BGP router does not store any state regarding information advertised to its peers. The router would send a withdrawal to all its peers, irrespective of whether it had sent the same message. Internet-scale routers deal with millions of routes, so it was not practical to store each prefix's state for each peer. In data centers, BGP works at a much smaller scale (tens of thousands of prefixes) and typically has more memory resources. Thus, we can maintain the state of advertisements sent to each peer and check if a particular update needs sending. This feature eliminates pathological BGP withdrawals. Another class of pathological routing messages is AADup: a

Update Type	Description	DC Solution
WWDup	Repeated BGP withdrawals for unreachable prefixes	Store advertisement state in routers to suppress duplicate withdrawals
AADup	Implicit route withdrawal replaced by a duplicate of the same route	Store advertisement state in routers to suppress duplicate announcements
AADiff	Route is replaced by an alternate route	Fixed set of LOCAL_PREF values to avoid pathological metric changes
TUp/TDown	Prefix reachability oscillation	Monitor failures and automatically drain traffic from faulty devices

Table 3: Pathological BGP Updates found in the Internet by Labovitz et al. [32] and how we fix those in the data center

route is implicitly withdrawn and replaced by a duplicate. We stop AADups with our stateful BGP implementation as well.

The other types of BGP messages causing routing instabilities are AADiff (an alternate route replacing the old one) and TUp/TDown (prefix reachability oscillation). AADiffs happen due to MED (multi-exit discriminator) or LOCAL_PREF (local preference) oscillations in configurations that map these values dynamically from the IGP metric. As a result, when internal topology changes, BGP will announce advertisements to its peers with new MED/LOCAL_PREF values, even though the inter-domain BGP paths are unaffected. Hot-potato BGP routing [46] is a similar type of routing instability where the internal IGP cost affects the BGP best path decision. We use a fixed set of LOCAL_PREF values. Thus, any change in LOCAL_PREF indicates a legitimate update in the routing preference. We do not use MED. TUp and TDown come from the actual *oscillating* hardware failures. Our monitoring tools detect such failures and automatically reroute traffic from malfunctioning components to restore stability.

4.3 BGP Misconfigurations

Mahajan et al. [36] analyzed BGP misconfigurations in the Internet. They found that those affected up to 1% of the global prefixes each day. The misconfigurations increase the BGP control plane overhead with generation of pathological route updates. They can also lead to disruption of connectivity. The two types of BGP misconfigurations were the following. First, the origin misconfiguration is when a BGP router injects an incorrect prefix to the global BGP table. Second, the export misconfiguration is when an AS_PATH violates the routing policy for an ISP. The former can happen in the data center. For example, imagine a router advertising more specific /64 prefixes instead of the aggregated /56 prefix. A router could also inject a prefix from a different pod's address space, hijacking the traffic. The latter is also possible in the data center. A router may incorrectly advertise a prefix outside the prefix's intended propagation scope due to a bug in the routing policy. However, in practice, they are rare in our data center, as all our route advertisement configurations are automatically generated and verified. Since we have visibility and control over the data center, we can detect these issues with monitoring/auditing tools and promptly fix them. We further discuss the causes of misconfigurations reported by Mahajan et al. and demonstrate how we can avoid these in our architecture.

Incorrect BGP Attributes. One of the leading causes for incorrect prefix injection is a router advertising prefixes assuming that they will get filtered upstream. For reliability (§3), we add filters on both ends of the BGP session to ensure incorrect prefixes get filtered at either end. Errors can also happen due to wrong BGP communities, address typos, and inaccurate summarization statements. We use a centralized framework [44] to generate the configuration for individual routers from templates. Thus, we can catch errors from a single source, instead of dealing with separate routers.

Interactions with Other Protocols. A typical pattern is to use IGPs such as OSPF for intra-domain routing and configure redistribution to advertise the IGP routes into BGP for inter-domain routing. Configuring redistribution can end up announcing unintended routes. However, that is not a problem with a single-protocol design that we have.

Configuration Update Issues. Mahajan et al also observed cases when upon BGP restart, unexpected prefixes got advertised due to misconfigurations. For instance, in one scenario, configuration changes were not committed to persistent storage, and a router restarted using the old configuration. In our implementation, we ensure BGP does not advertise prefixes until after processing all configuration constructs. Each router has a configuration database, and we use transactions to update it consistently. We can afford slower upgrade mechanisms in the data center due to increased redundancy; routers in the Internet cannot be unavailable for long periods of time.

Thus, our BGP-based routing design tailored for the data center, that realizes the high-level DC-oriented goals of uniformity and simplicity, is able to overcome BGP problems common in the Internet.

5 Software Implementation

Like any other software, our BGP agent needs updates to add new features/optimizations, apply bug fixes, be compatible with other services, etc. Extending a third-party BGP implementation (by network vendors or open source [22, 30]) is not trivial and can add substantial complexity. Additionally, they have long development cycles for upstreaming or releasing their updates, and this affects our pace of innovation. To overcome those challenges, we develop an in-house BGP agent in C++ to run on our FBOSS [18] switches. In this section, we

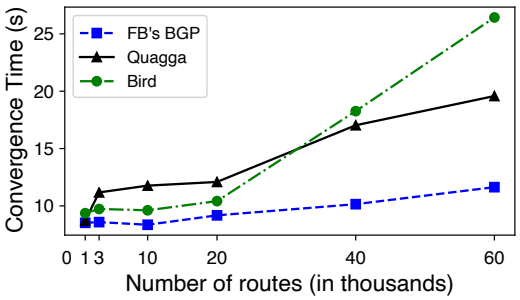

Figure 5: FB's BGP vs Quagga vs Bird (convergence time)

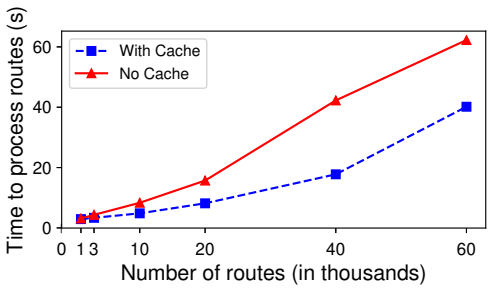

Figure 6: Impact of Policy Cache

present the main attributes of our agent.

Limited Feature Set. There are dozens of RFCs related to BGP features and extensions, especially to support routing for the Internet. Third-party implementations have support for many of these features and extensions. This increases the size of the agent codebase and its complexity due to interactions between various features. A large and complex codebase makes it harder for engineers to debug an issue and find a root cause, extend the codebase to add new features, or to refactor code to improve software quality. Therefore, the implementation of our BGP agent contains only the necessary protocol features required in our data center, but it does not deviate from the BGP RFCs [6–8]. Additionally, we only implement a small subset of matches and actions to implement our routing policies. We summarize the limited protocol features and match-action fields in Appendix A.

Multi-threading. Many BGP implementations are single-threaded (e.g., Quagga [30] and Bird [22]). Modern switches contain server-grade multi-core CPUs which allow us to run the BGP control plane at the scale of our data center. Our implementation employs multiple system threads, such as the peer thread and RIB thread, to leverage the multi-core CPU. The peer thread maintains the BGP state machine for each peer and handles parsing, serializing, sending, and receiving BGP messages over TCP sockets. The RIB thread maintains Loc-RIB (the main routing table), calculates the best path and multipaths for each route, and installs them to the switch hardware. To further maximize parallelism in the context of each system thread, we employ lightweight application threads `folly::fibers` [3]. These have low context-switching cost and execute small modular tasks in a cooperative manner. The fiber design is ideal for the peer thread as BGP session management is I/O intensive. To ensure lock-free property between system threads, we use message queues between fiber threads, running on the same or different systems threads.

To evaluate our BGP agent's performance, we compare it against two popular open source BGP stacks: Quagga [30] and Bird [22]. We run them on a single FSW device that is receiving both IPv4 and IPv6 routes from 24 SSWs. We compare their initial convergence time; this represents the time period between starting the BGP process to network convergence; this includes time for session establishment, and receiving and processing all route advertisements. In Fig. 5, we show the average over 5 runs. We observe that our BGP agent constantly outperforms other software and provides a speedup as high as 1.7*X* (Quagga) and 2.3*X* (Bird).

Policy. To improve policy execution performance, we added a few optimizations again building on our uniform design. Most of the peering sessions, from a device's point of view, are either towards uplink or downlink devices sharing the same inbound/outbound policies. Here, we made two observations: (1) prefixes learned from the same peer usually share the same BGP attributes, and (2) when routes are sent to the same type of peers (uplink or downlink peers), the same policy is applied for each peer separately. Peer groups help to avoid repetition in configuration, however, policies are still executed for routes sent/received from each peer separately. To leverage (1), we implemented *batching* in policy execution, where a set of prefixes and their shared BGP attributes are given as input to the policy engine. The policy engine performs the operation of *matching* the given BGP attributes and the prefixes sharing those attributes, and returning the accepted prefixes and their modified BGP attributes, based on the policy *action*. To avoid re-computations of (2), we introduced a policy cache, implemented in the form of an LRU (least recently used) cache containing <policy name, prefix, input BGP attributes, output BGP attributes> tuples. Once we apply the policy for routes to a peer and store that result in the policy cache, other peers in the same tier sharing the same policy can use the cached result and avoid re-execution of the policy. To show its impact, we run an experiment with and without the cache. We run them on a single FSW device that is sending IPv6 routes to 24 SSWs. We compare their time to process all route advertisements, which includes the time to apply outbound policy for each peer. In Fig. 6, we show the average over 5 runs. We observe that policy cache improves the time to process all routes by 1.2-2.4*X*.

Service Reachability. For flexible service reachability (§3), we want a service to inject routes for virtual IP addresses (VIPs) corresponding to the service directly to the RSW. However, current vendor BGP implementations commonly do not allow multiple peering sessions from the same peer address, which meant we would have to run a single injector service on every server and the applications on the server will need to interact with the injector to inject routes to the RSW. This becomes operationally difficult since application owners do

not have visibility to the injection process. There also exists a failure dependency as (i) applications need to monitor the health of the injector service to use it, and (ii) the injector needs to withdraw routes if the application fails. Instead, our BGP agent can support multiple sessions from the same peer address. Applications running on a server can directly initiate a BGP peer session with the BGP agent on the RSW and inject VIPs for service reachability. Thus, we do not have to maintain the cumbersome injector service to workaround the vendor BGP implementation constraint, and we also remove the application-injector dependency.

Instrumentation. Traditionally, operators used network management tools (e.g. SNMP [27], NETCONF [20], etc) to collect network statistics, like link load and packet loss ratio, to monitor the health of the network. These tools can also collect routing tables and a limited set of BGP peer events. However, extending these tools to collect new types of data—such as BGP convergence time, the number of application peers, etc—is not trivial. It requires modifications and standardization of the network management protocols. Facebook uses an in-house monitoring system called ODS [9,18]. Using a Thrift [1] management interface, operators can customize the type of statistics they want to monitor. Next, ODS collects these statistics into an event store. Finally, operators both manually and through an automated alerting tool, query and analyze the data to monitor their system. By integrating our BGP agent with this monitoring framework, we treat BGP like any other software. This allows us to collect fine-granular information on BGP's internal operation state, e.g. the number of peers established, the number of sent/received prefixes per peer, and other BGP statistics mentioned above. We monitor these data to detect and troubleshoot network outages (§6.3).

6 Testing and Deployment

The two main components we routinely test and update are configurations and the BGP agent implementation. These updates introduce new BGP features and optimizations, fix security issues, change BGP routing policies for improving reliability and efficiency. However, frequent updates to the control plane lead to increased risk of network outages in production due to new bugs or performance regressions. We want to ensure smooth network operations, avoid outages in the data center, and catch regressions as early as possible. Therefore, we developed continuous testing and deployment pipelines for quick and frequent rollouts to production.

6.1 Testing

Our testing pipeline comprises three major components - unit testing, emulation and canary testing.

Emulation is a useful testing framework for production networks. Similar to CrystalNet [35], we develop a BGP emulation framework for testing BGP agent, BGP configurations, and policy implementations, and modeling BGP behavior for the entire network. Emulation is used also for testing BGP behavior under failure scenarios – link flaps, link down, or BGP restart events. We also use emulation to test agent/config upgrade processes. The advantage of catching bugs in emulation is that they do not cause service disruptions in production. Emulation testing can greatly reduce developer's time and amount of physical testbed resources required. However, emulation cannot achieve high fidelity as it does not model the underlying switch software and hardware. Using emulation for BGP convergence regression is challenging as linux containers are considerably slower than hardware switches.

After successful emulation testing, we proceed to canary testing in production. We run a new version of the BGP agent/config on a small fraction of production switches called canaries. Canary testing allows us to run a new version of the agent/config in production settings to catch errors and gain confidence in the version before rolling out to production. We pick switches such that canaries can catch issues arising in production due to scale – e.g., delayed switch convergence. Canaries are used to test the following scenarios: (i) transitioning from old to new BGP agent/config (this occurs during deployment), (ii) transitioning from new to old BGP agent/config (when issues were found in production, we have to rollback to stable BGP version), and (iii) BGP graceful restart (which is an important feature for smooth deployment of BGP agent/config). Daily canaries are used to run new versions for longer periods (typically a day). Production monitoring systems will generate alerts for any abnormal behaviors. Canary testing helps us catch bugs not caught in emulation as it closely resembles BGP behavior in production, such as problems created by changes in underlying libraries.

6.2 Deployment

Once a change (agent/config) has been certified by our testing pipeline, we initiate the deployment phase of pushing the new agent/config to the switches. There is a trade-off between achieving high release velocity and maintaining overall reliability. We cannot simply switch off traffic across the data centers and upgrade the control plane in one-shot, as that would drastically impact services and our reliability requirements. Thus, we must ensure minimal network disruption while deploying the upgrades. This is to support quick and frequent BGP evolution in production. We devise a push plan which starts rolling out the upgrade gradually to ensure we can catch problems earlier in the deployment process.

Push Mechanisms. We classify upgrades in two classes: disruptive and non-disruptive, depending on if the upgrade affects existing forwarding state on the switch. Most upgrades in the data center are non-disruptive (performance optimizations, integration with other systems, etc.). To minimize routing instabilities during non-disruptive upgrades, we use BGP *graceful restart (GR)* [8]. When a switch is being upgraded, GR ensures that its peers do not delete existing routes for a

Phase	Specification
P1	Small number of RSWs in a random DC
P2	Small number of RSWs (> P1) in another random DC
P3	Small fraction of switches in all tiers in DC serving web traffic
P4	10% of switches across DCs (to account for site differences)
P5	20% of switches across DCs
P6	Global push to all switches

Table 4: Specification of the push phases

period of time during which the switch's BGP agent/config is upgraded. The switch then comes up, re-establishes the sessions with its peers and re-advertises routes. Since the upgrade is non-disruptive, the peers' forwarding state are unchanged. Without GR, the peers would think the switch is down, and withdraw routes through that switch, only to re-advertise them when the switch comes back up after the upgrade.

Disruptive upgrades (e.g., changes in policy affecting existing switch forwarding state) would trigger new advertisements/withdrawals to switches, and BGP re-convergence would occur subsequently. During this period, production traffic could be dropped or take longer paths causing increased latencies. Thus, if the binary or configuration change is disruptive, we drain (§3) and upgrade the device without impacting production traffic. Draining a device entails moving production traffic away from the device and reducing effective capacity in the network. Thus, we pool disruptive changes and upgrade the drained device at once instead of draining the device for each individual upgrade.

Push Phases. Our push plan comprises six phases P1-P6 performed sequentially to apply the upgrades to agent/config in production gradually. We describe the specification of the 6 phases in Table 4. In each phase, the push engine *randomly* selects a certain number of switches based on the phase's specification. After selection, the push engine upgrades these switches and restarts BGP on these switches. Our 6 push phases are to progressively increase scope of deployment with the last phase being the global push to all switches. P1-P5 can be construed as extensive testing phases: P1 and P2 modify a small number of rack switches to start the push. P3 is our first major deployment phase to all tiers in the topology. We choose a single data center which serves web traffic because our web applications have provisions such as load balancing to mitigate failures. Thus, failures in P3 have less impact to our services. To assess if our upgrade is safe in more diverse settings, P4 and P5 upgrade a significant fraction of our switches across different data center regions which serve different kinds of traffic workloads. Even if catastrophic outages occur during P4 or P5, we would still be able to achieve high-performance connectivity due to the in-built redundancy in the network topology and our backup path policies—switches running the stable BGP agent/config would re-converge quickly to reduce impact of the outage. Finally, in P6, we upgrade the rest of the switches in all data centers.

Push Monitoring. To detect problems during deployment, we have BGPMonitor, a scalable service to monitor all BGP speaking devices in the data center. All BGP speakers relay advertisements/withdrawals they receive to BGPMonitor. BGPMonitor then verifies the routes which are expected to be unchanged, e.g., routes for addresses originating from the switch. If we see route advertisements/withdrawals within the window of a non-disruptive upgrade, we stop the push and report the potential issue to an engineer, who analyzes the issue and determines if push can proceed. One of our outages was detected using BGPMonitor (§6.3).

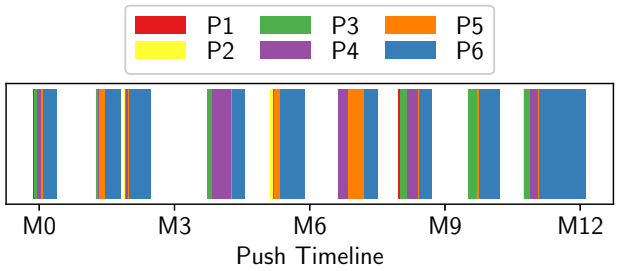

Figure 7: Timeline of BGP push phases over a year

Release	Total	P1	P2	P3	P4	P5	P6
7	0.57	0	0	0.28	0.20	0.82	0.56
8	0.43	0	0	0	0.12	0.13	0.54
9	0.51	0	0.94	0.95	1.12	0.25	0.49

Table 5: Push error percentages for the last 3 pushes for different push phases.

Push Results. Figure 7 shows the timeline of push releases over a 12 month period. We achieved 9 successful pushes of our BGP agent to production. On average, each push takes 2-3 weeks. Figure 7 highlights the high release velocity that we are able to achieve for BGP in our data center. We are able to fix performance and security issues as well as support new features at fast timescales. This also allows other applications, which leverage the BGP routing features, to innovate quickly. P6 is the most time-consuming phase of the push as it upgrades majority of the switches. We catch various errors in P1-P5, and thus, some of these phases can take longer (more than a day). Figure 7 also highlights the highly evolving nature of the data center. Our data centers are undergoing different changes to the BGP agent (adding support for BGP constructs, bug fixes, performance optimizations and security patches) for over 52% of the time in the 12 month duration.

Ideally, each phase should upgrade all the switches (100%). For instance, in one push, we fixed a security bug and we needed all the switches to run the fixed BGP agent version to ensure the network is not vulnerable. However, various devices were not reachable for a multitude of reasons. Devices are often brought down for various maintenance tasks, thus making them unreachable during push. Devices can also be experiencing hardware or power issues during the push phases. We cannot predict the downtime for such devices, and we

do not want to block the push indeterminately because of a small fraction of these devices. Hence, for each phase, we set a threshold of 99% on the number of devices we want to upgrade in each phase, i.e., 1% of the devices in our data centers could be running older BGP versions. We expect these devices will be upgraded in the next push phases. We report the push errors (number of devices which did not get upgraded) encountered in the last 3 pushes of Figure 7 in Table 5. We upgrade more than 99.43% of our data center in each push. These numbers indicate that there is always a small fraction of the data center which is undergoing maintenance. We try to upgrade these devices in the next push.

6.3 SEVs

Despite our testing and push pipeline, the scale and evolving nature of our data center's control plane (§6.2), the complexity of BGP and its interaction with other services (e.g. push, draining, etc), and the inevitable nature of human errors make network outages an unavoidable obstacle. In this section, we discuss some of the major routing-related Site EVents (SEVs) that occurred over a 2 year period. Errors and routing issues can arise due to (1) a recent change in configuration or BGP software, or (2) latent bugs in the code which are triggered due to a previously unseen scenario. We use multiple monitoring tools to detect anomalies in our network. These include (i) event data stores (ODS [9]) to log BGP statistics like downtime of BGP sessions at a switch, (ii) netsonar [34] to detect unreachable devices, and (iii) netnorad [10] to measure server-to-server packet loss ratio and network latency.

We experienced a total of 14 SEVs. These BGP-related SEVs were caused due to a combination of errors in its policy, software and interaction with other tools (e.g. push framework, draining framework, etc) in our data centers.

One set of SEVs were caused due to incomplete/incorrect deployment of policies. For example, one of the updates required both changing communities set in a policy at one tier and changing policies that act on those communities at another tier. It also required the first to be applied after the latter. However, during a push, policies were applied in an incorrect order. This created blackholes within the data center, degrading performance of multiple services.

Another set of SEVs were caused due to an error in BGP software. One SEV was caused by a bug in implementation of a feature called max-route limit that limits the number of prefixes received from a peer. The bug was that the max-route counter was getting incremented incorrectly for previously announced prefixes. This made BGP tear down multiple sessions, leading services to experience SLA violations.

We also experienced problems due to interactions between different versions of the BGP software. In one SEV, different versions were using different graceful restart parameters [8]. During graceful restart, the old version of BGP used stale paths for 30s. However, the new version deferred sending new routes for as long as 120s, waiting for receiving End-of-RIB from all peers. Hence, the old version purged stale paths learned from its peer before receiving them from the new version. This resulted in temporary traffic loss for $\sim$ 90s. BGPMonitor detected this outage during the push phases.

All these outages were resolved by rolling back to a previous stable version of BGP, followed by pushing a new fixed version in the next release cycle. Our design principles of uniformity and simplicity, while helpful, do not address issues such as software bugs and version incompatibilities, for which special care is needed. Our aim is to create a good testing framework to prevent these outages. We created the emulation platform during the later phases of our BGP development process and evolved ever since. As a follow-up to the aforementioned SEVs, we added new test cases to emulate those scenarios. As part of our ongoing work (§7), we are exploring ideas to further improve our testing pipeline.

7 Future Work

This section describes some of our ongoing work based on the gaps we have identified during our past years of data center network operations.

Policy Management. BGP supports a rich policy framework. The inbound and outbound policy is a decision tree with multiple rules capturing the policy designer's intent. Although the routing policies are uniform across tiers in our design, it is non-trivial to manage and reason about the full distributed policy set. Control plane verification tools [13, 15, 24, 40] verify policies by modeling device configurations. However, existing tools cannot scale to the size of our data centers, and they do not support such complex intent as flexible service reachability. Extending network verification to support our policy design at scale is an important future direction. Network synthesis tools [12, 16, 17, 19, 43] use high-level policy intents to produce policy-compliant configurations. Unfortunately, the policy intent language used by these tools cannot model all our policies (§2). Additionally, the configurations generated by them do not follow our design choices (§3). Extending network synthesis to support our BGP design and policies is also an ongoing direction we are pursuing.

Evolving Testing Framework. Policy verification tools assume the underlying software is error-free and homogeneous across devices. 8 of our SEVs occurred due to software errors. Existing tools cannot proactively detect such issues. To compensate, we use an emulation platform to detect control-plane errors before deployment. Some routing issues, like transient forwarding loops and black holes, materialize while deploying BGP configuration and software updates in a live network. Our deployment process monitoring (§6.2) demonstrates that the control plane is under constant churn. 10 of our SEVs were triggered while deploying changes. To address that, we are extending our emulation platform to mimic the deployment pipeline and validate the impact of various deployment strate-

gies. We are further exploring techniques to closely emulate our hardware switches and combined hardware/software failure scenarios. We are also extending our testing framework to include network protocol validation tools [45] and fuzz testing [31]. Protocol validation tools can ensure our BGP agent is RFC-compliant. Fuzz testing can make our BGP agent robust against invalid, unexpected, or random external BGP messages with well-defined failure handling.

Load-sharing under Failures. Over the past few years, we observe that hardware failures or drains can create load imbalance. For example, SSW's uplinks to the DC aggregation layer are not balanced when the failure of an SSW-FSW link (or SSW/FSW node) creates topology asymmetry in the spine plane. If one of an RSW's (say R) four upstream FSWs (say F) cannot reach one of its four SSWs, then F's SSWs would serve 1/4 of the traffic over 3 uplinks unlike the other 3 FSWs that serve 1/4 of the traffic over 4 uplinks. To balance traffic load across SSW's uplinks, R should reduce the traffic sent towards F from 1/4 to 3/15, and shift the remaining traffic to the other 3 FSWs. Although a centralized controller would be the most direct way to shift traffic to balance the load, we are considering an approach like Weighted ECMP [48] to leverage our BGP-based routing design.

8 Related Work

Routing in Data Center. There are different designs for large-scale data center routing, some are based on BGP while others use a centralized software-defined networking (SDN) design. An alternative BGP-based routing design for data centers is described in RFC7938 [11]. Our design differs in a few significant ways. One difference is the use of BGP Confederations for pods (called "clusters" in RFC7938). That enables our design to stick with the two-octet private ASN numbering space and reuse the same ASN on all rack switches. Thus, we also do not use the "AllowAS In" BGP feature in our design and maintain native BGP loop prevention. The second difference is our extensive use of route summarization in order to keep the routing tables small and improve the stability and convergence speed of the distributed system. The RFC7938 proposes keeping full routing visibility for all prefixes on all rack switches. Another major difference is our extensive use of the routing policies to implement strict adherence to the reachability and reliability goals, realize the different operational states of the devices, establish pre-determined network backup paths, and provide means for host-signaled traffic engineering, such as primary/secondary path selection for VIPs.

Singh et. al [42] showed that Google uses an SDN-based design for its data center network routing. It has a central route controller to collect and distribute link state information over a reliable out-of-band Control Plane Network (CPN) that runs a custom IGP for topology state distribution. Their reasoning behind building a centralized routing plane from scratch was to be able to leverage the unique characteristics and homogeneity of their network which comprises custom hardware. We decided to use a decentralized BGP approach to take advantage of BGP's extensive policy control, scalability, third-party vendor support, operator familiarity, etc.

Operational Framework. CrystalNet [35] is a cloud-scale, high-fidelity network emulator used by Microsoft to proactively validate all network operations before rolling out to production. We use an in-house emulation framework to easily integrate with our monitoring tools and deployment pipelines. Janus [14] is a software and hardware update planner that uses operator specified risks to estimate and choose the push plan with minimal availability and performance impact on customers. We use a framework similar to Janus for our maintenance planning, which includes disruptive BGP agent/config push. Govindan et. al [26] conducted detailed analysis of over 100 high-impact network failure events at Google. They discovered that a large number of failures happened when a network management operation was in progress. Motivated by these failures, they proposed certain design principles for high availability, e.g. continuously monitor the network, use in-house testing and rollout procedures, make (network) update the common case, etc. We acknowledge these principles; they have always been a part of our operational workflow.

BGP at Edge. EdgeFabric [41] and Espresso [47] also run BGP at scale. However, they are deployed at the edge for the purpose of CDN traffic engineering. They are both designed by content providers to overcome challenges with BGP when dealing with large traffic volumes. They have centralized control over routing while retaining BGP as the interface to peers. They control which PoP and/or path traffic to a customer should choose as a function of path performance.

9 Conclusion

This paper presents our experience operating BGP in large-scale data centers. Our design follows the principles of *uniformity* and *simplicity*, and it espouses tight integration between the data center topology, configuration, switch software, and DC-wide operational pipeline. We show how we realize these principles and enable BGP to operate efficiently at scale. Nevertheless, our system is a work in progress. We describe some major operational issues we faced and how these are informing our routing evolution.

Acknowledgments. We thank many Facebook colleagues who have contributed to this work over the years and toward this paper. These include Allwyn Carvalho, Tian Fang, Jason Wilson, Hany Morsy, Mithun Aditya Muruganandam, Pavan Patil, Neil Spring, Srikanth Sundaresan, Sunil Khaunte, Omar Baldonado, and many others. We also thank the anonymous reviewers for their insightful comments. This work is supported by the National Science Foundation grants CNS-1637516 and CNS-1763512.

References

[1] Apache Thrift. http://thrift.apache.org/.

[2] BGP Path Hunting. https://paul.jakma.org/2020/01/21/bgp-path-hunting/.

[3] folly::fibers. https://github.com/facebook/folly/tree/master/folly/fibers.

[4] 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/.

[5] Standard for local and metropolitan area networks: Media access control (mac) bridges. *IEEE Std 802.1D-1990*, pages 1–176, 1991.

[6] A Border Gateway Protocol 4 (BGP-4). https://tools.ietf.org/html/rfc4271, 2006.

[7] Autonomous System Confederations for BGP. https://tools.ietf.org/html/rfc5065, 2007.

[8] Graceful Restart Mechanism for BGP. https://tools.ietf.org/html/rfc4724, 2007.

[9] Facebook's Top Open Data Problems. https://research.fb.com/blog/2014/10/facebook-s-top-open-data-problems/, 2014.

[10] NetNORAD: Troubleshooting networks via end-to-end probing. https://engineering.fb.com/core-data/netnorad-troubleshooting-networks-via-end-to-end-probing/, 2016.

[11] Use of BGP for routing in large-scale data centers. https://tools.ietf.org/html/rfc7938, 2016.

[12] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Aed: incrementally synthesizing policy-compliant and manageable configurations. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 482–495, 2020.

[13] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast and general network verification. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[14] Omid Alipourfard, Jiaqi Gao, Jeremie Koenig, Chris Harshaw, Amin Vahdat, and Minlan Yu. Risk based planning of network changes in evolving data centers. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 414–429. ACM, 2019.

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

[16] 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 *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.

[17] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 437–451, 2017.

[18] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 342–356. ACM, 2018.

[19] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-wide configuration synthesis. In *International Conference on Computer Aided Verification*, pages 261–281. Springer, 2017.

[20] Rob Enns, Martin Bjorklund, and Juergen Schoenwaelder. Netconf configuration protocol. Technical report, RFC 4741, December, 2006.

[21] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation - Volume 2*, NSDI'05, page 43–56, USA, 2005. USENIX Association.

[22] Ondrej Filip, Libor Forst, Pavel Machek, Martin Mares, and Ondrej Zajicek. The bird internet routing daemon project. *Internet: www. bird. network. cz*, 2011.

[23] P. Francois, O. Bonaventure, B. Decraene, and P. Coste. Avoiding disruptions during maintenance operations on bgp sessions. *IEEE Transactions on Network and Service Management*, 4(3):1–11, 2007.

[24] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 300–313, 2016.

[25] Les Ginsberg, Stefano Previdi, and Mach Chen. IS-IS Extensions for Advertising Router Information. RFC 7981, October 2016.

[26] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 58–72. ACM, 2016.

[27] David Harrington, Randy Presuhn, and Bert Wijnen. Rfc3411: An architecture for describing simple network management protocol (snmp) management frameworks, 2002.

[28] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery.

[29] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally deployed software defined wan. In *Proceedings of the ACM SIGCOMM Conference*, Hong Kong, China, 2013.

[30] Paul Jakma and David Lamparter. Introduction to the quagga routing suite. *IEEE Network*, 28(2):42–48, 2014.

[31] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[32] C. Labovitz, G. R. Malan, and F. Jahanian. Origins of internet routing instability. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, volume 1, pages 218–226 vol.1, 1999.

[33] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed internet routing convergence. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, page 175–187, New York, NY, USA, 2000. Association for Computing Machinery.

[34] Jose Leitao and David Rothera. Dr NMS or: How facebook learned to stop worrying and love the network. Dublin, May 2015. USENIX Association.

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

[36] Ratul Mahajan, David Wetherall, and Tom Anderson. Understanding bgp misconfiguration. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, page 3–16, New York, NY, USA, 2002. Association for Computing Machinery.

[37] Zhuoqing Morley Mao, Ramesh Govindan, George Varghese, and Randy H. Katz. Route flap damping exacerbates internet routing convergence. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, page 221–233, New York, NY, USA, 2002. Association for Computing Machinery.

[38] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, pages 393–407. ACM, 2018.

[39] John Moy. Ospf version 2. STD 54, RFC Editor, April 1998. http://www.rfc-editor.org/rfc/rfc2328.txt.

[40] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, P Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. *arXiv preprint arXiv:1911.02128*, 2019.

[41] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 418–431. ACM, 2017.

[42] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[43] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Synthesis of fault-tolerant distributed router configurations. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(1):1–26, 2018.

[44] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page

426–439, New York, NY, USA, 2016. Association for Computing Machinery.

[45] Keysight Technologies. Ixanvl™–automated network validation library.

[46] Renata Teixeira, Aman Shaikh, Tim Griffin, and Jennifer Rexford. Dynamics of hot-potato routing in ip networks. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, page 307–319, New York, NY, USA, 2004. Association for Computing Machinery.

[47] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 432–445. ACM, 2017.

[48] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. Wcmp: Weighted cost multipathing for improved fairness in data centers. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

A BGP Agent Features

As mentioned in §5, our BGP agent contains only those necessary protocol features that are required in our data center. We summarize the different agent features in Table 6. Additionally, we only implement a small subset of matches and actions mentioned in Table 7 to implement our routing policies specified in §3.

	Feature	Description	Rationale
Core Feature	eBGP	Establish external BGP session	To Exchange and forward route updates
	Confederations	Divide an AS into multiple sub ASes	To use the same private ASNs within a pod
	eBGP Multipath	Select and program multipath	To implement ECMP-based load-sharing
	IPv4/IPv6 Addresses	Support IPv4/IPv6 route exchange	To enable dual-stack
	Route Origination	Send update for IP prefixes assigned to a switch	
	Route Aggregation	Send update for less-specific IP prefixes aggregating (summarizing) more-specific routes	To minimize number of route updates
	Remove Private AS	Remove Private ASNs within AS-PATH	To reuse private ASNs.
	In/Out-bound Policy	Support BGP policies specified in §2	
	Dynamic Peer	Accept BGP session initiation from a range of peer addresses	To allow VIP injection from any server
Operational Feature	Graceful Restart	Wait for small graceful time period before removing routes	To reduce network churn
	Link Fail Detection	Fast BGP session termination upon link failure	To converge faster
	Propagation Delay	Delay advertisements of new routes	To wait for convergence before receiving traffic
	FIB Acknowledgement	Advertise routes after installation to hardware	To avoid blackholes if peer converges before us
	Max-route-limit	Limit number of prefixes received from a peer	To disallow unexpected volume of updates
	Peer Groups	Define and reuse peer configurations for multiple peers	To make configuration compact

Table 6: Core and operational BGP features

Match Fields	Action Fields
as-path	add/delete/set as-path
community-list	add/delete/set community
origin	set origin
local preference	inc/dec/set local preference
as-path-length	permit
prefix-list	deny

Table 7: Policy match-action fields