Practical Intent-driven Routing Configuration Synthesis

Paper #452, 12 pages body, 15 pages total

Abstract

Configuration of production datacenters is challenging due to their scale (many switches), complexity (specific policy requirements), and dynamism (need for many configuration changes). This paper introduces Aura, a production-level synthesis system for datacenter routing policies. It consists of a high-level language, called RPL, that expresses the desired behavior and a compiler that automatically generates switch configurations. Unlike existing approaches, which generate full network configuration for a static policy, Aura is built to support frequent policy and network changes. It generates and deploys multiple parallel policy collections, in a way that supports smooth transitions between them without disrupting live production traffic. Aura has been deployed for over two years in Meta datacenters and has greatly improved our management efficiency. We also share our operational requirements and experiences, which can potentially inspire future research.

1 Introduction

Stable and efficient routing in large data centers is crucial for many online service providers. Routing misconfiguration can lead to packet drops, traffic black holes, performance degradation, and service downtime [2, 3, 8, 19]. Traditionally at Meta datacenters, routing configuration relied on human operator expertise to manually translate high-level routing policies into low-level switch configurations. This approach has two key problems.

First, manual generation of policies is often error-prone, especially with the enormous increase in scale (thousands of switches across multiple data centers), complexity (policies that describe specifications unique to a network), and dynamism (configuration changes to accommodate failures or maintenance of switches) of modern datacenters [9, 12].

Second, manually crafting configurations for datacenters is a time-consuming process. Earlier data centers at Meta were uniform and the similar configurations could be used to provision new data centers. However, there is a need to support diverse topologies required for various AI applications or existing topologies that are modified to accommodate resource shortages, caused by supply chain bottlenecks. With the need to support diverse topologies, existing configurations can no longer be reused and each new topology implies long process of manual configuration.

Recently, researchers have proposed configuration synthesis [6, 7, 11], which automatically generates switch configurations based on high-level policies. These systems usually provide a declarative language for operators to define the intended routing policies and then automatically synthesize low-level routing configurations, which implement these policies. While these solutions work well in principle, there are still a few challenges that are not handled by one-shot automated configuration synthesis.

Challenge 1: Handling dynamic configurations. Configuration changes are frequent in networks, because of many dynamic events. The dynamism is driven by different business objectives (shifting services from one data center to another), making network operations more efficient (e.g., smarter load balancing or more redundancy to failure), safely testing new protocols, or even performing regular routine maintenance of switches. Configuration synthesis should be able to generate configurations that natively handle dynamism.

Challenge 2: Expressing conditional policies. Current declarative languages express routing policies in a way that is not aligned with the realities of a production network. First, they treat each switch as live and ready to serve traffic. Yet, in large-scale data centers, switches can be in different operational states at the same time, and thus we need to be able to express routing policies that depend on these states. Second, current declarative languages specify all switches at a fixed granularity (e.g., one specific switch [6] or all switches in the specific role [7]). However, operational needs require specifications at a flexible granularity. Some intents in high-level policies may require specific switch or set of switches in one location, while others may require all switches in a given role.

Challenge 3: Reconfigurations at scale. Existing brownfield migration systems, plan out the configuration change

without disrupting production traffic by creating intermediate configurations that would help transitioning the network from old configuration to a new one [15, 20]. Although such techniques provide a safe, non-disruptive mechanism to change configurations, they can be expensive to carry out migrations. Our experiences show that deploying a new configuration in switches often takes a much longer time (minutes to hours) than computing the configuration (e.g., seconds), primarily because of the scale of the network. Given this constraint, migrating configurations via transitioning would take a very long time, hindering network operations.

In this paper, we introduce our configuration synthesis system Aura[1], which supports flexible granularity of policy intents, conditional intents and scalable synthesis to BGP configurations, to smoothly aid network dynamics. Our contributions are as follows:

- We propose a novel configuration synthesis approach, which pre-compiles a set of possible paths for the datacenter, called "base paths", given a set of high-level policies. Our insight is that given datacenter regularity and symmetry, any path can be expressed as one of a small set of base paths. Network operators can use these base paths to support dynamic configuration.
- We define a new declarative language called RPL (routing policy language), which allows operators to define policy intents with flexible switch granularities and activation conditions, based on switch states.
- Aura leverages configuration staging and uses labels to activate them. When there is a need to change configurations, routes are announced with appropriate labels (e.g., BGP community tags). Switches on receiving the labels, check the appropriate configuration that match the conditions and activate the configuration. This minimizes the need to reconfigure the network every time there is a need to change the configuration.

In addition to our intellectual contributions, Aura has been partially deployed in Meta datacenters for over two years, compiling all policies on hundreds of thousands of switches daily. To the best of our knowledge, we are the first to share operational experiences in building and deploying a configuration synthesis system. Our work unveils a unique set of challenges to academia, which can inspire future research.

2 Background and Challenges

As discussed earlier, large-scale datacenters need automated configuration synthesis. Even with fully automated configuration synthesis, carrying out a data center-wide policy change still requires careful planning at every step, to ensure that the network remains operational throughout the change. We need to gradually roll out a policy change, with minimum

[1]We chose the name Aura, because it represents the essence (of an individual).

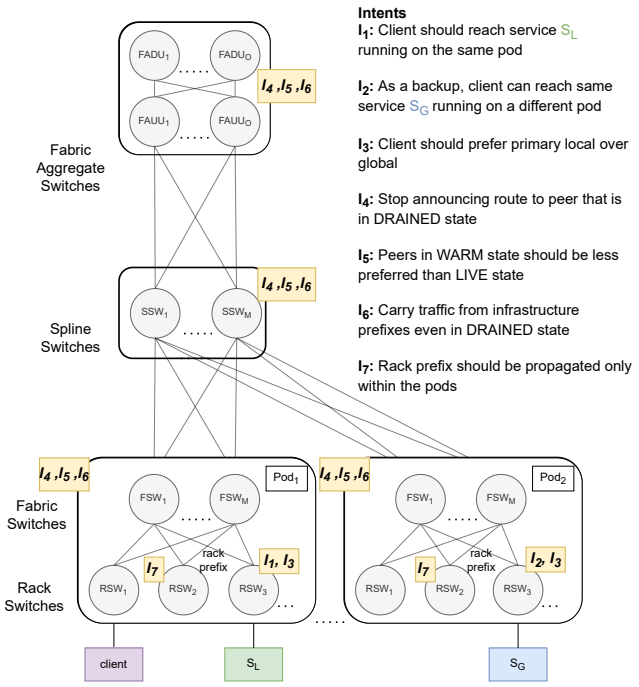


Figure 1: Data Center Topology with sample intents.

disruptions and safe fallback mechanisms. And we need to support such rollout across hundreds of thousands of switches, in different states of readiness. In this work, we stress one overlooked goal of configuration synthesis: producing configuration changes that lead to the shortest time to complete the reconfiguration, and making the process automated, non-disruptive to production traffic and with minimal operator burden. In this section, we start by describing our data center topology and routing intents. We then discuss the operational challenges of configuration synthesis in large data centers.

2.1 Background

We use Figure 1 to illustrate our production network topology [5, 24], a few sample intents (I_1 to I_7) used in our network, and the remaining open challenges, which our work addresses.

Data Center Topology: Figure 1 shows one example topology that is widely used in our network. It consists of a hierarchy of four layersc with thousands of switches at each layer. Switches at the same layer share the same *switch role*. The servers are connected to leaf rack switches (RSW). RSWs are connected to fabric switches (FSW). RSWs and FSWs are grouped into pods. Spine switches (SSW) represent the intermediate layer that connects pods. Data centers are distributed over multiple buildings and are interconnected via the Fabric Aggregation (FA) layer. The FA consists of two layers: FAUU (uplink) and FADU (downlink). FAUU connects to data-center-external networks, while FADU aggregates downstream-data-center networks. This topology enables several disjoint end-to-end paths between any two server racks

for failure resilience. Other topologies in Meta datacenters share the same properties with different numbers of layers and switch roles. Other large production data centers exhibit similar symmetry of roles and hierarchy of switch roles [21], and likely face similar challenges to configuration synthesis.

Routing: We use the Border Gateway Protocol (BGP) to disseminate routing information through the network and provide connectivity to end servers [4]. We configure BGP policies to manage how this information is shared across the network and to control traffic flow objectives, such as traffic load-balancing, redundancy, and path preference. Meta data centers are also experimenting with our own routing protocols such as OpenR [13] that is being rolled out into parts of the data center.

Intents, policies, and configurations: We define an *intent* as a high-level description of a routing goal, e.g., all rack prefixes should be propagated within pods. Figure 1 shows seven intent examples to achieve different goals in traffic control. I_1, I_2 and I_3 define reachability goals for a client to a service, I_4 and I_5 help in managing networks during maintenance, I_6 is an exception to I_4 that aims to provide reliability under failure, and I_7 confines the propagation scope of a route. We will elaborate more about these intents from §2.2 to §2.4. We define a *policy* to be a collection of intents, e.g., a combination of intents I_1 to I_7. At Meta, there are many data centers, and we define a collection of policies known as *configurations* for every data center. The synthesis process starts with a policy specified by a domain-specific language and produces a set of switch configurations.

2.2 Handling Dynamic Configurations

Configuration changes are frequent in production and often impose a high operational burden to ensure live production traffic is not affected. Previous synthesis systems focus on generating one snapshot of the entire network's configurations [6, 7]. When changes happen, they have to rerun the entire synthesis and generate another snapshot. There are three common scenarios that necessitate configuration changes:

Intent changes: Intents describe high-level objectives of reachability, aggregation, and route propagation (see [4] for routing objectives). They can change due to various business needs – a service migrating from one data center to another, a better load balancing strategy, a more resilient failure recovery, etc. Consider intent I_1; if the local service S_L is moving to a different pod, then I_1 no longer holds and should be removed. To support changes between existing intent collections, we need a synthesis approach that can generate multiple configurations in a switch and control when each would activate. The same approach can add new policies by enriching existing switches with new, inactive configurations. New configurations can then be activated gradually throughout the production network.

Policy implementation changes: We can implement intents expressed by operators in different ways, by using different protocols, or different mechanisms in one protocol. For example, in BGP, one can carefully set MED values, IGP values, or local preference values to achieve the same intents.In our data centers, we are exploring alternative routing protocols for better scalability, e.g. our home-grown intra-domain routing OpenR [13]. Thus, the same network-wide intents can be supported in OpenR with a completely different set of configurations compared to BGP. To facilitate testing, we need to gracefully roll out OpenR configurations and replace old BGP configurations. Such evolution requires a large amount of changes to configurations on all switches. To minimize impact to business, we need a way to gracefully migrate between configurations, and even roll back changes should they prove inefficient.

Switch state changes: Switches constantly undergo changes due to failures, new builds, and regular maintenance. In all of these scenarios, we need to gracefully remove the impacted switches from serving traffic, to minimize disruptions to services. Switch state changes are common in our production network. In the month of August 2022, there were about 745K drain events with about 8K drain events on average per day.

2.3 Expressing Conditional Policies

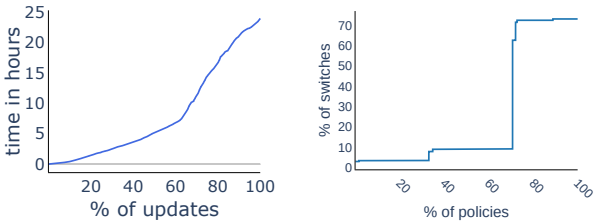
Conditional policies require different configurations for a given switch, depending on network conditions. There are three common classes of network conditions:

Intents that apply to a subset of switches: Operators need the flexibility to define groups of switches *where* a policy intent is applied. Some intents should be applied globally, while others may be relevant only for a specific service in a particular region to have customized routing solution. It is often needed during migration or deployment to accomodate the changing capacity. For example, intents I_1 to I_3 specify preference for local service S_L over global service S_G only for RSWs in Pod_1 and Pod_2, whereas intent I_7 applies to all RSWs and restricts rack prefixes to pods. This means our intent specification and synthesis should support conditions that define groups of switches at different topological granularity.

Intents that apply to switches in a certain state: Data center topologies support multiple paths between any pair of server racks. Path selection depends on switch states. We define three operational states for each switch: LIVE, DRAINED, and WARM. A LIVE switch is in operation; it allows all traffic and announcements to go through. Conversely, a DRAINED switch is brought down for maintenance; it should not carry any live production traffic. A switch being drained goes through an intermediate WARM state when prefixes are gradually removed from its announcements. In this state, the switch could carry traffic for any prefix if the prefix does not have other paths involving only LIVE switches. Switch configurations should suppress announce-

ments from DRAINED switches (reflected in intent I_4), and favor announcements from LIVE switches over those from WARM switches (reflected in intent I_5). To keep the production network operational, operators need a synthesis approach, which generates multiple configurations, for different switch states. Only one configuration is active at the time at a given switch, depending on the current state of the switch. For instance, intent I_4 specifies to stop announcing a route to peers in DRAINED states and intent I_5 specifies that we prefer peers in LIVE states than those in WARM states.

Exceptions in failure scenarios: Network operators need to specify exceptions to their policies for failure resilience. For instance, as per intent I_4, DRAINED switches do not carry any live traffic. However, an exception to this intent is I_6, which requires DRAINED switches to carry traffic towards infrastructure prefixes (e.g., those prefixes belonging to the management plane). Intent specification language needs to support exceptions for critical prefixes (e.g., infrastructure prefixes), and our synthesis process must generate corresponding configurations that treat those specific prefixes differently from others.



(a) Configuration update times. (b) Reconfigured switches.

Figure 2: Switch reconfiguration metrics.

2.4 Reconfigurations At Scale

It is challenging to support dynamic policies and conditional policies at production scale. Switch reconfiguration in a live production network is expensive, because it typically requires transferring away all the services that use the network and draining all routes from the switch. After these actions, we can bring down the switch, change the configurations, and then bring up the switch again. Finally, the switch would be tested before reintroducing the routes that it carried before reconfiguration. Typically, not all switches in the data center are reconfigured at the same time. Instead, reconfiguration is achieved in a phased manner, where only a portion of the network is reconfigured at a time. This process could take many weeks to update all switches, given the scale of a production network, as well as the complexity of the phased deployment [9]. Figure 2(a), shows CDF of configuration update times for policies that were changed at Meta in the last three months of 2021. The median reconfiguration time per switch is 5.2 hours

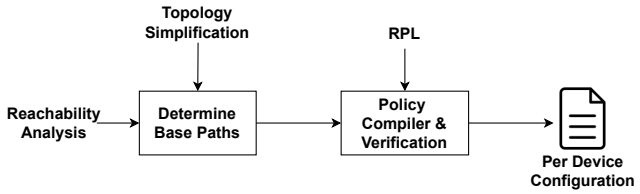


Figure 3: Aura architecture.

Changing policies (e.g., switching between collections of intents) also has a large footprint, that is, it involves the configuration of many switches. Figure 2(b) shows the percentage of switches in our network that required configuration update for different policy changes. An average policy change requires configuration of at least 25.6% of switches. Moreover, the time and complexity taken for configuring switches differ depending on a switch's role. On the one hand, changing O(10K) number of RSWs can take much longer than reconfiguring O(5K) of FSWs. On the other hand, aggregate switches carry a larger portion of traffic, so their changes should be planned more carefully. Reconfigurations are only going to get longer as we typically double the number of switches every five years [9].

3 Aura Design

To address the outlined challenges, we present Aura in Figure 3. The goal of Aura is to allow network operators to express flexible policies that are capable of handling various network reconfiguration scenarios at scale while minimizing disruption to production traffic. Aura handles the challenges outlined in §2 as follows:

- To support multiple configurations in a switch (as seen in § 2.2), Aura uses a set of base paths to pre-configure the network. Base paths are a collection of paths across different types of switches, which have the property that any propagation path in the network can be expressed as the base path or a sub-path of the base path. In §3.1, we explain how Aura uses topological features such as symmetry and hierarchy, along with reachability requirements to determine the set of base paths.
- To handle dynamic policies (as seen in § 2.2), Aura uses a labeling mechanism. If we are synthesizing into BGP configurations, Aura generates configurations that match on dedicated community tags and maps them to a corresponding policy. This results in multiple configurations defined in a single configuration file. In §3.2, Aura uses dedicated community tags that are attached by every switch to indicate its current state. Together with staging, this behaves as if there is a change between an old and a new configuration without the need for switch reconfiguration. Minimizing reconfigurations helps in supporting changes in policies at scale (as seen in §2.4).
- To express conditional policies (as seen in § 2.3), we intro-

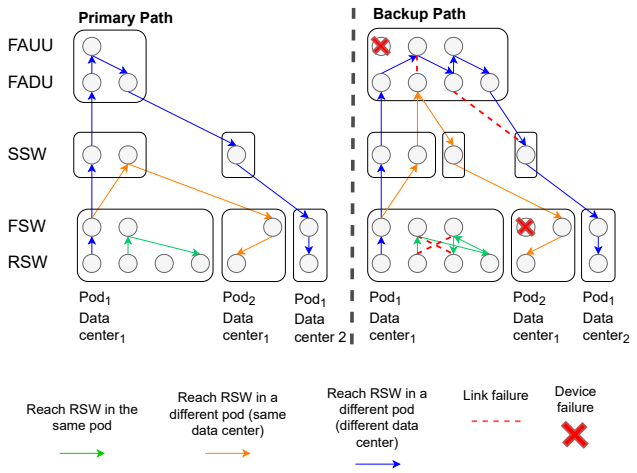


Figure 4: Extracting base paths.

duce conditions that depend on switch state.

- We design a routing policy language (RPL) for Aura (§4), which uses base paths, switch conditions, route propagation conditions, and route preferences to express a set of policies.
- Finally, a compiler generates BGP configurations from RPL and tests configurations in the network (§5).

3.1 Base Paths: Minimizing Reconfiguration

Aura minimizes reconfiguration of large production networks by pre-compiling the network with a set of paths known as the *base paths*. The base path concept stems from the following two insights.

- *Policies can be defined on abstract paths:* Modern data center topologies are usually hierarchical, symmetric, and uniform. To be resilient to failures and ease the operation, most data centers classify switches into different roles according to their layers in the FAT-tree like topology [21]. For example, every switch in the lowest layer (RSW in our topology) has a south bound connection to servers in a rack, and a north bound interface up to the aggregate switches (FSW in our topology). Switches of the same role are functionally equivalent. Thus, base paths can be defined as abstract paths using these abstract switch roles. In turn, policy intents can be expressed by using the base path or a sub-path of the base path.
- *Staging policies for dynamic scenarios:* Our base path set contains not only the preferred paths under a regular, static scenario, but also alternative paths under many dynamic scenarios, such as failures, migration, or maintenance as discussed in §2.2. When these changes occur, we simply need to select configurations that correspond to a different subset of base paths. Aura's synthesis has already deployed all configurations in the individual switches, and we just need to deactivate one configuration and activate the other.

We identify base paths by first *simplifying datacenter topology* and then *performing comprehensive reachability analysis* on that topology.

Topology simplification. We simplify a datacenter topology by abstracting multiple switches that share some given characteristic as a single switch. We can make abstractions at different granularities. For example, we can abstract all RSWs into a single node, or abstract all RSWs *in the same pod* into a single node. It is important to find the right granularity to maintain reachability without jeopardizing scale. Abstractions at too fine granularity create many paths and jeopardize synthesis scalability and operational maintainability. Abstractions at too coarse granularity (e.g., single role – single switch [6]) do not allow us to stage paths at different scenarios, such as intents I_1 and I_2, which requires traversing through specific switches in the network. Moreover, role-based abstraction can express these policies, but configuring switches to support them is a challenge. For example, intent I_2 requires visiting the same switch role multiple times. Aura's compiler needs a way to keep track of propagating the announcement through such paths (see §5).

Reachability analysis. We find a balanced abstraction by performing a primary reachability analysis between RSW switches on the minimal topology. We then extend these paths to support alternative paths, to accommodate cases when nodes on a primary path fail. Any duplicate paths will be collapsed into a single path. Figure 4 illustrates this process. We start with three primary paths: one between two RSWs in the same pod, the second between two RSWs in different pods, and the third between two RSWs in different datacenters. Then we consider the failure of the direct FSW, direct SSW, or direct FA switch (or links connecting them) on the primary paths. For each failure we add nodes to express the backup path. Note that we first consider single switch or link failures on each primary path. Further, we consider larger failure scenarios such as regional failures and disasters. For example, we consider backbone failure and craft the paths to provide intra-region paths by traversing the FA layer. Currently, we generate base paths according to our reachability objective, which is, providing reachability of intents in the presence of at most two failures (either link or device or both)[2]. Note that the process of generating base paths is not unique to Meta's network. Any datacenter network can exploit its symmetry and hierarchical nature to derive its base paths.

3.2 Staging and Labeling: Supporting Dynamic Configurations

We support network changes, while minimizing reconfiguration via staging and labeling. These mechanisms help us

[2]We determined any failures beyond that would degrade capacity, thus this would no longer be a reachability problem. We deploy other measures to handle such large-scale failures, which are beyond the scope of this work.

support parallel configurations in switches (with only one configuration being active) and to support conditional intents.

- *Staging configuration snippets and activating them with labels* are the two key techniques to allow coexistence of multiple policies. Each policy is implemented as a set of configurations and loaded onto the switches. A policy may have a condition expressed as a combination of labels. Only when the conditions are satisfied, the policy will be activated.
- *Propagating label with routes* is the way we achieve conditional policies. The labels are translated into BGP community tags in our synthesis process, and are attached to routes and propagated through switches. Each switch can match the conditions based on the current switch state, attach its own state as community tags, and announce it to downstream switches.

We provide two examples to show how staging and labeling support dynamic scenarios.

Supporting OpenR deployment. In our data centers, we are developing an alternative routing protocol known as OpenR [13] for better scalability. One challenge of applying Aura to production is how to gracefully migrate the switch configurations from an old to a new set of configurations. BGP and OpenR configurations are completely different and switching between them requires drastic changes to the fleet. We use Aura to support this process with no disruptions, as illustrated in Figure 5.

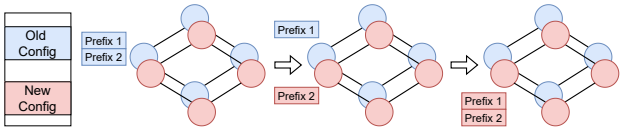


Figure 5: Staging and subscription used for migration.

Both versions of configurations are staged on all switches[3], each snippet starts with an activating condition, mapped into different BGP community tags. In Figure 5, we illustrate this as red and blue tags. Network operators can then attach tags to prefixes (prefix 1 or 2) to implement the policy for that particular prefix. By this, Aura simultaneously supports both the old and new policy. This technique also gives the network operator the flexibility to shift traffic in any order they like, and it also offers the fallback opportunity, should the new policy have some unforeseen effects on traffic when it is deployed.

Initially, all prefixes (prefix 1 and 2) use the old configuration (blue). We start with prefixes of less critical services first to avoid business disruptions. At the origin, $Prefix_2$ is announced with the red tag, so when the announcement arrives at intermediate switches, the corresponding red configuration is activated and the blue configuration becomes inactive. Other prefixes are gradually on-boarded to the new configuration by switching their tags in announcements. If the new configuration has any issue, we can safely switch back to the old version by controlling community tags at the origin.

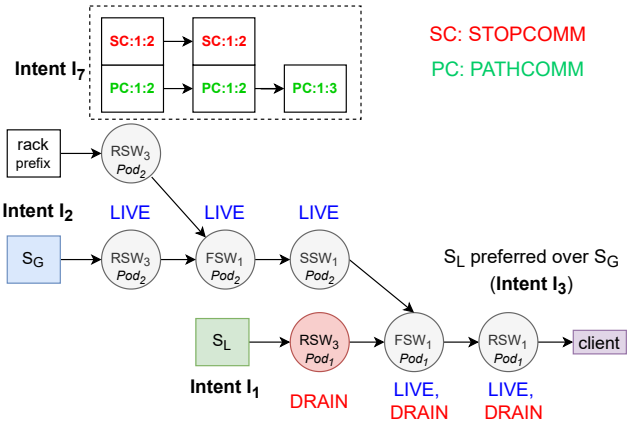


Figure 6: Supporting conditional policies.

Handling switch maintenance. As explained in §2.2, routes from LIVE switches are preferred over those from WARM switches (intent I_5), and routes from DRAINED switches should not be used (intent I_4). To reflect this policy, we stage parallel configurations on each switch[4]. Each configuration describes actions for a given state of this switch and a given state of the neighboring switch sending announcements to this switch. The actions could be adding/removing community tags, setting preferences, and accepting/rejecting routes. Taking intent I_1 and I_2 as an example, as shown in Figure 6, there are two paths to the client to reach the local service S_L and the global service S_G. Intent I_3 specifies that I_1 (primary) is preferred over I_2 (backup). The RSW switch along the primary path is going through a DRAINED state, therefore it attaches the DRAINED tag to its announcement. On the other hand, the backup path contains all live switches and the announcement only contains the LIVE tag. The downstream switches on receiving both the announcements would prefer the backup path to global prefix over the primary path to the local prefix. To support this, switches in both primary and backup paths should be configured to match the DRAINED tag and allocate a lower preference to the primary path. In §5, we show how to implement it with concrete BGP attributes.

4 RPL: Expressing Conditional Policies

Our routing policy language – RPL – allows a network operator to express high-level routing intents, by defining paths

[3]FBOSS supports concurrent deployment of BGP and OpenR configurations.

[4]FBOSS supports parallel configurations for each device state. We believe any other switch could also achieve this by pushing necessary configurations from the control plane.

Syntax			
name	::=	*string*	
rx	::=	*regular exp*	
lit	::=	*name* | ∼*name*	*names*
comp	::=	<| >| =	*comp*
bp	::=	*name {hops lit (->lit)*}*	*base path*
B_{bp}	::=	*base-paths {bp+}*	*base path block*
loc	::=	*name {regex-def rx (. rx)*}*	*location*
B_{loc}	::=	*locations {loc+}*	*location block*
tag	::=	*name rx*	*tag*
B_{tag}	::=	*tags {tag+}*	*tags block*
B_{top}	::=	*topology {* *name {* B_{loc} | B_{tag} | B_{bp} *}* *}*	*topology block*
B_{rout}	::=	*routing {topology lit}*	*routing block*
o	::=	*origins {location lit}*	*origin*
pc	::=	*propagate-condition {* *lit (,lit)** *}*	*propagation condition*
p	::=	*name {hops lit (->lit)*}*	*path*
B_{path}	::=	*paths {p+}*	*path block*
B_{prop}	::=	*propogation {pc B_{path}}*	*propagation block*
pref	::=	*preference {lit (comp lit)+}*	*preference*
pol	::=	*policies {* *name {* B_{rout} | *o* | B_{prop} | *pref* *}* *}*	*policy*

Table 1: RPL block and leaf statements.

and path preferences. We had two choices while designing RPL – to make it a domain specific language (DSL), that is, its syntax is designed from scratch or to make it an embedded language based on Python. Embedded languages typically benefit from existing IDE, debugger, type ahead assistant and error messages, which allows for quick adoption by network operators. However, embedded languages are hard to verify. For instance, in Python, users can override basic operators making it hard to verify a program without running it. On the other hand, with DSL, the language itself can be defined in a way that it can reject wrong programs and enforce invariants we care about. We could also apply any number of static analysis techniques to determine the effect without running the program. Therefore, we traded-off the flexibility provided by embedded languages for correctness and verifiablity of DSL.

RPL's syntax is designed from scratch and is based on ANTLR [1] – an engine that provides basic syntax parsing. Table 1 shows the collection of statements supported by RPL. Statements are used to identify base paths, location of switches, describe tags and define policies with their preferences and conditions. Groups of these statements, known as "blocks," are used to describe different components of the policy. For example, the topology block (B_{top}), describes the topology containing the locations of switches, tags and base paths. The RPL program required to support all policies discussed in Figure 1 is shown in Figure 7. To handle the challenges of expressing flexible policies (§2.3), RPL supports the following features.

Minimizing reconfigurations: RPL allows network operators to specify topology block (shown in lines 1–23) that helps in pre-compiling the network. Specifically, in the topology block, the operator can specify scopes of devices (§3.2), tags (§3.2) and base paths (§3.1) that could be used by policies. For example, to support intents I_1 to I_7, topology block `f16` is sufficient. As described in §3.1, network operators are free to specify any number of intents that could be supported by this topology block. If operators want to support other intents, they could always extend the topology block, but would require reconfiguration. From our operational experience, we show in §6 that these changes configure much lesser number of switches than competing approaches.

Supporting dynamic configurations: Network operators can specify different policy in the `policies` block of an RPL configuration file. This allows them to dynamically change policy over time. Each policy block, contains the `name` of the policy (e.g., RSW_REACHABILITY), `routing` (topology block used by the policy), `origin` (origination location of routes), `propagation` (set of paths used by the policy) and `preference` (preference among paths or tags). In Figure 7, policy RSW_REACHABILITY implements the intents I_1 to I_5, policy ALLOW_INFRA implements intent I_6 and policy LIMIT_RACK_PREFIX implements intent I_7. A network operator may choose to apply any one of these policies for appropriate prefixes. For instance, as described in an example in §3.2, network operators may apply RSW_REACHABILITY for services S_L and S_G and apply ALLOW_INFRA for infrastructure prefixes.

Expressing conditions with scopes: To support flexible granularity of intents, RPL introduces the notion of a *locations*, i.e., one can define a switch in RPL at different levels of granularity, such as switch role, switch ID, switch plane, fabric, region, and the datacenter. A location can be defined in the topology block (as shown in lines 3–12 in Figure 7). Location definition consists of the switch role, followed by switch number, pod number, fabric number, and datacenter name. Some elements can also be replaced by wildcards. The naming convention for our production network leverages the symmetry of the network to keep it simple and uniform. For example, the first spine plane in every data center would have the same identification number [4]. Network operators can leverage these naming conventions to define a scope. For example, I_1 requires the route to propagate through the first RSW present in Pod_1, present in the first fabric in the Altoona (ATN) datacenter, scope RSW.3.2.1.1.ATN can be used.

Expressing different switch states: RPL introduces the

`tags` block to indicate tags that will be used to identify switches in a given state. An example of the tags block is shown from lines 13–17 in Figure 7, which defines tags LIVE, DRAINED and WARM. These tags will eventually be mapped in the synthesis process into BGP community tags, which would be attached from every switch to communicate the state of the switch to its neighbors (see §5 for implementation details). RPL also allows network operators to limit the propagation of a route announcement by using `prop-condition` statement. In Figure 7, we limit the propagation of announcements based on the switch state to only LIVE and WARM switches (line 29).

Handling exceptions: As per I_6, DRAINED state switches are required to carry traffic for infrastructure prefixes. Network operators can define a new policy ALLOW_INFRA, that allows DRAINED switches to propagate routes. Network operators can use the ALLOW_INFRA policy for applicable prefixes (as described in §3.2).

5 Compiler Implementation

In this section, we discuss how Aura takes RPL specification and synthesizes BGP configurations for various switches. Aura uses properties of BGP to flexibly map the RPL specification to switch configurations.

5.1 Supporting Base Paths

The production network would need to be pre-compiled to support the base paths (specified in the topology block in RPL). To achieve this, Aura utilizes reserved community tags and generates rules to match these community tags. The key idea of using community tags is similar to that of source routing [22]. To implement a policy for a prefix, appropriate community tags are attached to the prefix while announcing it. On receiving the announcement, switches match on the community tags and take appropriate action of allowing or denying the route or modifying the tags. Although the idea seems straightforward, the challenge is how to come up with *systematic community assignment that can be interpretable and maintainable at scale*. Our key idea is an easy to interpret and debug encoding scheme that translates the base paths and other intents directly into different bits in the community attribute.

Encoding paths: Aura encodes the base paths into a structure called *path community tags* (PathComm for short). Depending on the number of base paths, Aura allocates a unique PathComm with a format of *PATHCOMM:P(1-10):H(1-6)*, where numbers in parentheses denote the bit sizes of the fields. The 10 most significant bits denote the unique base path ID P and the least significant 6 bits denote the hop number H. Some paths have switch roles that occur multiple times, e.g., intent I_2, where switch roles RSW and FSW occur twice. For such paths, the hop count field is used to keep track of where the

```
1  topology{
2      f16{
3          locations{
4              R1P1 { regex-def: RSW.1.1.1.1.ATN }
5              R3P1 { regex-def: RSW.3.1.1.1.ATN }
6              R3P2 { regex-def: RSW.3.2.1.1.ATN }
7              F1P1 { regex-def: FSW.1.1.1.1.ATN }
8              F1P2 { regex-def: FSW.1.2.1.1.ATN }
9              S1PL2 { regex-def: SSW.1.2.1.1.ATN }
10             FSW { regex-def:: FSW* }
11             RSW { regex-def:: RSW* }
12         }
13         tags{
14             LIVE L
15             WARM W
16             DRAIN D
17         }
18         base-paths{
19             B1 {hops RSW → FSW → RSW}
20             B2 {hops RSW → FSW → SSW → FSW → RSW}
21         }
22     }
23 }
24 policies{
25    RSW_REACHABILITY{
26         routing{topology f16}
27         origin{location RSW}
28         propagation{
29             prop-condition (L or W)← I4
30             paths{
31                 path P1 R1P1 → F2P1 → R3P1 ← I1
32                 path P2 R1P1 → F2P1 → S1PL2 → F2P2 → R3P2 ← I2
33             }
34         }
35         preference{
36             P1 > P2 ← I3
37             L > W ← I5
38         }
39     }
40     ALLOW_INFRA{
41         # Same routing, origin, paths as RSW_REACHABILITY
42         propagation{
43             prop-condition (L or W or D)← I6
44         }
45         preference{
46             L > W > D← I6
47         }
48     }
49     LIMIT_RACK_PREFIX{ ... } ← I7
50 }
```

Figure 7: RPL configuration describing policies.

announcement is on the path. Aura configures the switches to support intent I_2 as follows. At RSW on the first hop, the switch matches announcements that contain the PATHCOMM:1:1, modifies the community tag to PATHCOMM:1:2, (i.e., increments the hop count), and allows the announcement to be advertised to FSW. Similarly, Aura configures FSW to match, modify and advertise the announcement to the corresponding next-hop, that is RSW. When the announcement reaches RSW again at the last hop, the switch matches PATHCOMM:1:3, it stops announcing the announcement further. Therefore at RSW, there are two sets of rules, one that matches the announcement on the first hop (PATHCOMM:1:1) and the other that matches the announcement on the third hop (PATHCOMM:1:3). The use of the community tag ensures the abstraction of the switch role is maintaine,d without needing to split roles of switches further to support multiple hops

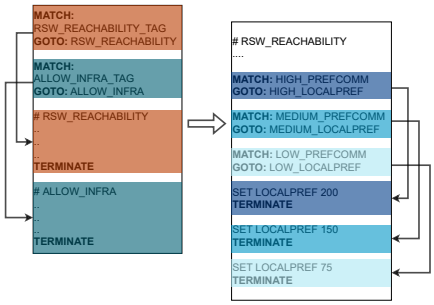


Figure 8: Aura generated configuration.

over the same switch type on the same path.

Containing announcements: Some intents require announcements to be contained to a specific location. For instance, intent I_7 limits rack prefix to pods (Figure 6). To support this, Aura uses a dedicated community tag known as *stop community* (StopComm for short). Similar to PathComm, StopComm follows the format of *STOPCOMM:P(1-10):H(1-6)*, where P and H are used to denote the path ID and the hop number to stop announcement respectively. For intent I_7, Aura appropriately configures RSW to stop propagating the announcement when it receives STOPCOMM:1:2 and restricts the announcement to pods.

During configuration synthesis, Aura generates appropriate match action rules to match on the corresponding communities and take appropriate action. That is, for PATHCOMM, the action would be to modify the PATHCOMM to reflect the hop changes and for STOPCOMM, it would remove the community tags and prevent the forwarding of the announcement.

5.2 Supporting Dynamic Network

Staging: Aura has reserved community tags to accommodate different policies. Aura's compiler incrementally allocates the community tags based on the order in which they are specified in RPL. For instance, from Figure 7, a dedicated community tag known as policy community (PolComm) will be allocated for RSW_REACHABILITY and ALLOW_INFRA policies. Similar to PathComm and StopComm, PolComm follows the format of POLCOMM:POL(1-16), where POL denotes the policy ID. Figure 8, illustrates how the staged community tags are used as a pointer to different segments of the BGP configuration file. BGP configurations are processed sequentially by switches. At the beginning of the configuration are match statements, matching the staged community tags. The corresponding action on a match is a GOTO statement pointing to the section of the configuration implementing the policy. At the end of the section is the TERMINATE command that terminates the processing of the configuration. We discuss consistency guarantee from a practical perspective in §7.

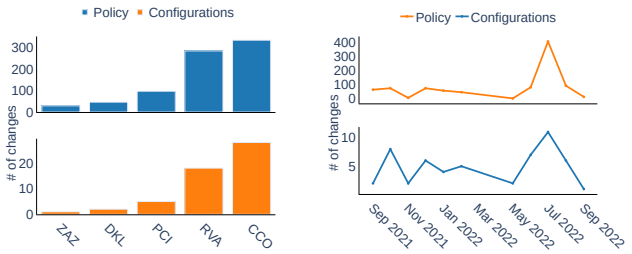
Supporting preferences among paths: Within each policy, network operators may specify preferences among intents (see §4). To accommodate this, Aura uses preference community (PrefComm) with a format of *PREFCOMM:X*, where X denotes the preference value. For every X value there is a one-to-one mapping to a local preference value. Currently, at every switch, there are twelve preferences matching twelve local preference values. If a switch receives an announcement with PrefComm, the configuration generated by the compiler contains a rule that matches on X and the action is a GOTO statement that implements the appropriate localpref. This is also illustrated in Figure 8, where there are match statements for setting high, medium, and low local preference values.

5.3 Validating Configuration

To verify the correctness of the BGP compiler output, Aura uses an emulation-based, routing policy validation framework. We run a container-based high-fidelity emulation of FBOSS switch [9], which constructs an overlay network among containers emulating a small-scale data center without relying on hardware switches. The topology contains multiple switches of the same role to mimic cross-POD and cross-DC route propagation. We load the configurations generated by Aura to the BGP agent on the emulation switches. The switches exchange routes according to the rules in the BGP policy and broadcast End-of-Rib (EoR) messages on convergence. On receiving all EoR messages, the verification framework collects Routing Information Base (RIB) from all switches for validation. The validation algorithm verifies whether the routing status is identical to the RPL policy. Here, the basic idea is to traverse the switch network graph as specified in the RPL propagation path, and check whether all routing paths are correctly present as specified in RPL (Algorithm in Appendix §A). First, the algorithm checks whether the prefixes are originated as intended. It mines the originated prefixes from the first hop switches of the RPL propagation path and inserts the found switch and prefix information into a queue for Breadth-First Search (BFS) traversal. Then, it pops data from the queue, looks up the given switch and prefix and collects the RIBs from the switches. From the next switches' RIBs, the algorithm searches for the matching prefix and community tag and checks if the next hop of the path is the current switch. If so, it marks the routing path as *visited* and pushes the next path information into the queue to continue traversal. This procedure is repeated until the algorithm traverses all propagation paths in RPL. We also perform additional verification on the RPL (described in Appendix C).

6 Evaluation

We first measure the configuration changes made in our production data center, and use these measurements to evaluate Aura by showing how it minimizes switch re-configurations (§5.1), has a flexible language to express policies (§4) and



(a) Region changes. (b) Changes over time.

Figure 9: Intents/policies generated by Aura.

overall reduces operator burden. Aura's compiler is implemented in Python, and has around 13.8 K lines of code. The routing policy verification is developed in Python with around 1,200 lines of code.

We capture the intent changes made by network operators for thirteen months from Sept, 2021 to Sept, 2022. During this time, Aura supported five different data center regions, where each region has a separate configuration. Figure 9(a) shows the number of configuration and policy versions generated by network operators across all Aura-supported regions. We introduced Aura to CCO and RVA datacenter regions in September 2021 and thus these two regions have the highest number of changes. We eventually rolled it out to the remaining regions this year, with ZAZ being the most recent data center running Aura. Over time, across these data center regions, there were 54 different versions of configurations and 840K different policies. On average, there were 10.8 versions of the configurations and 168 versions of policies per region. There are fewer configuration changes than policy changes as not all policy change would require reconfiguration. Many policies would have already been pre-compiled by Aura in the network (as seen in §3.1 and §3.2).

Figure 9(b) shows a timeline of the policy and configuration changes that occurred across all Aura-supported data center regions every month. Every month on average, we changed 5 configurations and 87.3 policies. The largest frequency of updates was observed in July 2022, where we made several updates to introduce a new propagation path from the backbone network to our data centers. We discuss our experience of this roll-out in Section 7.

6.1 Minimize Switch Changes

To show the benefits of pre-compiling the network, we compare Aura with Propane [6]. Aura creates snapshots of policies and configures the network to support multiple common policies in parallel, as opposed to Propane, which compiles the configuration as and when there is a policy that needs to be supported in the network. A key drawback of Propane is that a change in policy may lead to reconfiguration of a large number of switches. We use simulation to quantify the benefit of our base path design in practice. We simulate a scenario where we replay all the policy changes made by network operators as shown in Figure 9(b). That is, whenever a policy is changed, we determine the switches that require re-configuring in case when Aura is run versus the case when Propane is run, to synthesize switch configurations.

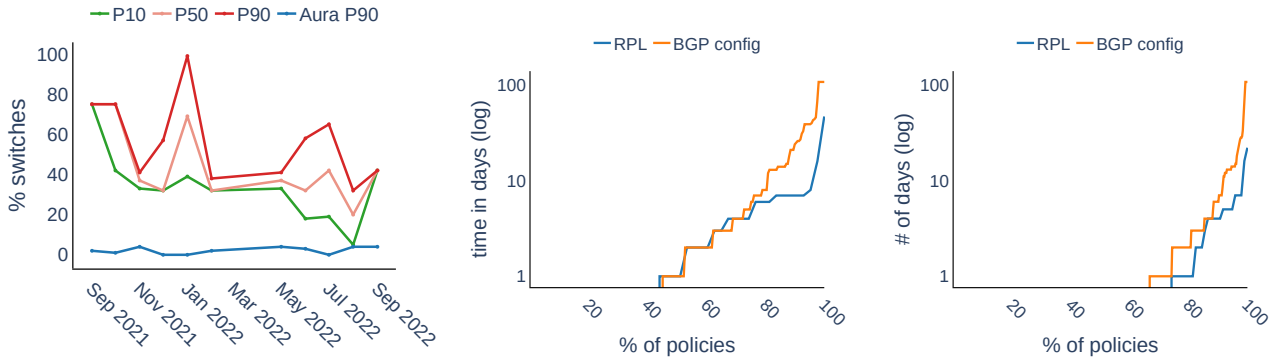
Propane involves too many switch re-configurations: Figure 10(a), shows the number of switches in all datacenter regions supported by Aura that are reconfigured by Propane. For every month, we show the 10^{th}, 50^{th} and 90^{th} percentile of switch re-configurations that are needed. As described in §2.4, this is time-consuming and disruptive to production traffic. Even the median number of policy changes on average can reconfigure 46.8% of all switches. Given the size of our networks, this could be in order of tens of thousands of switches. On the other hand, Aura reconfigures the network only when new policies cannot be supported by the pre-compiled configurations. Even for the 90^{th} percentile of cases, Aura only reconfigures 2.1% of the switches on average.

6.2 Aura Reduces Operational Burden

During policy generation, Meta adopts a peer review process, where network operators generate a configuration and ask other operators to review the changes. In case of non-Aura-supported regions, network operators send the new BGP configuration for review. On the other hand, for Aura-supported regions, network operators send the new RPL configuration for review.

Non-Aura regions take longer to evaluate: Figure 10(b) shows the time (log scale) taken in peer review for Aura and non-Aura regions. On average, we find that non-Aura regions take 8 days per policy and Aura-regions take 3.6 days per policy. Some policies can take much longer to review than others. For instance, the most amount of review time taken for a policy by non-Aura-region was about 108 days. This policy introduced a new fairness goal in the fabric aggregation layer, which was carried out in phases where multiple policies in non-Aura-regions had to be implemented before the given policy. On the other hand, for Aura-generated policies, the maximum review time was about 47 days. Similar to the non-Aura case, this specific policy involved introducing new backup paths and had to be extensively tested along with other Aura-generated policies.

Non-Aura regions generate many more code revisions: One key factor for reviewing configurations from non-Aura-supported regions are code revisions. Once the reviewer gives feedback via code review, the author addresses the comments and gets back to the reviewer. This process goes on until the author has addressed all the comments and the reviewer has no other feedback. Since the raw BGP configurations are much harder to understand than RPL, revision process tends to be error prone. Figure 10(c) shows the number of days



(a) Percentage of switches reconfigured.

(b) Time taken to review.

(c) Number of revisions.

Figure 10: Aura performance.

taken to generate a revision for Aura and non-Aura regions. On average, Aura-supported regions take 1.1 day per revision and non-Aura-supported regions have 3.2 days per revision. This difference is pronounced at the tail, where the maximum time taken for a revision was 22 days for Aura-supported region. However, for non-Aura-supported region, the maximum number of revisions was 107 days.

6.3 Aura Configuration Properties

Breakdown of RPL policies. All policy versions were encoded in 83.2 K lines of RPL by network operators. The most common statements used to define the policies include the propagation condition, propagation policy, origin and routing – this constitutes 1.9K–2.5K lines of code. There are also 382 preference statements. A large number of origin (2.5K), routing (2.5K) and path (2.3K) statements reflect the support for multiple backup paths in the network. Similarly, a large number of prop-condition statements, reflects the frequent need to restrict the scope of propagation in the network. Other statements including location, signature, base path, routing protocols and device states are used only once per configuration, and there were only 54 such statements.

Aura's Performance. Aura can be broken down into three stages: time taken to process and validate RPL, compiler execution time and the per-switch configuration generation time. RPL processing and validation takes 4.3 seconds and the compiler contributes 2.9 seconds, both of which are negligible in the entire process. The majority of the time is taken by BGP configuration generation, which takes about 88 seconds. This is expected, as the complexity of Aura mainly lies in the BGP compiler, which must synthesize BGP configurations for every switch in the network. Before Aura, network operators manually encoded intents into switch configurations. Based on our interviews with network operators, these configurations took about 26 weeks to generate, validate and safely deploy. Aura also scales when the number of policies increases, which typically occurs as the network grows. We artificially increase the number of policies to test the configuration synthesis times. We find that the synthesis times are about the same (about 90 seconds) for 50, 100 and 150 policies.

7 Operational Experience

RPL supporting verification for non-Aura regions. During Aura deployment, for the regions that are not Aura ready, i.e., those using legacy BGP configurations, operators still craft the intent in RPL. In this case, although RPL-based intent is not used to generate BGP policies, it is used for verification. The existing verification tool uses RPL as the source of truth and verifies it against manually generated BGP configuration. From this experience, RPL enables global verification before and during Aura deployment, which guarantees the correctness of the policy migration.

Identifying latent failures. The verification approach helped identifying several errors during the Aura deployment, such as intent specification errors, faulty compiler logic, BGP agent bugs, etc. We summarize the identified issues in Table 2 in Appendix B. We now look into two issues. First, when we configured multiple backup paths to reach RSW within the same pod – a primary path ($RSW \rightarrow FSW \rightarrow RSW$), a backup path via a different FSW, and a second longer backup path ($RSW \rightarrow FSW \rightarrow RSW \rightarrow FSW \rightarrow RSW$). When the primary path fails, the wrong configuration was used, resulting in using the longer backup path, instead of the shorter one, causing performance degradation. The second issue was a drained FSW switch, which did not announce the infrastructure prefixes to the connected RSWs. When another FSW switch failed, the external world lost connectivity to the RSW's management plane, which should have existed through the drained FSW switch. Such latent failures are hard to detect in production as they only manifest themselves during certain scenarios and the verification helped fix these errors before deployment.

Consistency guarantee in practice: One concern for policy

update is the consistency during convergence. Aura tackles the problem in a practical manner with three steps. First, we always drain a switch by moving the live traffic away, which is also called *disruptive config update* [9]. This prevents packet loss during transient state. Second, after the new path is enabled, we wait for a time period that is long enough for BGP convergence at a DC level. We conducted experiments to evaluate the convergence time scaling with the size of the network and chose the window to be tens of minutes. Finally, we use BMP monitoring to guarantee that the new routes are propagated in place before removing the drain configuration.

Ease of operation: After implementing Aura in some datacenter regions, we see significant improvement in experiences of network operators. For instance, after the last year's outage [3], we wanted to provide additional backup paths which included allowing routes over DRAINED switches with a lower preference. In Aura, we can support it with only 4 lines of RPL code change (details shown in Appendix D, Figure 11). This change took less than an hour to implement. On the other hand, we still had to support this change for non-Aura-supported regions, and it took three experienced engineers 30 days to make manual changes. During this time, the policy went through 6 rounds of review with over 40 comments for changes in the peer review process.

Supporting new networks: Our data center topology is constantly evolving to react to various deployment constraints and to new business requirements. For example, the recent global supply chain shortage pressed us to have a more condensed topology to reuse available ports. To accommodate the growing AI workload, we are developing a new AI backend topology. Adapting routing policy to a new topology is not trivial. Before Aura's deployment, it took network engineers up to 6 months to support a new topology. As an anecdotal example, in a recent deployment topology with Aura in June 2022 (ZAZ) took only 3 weeks by a single engineer.

Coordination with non-routing policies Aura focuses on specification and validation of routing policies. However, there are other policies that impact forwarding decisions. First, there are policies that are specific to services. Services have their own policies about the use of network resources (e.g., load balancing, replication) and can create varying traffic patterns that could lead to a routing policy adjustment. Future research is needed to streamline and coordinate intent changes between service and network layers. Second, there are the access control lists (ACLs) that restrict the flow of traffic across different network domains at Meta, which in turn affects routing policies. These ACLs also have a policy specification that is maintained by a different team. Detecting conflict and maintaining consistency across different intent management systems is an unsolved problem. Finally, while Aura manages all the data center routing policies, there are outside domains such as backbone and edge networks at Meta. These networks run different routing protocols and have their own policy intents. One of the extension of Aura is to support backbone routing intent using RPL, so that we can perform end-to-end verification. In the future, we plan to explore how to adapt Aura to other non-routing policies.

8 Related Works

Configuration synthesis: Propane [6], Propane/AT [7] and SyNet [10] use their own DSL or existing techniques to express intents. Then the intents are synthesized into low level configurations. Alternatively, rather than specifying intents, operators can partially specify the BGP configuration, and the synthesis approach can fill in the holes. Although these techniques help reduce operational burden, they cannot handle dynamic changes at scale without requiring constant reconfigurations. Propane [6] and Propane/AT [7] are also limited to BGP protocol, while Aura supports multiple routing protocols (BGP and OpenR). Finally, some configuration synthesis systems like SyNet [10] and ConfigAsure [18] are hard to scale to the size of large datacenter networks, like Meta.

Supporting dynamic configurations: Typically, systems focus on supporting dynamic configurations by switching from one configuration to the other. For instance, zUpdate [15] and Snowcap [20] determine a transition plan from one configuration to the other. However, these techniques involve shifting from one configuration to the other via several intermediate configurations. As seen in § 2.4, configuration updates to switches in a large network can take a lot of time, and transitioning across different configuration would only exacerbate this issue. There have been several other works in SDN [14, 16, 17], that help in transitioning from one configuration to the other. However, re-configuring in SDN context is different, than switch reconfiguration, as forwarding state is changed directly from a centralized controller, avoiding the challenges of a large distributed network.

Expressing intents: In recent years, there have been many efforts to raise the level of abstraction for low level configurations. Jinjing [23] introduces LAI to express ACL update synthesis and Propane [6] introduces RIR to express constraints on policy. Propane's RIR cannot express intents across different scopes, whereas both LAI and RIR do not support device state specifications and preferences based on these specifications.

9 Conclusion

Providing stable and efficient routing in large data centers is crucial. Existing synthesis systems generate configuration only once, but production networks require multiple reconfiguration to support their scale and dynamics. We present Aura, that enables network operators to express high-level intents to be automatically configured into the switch policy implementation with minimal reconfiguration.

References

[1] Antlr (another tool for language recognition). https://www.antlr.org/.

[2] United airlines jets grounded by computer router glitch. https://www.bbc.com/news/technology-33449693, 2015.

[3] Update about the 4 october outage. https://www.facebook.com/business/news/update-about-the-october-4th-outage, 2021.

[4] Anubhavnidhi Abhashkumar, Kausik Subramanian, Alexey Andreyev, Hyojeong Kim, Nanda Kishore Salem, Jingyi Yang, Petr Lapukhov, Aditya Akella, and Hongyi Zeng. Running BGP in data centers at scale. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 65–81. USENIX Association, April 2021.

[5] Alexey Andreyev, Xu Wang, and Alex Eckert. Reinventing facebook's data center network. https://engineering.fb.com/2019/03/14/data-center-engineering/f16-minipack/.

[6] 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.

[7] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. *SIGPLAN Not.*, 52(6):437–451, June 2017.

[8] Richard Chirgwin. Google routing blunder sent japan's internet dark on friday. https://www.theregister.com/2017/08/27/google_routing_blunder_sent_japans_internet_dark/, 2017.

[9] 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, 2018.

[10] 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.

[11] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 579–594, Renton, WA, April 2018. USENIX Association.

[12] 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*, SIGCOMM '16, page 58–72, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Saif Hasan, Petr Lapukhov, Anuj Madan, and Omar Baldonado. Open/r: Open routing for modern networks. https://engineering.fb.com/2017/11/15/connectivity/open-r-open-routing-for-modern-networks/, 2017.

[14] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. *ACM SIGCOMM Computer Communication Review*, 44(4):539–550, 2014.

[15] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zupdate: Updating data center networks with zero loss. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pages 411–422, 2013.

[16] Ratul Mahajan and Roger Wattenhofer. On consistent updates in software defined networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, pages 1–7, 2013.

[17] Jedidiah McClurg, Hossein Hojjat, Pavol Černỳ, and Nate Foster. Efficient synthesis of network updates. *Acm Sigplan Notices*, 50(6):196–207, 2015.

[18] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *Journal of Network and Systems Management*, 16(3):235–258, 2008.

[19] Jenni Ryall. Facebook, tinder, instagram suffer widespread issues. https://mashable.com/archive/facebook-tinder-instagram-issues, 2015.

[20] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: synthesizing network-wide configuration updates. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 33–49, 2021.

[21] 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 *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 183–197, New York, NY, USA, 2015. Association for Computing Machinery.

[22] Carl A Sunshine. Source routing in computer networks. *ACM SIGCOMM Computer Communication Review*, 7(1):29–33, 1977.

[23] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 214–226. 2019.

[24] Zhiping Yao, Hany Morsy, and Jasmeet Bagga. Opening designs for 6-pack and wedge 100. https://engineering.fb.com/data-center-engineering/opening-designs-for-6-pack-and-wedge-100/.

A Routing Policy Validation Algorithm

Algorithm 1 shows the algorithm used during emulation to verify Aura generated configuration.

Algorithm 1 RPL-based routing policy validation

```
procedure traverse(path, tag)
    ▷ path: a propagation path from RPL
    ▷ tag: a community tag that identifies policy

    ▷ Enqueue the originated prefixes into BFS queue
    Q = Queue()
    for sw in get_switches(path[0], all) do
        for R in RIB(sw) do
            if tag in R.tag then
                visited_sw = [sw]
                L = 0
                N = node(sw, route.prefix, L, visited_sw)
                Q.enque(N)

    ▷ Check if the prefix exists in RIBs
    while !Q.isEmpty() do
        N = Q.pop()
        L = N.L + 1
        for sw in get_switches(path[L], N.sw) do
            if sw in N.visited_sw then continue
            route_found = False
            for R in RIB(sw) do
                if R.prefix == N.prefix
                    and tag in R.tag
                    and R.nexthop == N.sw then
                    route_found = True
                    R.visited = True
                    if L < length(path) then
                        visited_sw.append(sw)
                        N_new = node(sw,R.prefix, L, visited_sw)
                        Q.enque(N_new)
            assert route_found == True

procedure routing_policy_validation(tag)
    prop_paths = get_propagation_paths_from_rpl(tag)
    for path in prop_paths do
        traverse(path, tag)
    ▷ Verify if there is any prefix leakage
    for sw in get_switches(all, all) do
        for R in RIB(sw) do
            if tag not in R.tag then continue
            assert R.visited == True
```

B Issues detected via emulation

We show the issues detected during emulation in Table 2. Broadly, they are categorized into compiler errors and BGP agent errors.

Category	Identified Issues
Compiler	Leaked intra-fabric prefix to FA
	Announced FA loopback address to unwanted scope
	VIP prefixes not withdrawn from drained devices
	VIP priority misconfigured
	Prefix originated from SSWs in one DC was not propagated to another DC
	Drained SSW did not forward default route to FSWs
	Prefixes from backbone not propagated to FADUs
	Drained FAUU does not withdraw default route
	FA loopback addresses leaked to RSW
BGP agent	Unsupported BGP action
	Config parsing crash in switches

Table 2: Identified issues by emulation

C Detecting Ambiguous Statements

Once RPL specification is complete, we use it as input into a compiler, which generates switch configurations. To avoid ambiguity, Aura compiler performs certain verification steps on the RPL specification.

Rule 1: Explicitly specify preference across all paths. It is possible that an operator designing a policy does not specify preferences across all the paths. For example, if there were four paths, P1–P4, and the preference rule stated $P_1 > P_2 > P_4$, preference for P3 is unspecified. One possible approach would be to assign a default preference value. However, depending on the preference values assigned to the other three paths, the total ordering of paths is unpredictable. For instance, if the default preference value is 100, there could exist two sets of ordering, $P_1(100) = P_3(100) > P_2(50) > P_4(25)$ or $P_1(200) > P_2(100) = P_3(100) > P_4(50)$. This non-determinism could make debugging routing behavior more challenging, and even worse, result in a hidden intent violation. Our solution is to detect underspecified preferences during verification and prompt the operator to explicitly define each preference.

A similar issue arises when the operator specifies two

Figure 11: Aura changes to support I_2.

parallel preferences: $P_1 > P_2, P_3 > P_4$. The order between paths in these different preferences can be interpreted as $P_1 > P_2 > P_3 > P_4$ or $P_3 > P_4 > P_1 > P_2$, and in several other ways. When verifying a RPL specification, Aura notifies operators to define ordering between all paths.

Rule 2: Detect hidden conditions. RPL supports adding a condition to a preference (Section 4). For example, the preference $P_1(BB\_DEFAULT\_ROUTE) > P_2 > P_3 > P_4$ means that for an announcement containing a community $BB\_DEFAULT\_ROUTE$, P_1 should have higher preference than other paths. However, there is an ambiguity. If the $(BB\_DEFAULT\_ROUTE)$ community is not attached, then the preference for P_1 is not specified, and the order of P_1 $(\neg BB\_DEFAULT\_ROUTE)$ and the rest of paths is unspecified as well. During verification, the operator is prompted to clear this ambiguity by explicitly specifying the preference for P_1 with and without the attached condition, and ensuring that the partial ordering is maintained.

D RPL changes to change intent

Figure 11 shows the RPL change done by network operators to allow DRAINED switches to propagate routes with a lower preference.