Topaz: Declarative and Verifiable Authoritative DNS at CDN-Scale

James Larisch*
Harvard University

Tim Alberdingk Thijm*
Princeton University

Suleman Ahmad
Cloudflare, Inc.

Peter Wu
Cloudflare, Inc.

Tom Arnfeld
Cloudflare, Inc.

Marwan Fayed
Cloudflare, Inc.

ABSTRACT

Today, when a CDN nameserver receives a DNS query for a customer's domain, it decides which CDN IP to return based on service-level objectives such as managing load or maintaining performance, but also internal needs like split testing. Many of these decisions are made a priori by *assignment systems* that imperatively generate maps from DNS query to IP address(es). Unfortunately, imperative assignments obfuscate nameserver behavior, especially when different objectives conflict.

In this paper we present Topaz, a new authoritative nameserver architecture for anycast CDNs which encodes DNS objectives as declarative, modular programs called *policies*. Nameservers execute policies directly in response to live queries. To understand or change DNS behavior, operators simply read or modify the list of policy programs. In addition, because policies are written in a formally-verified domain-specific language (`topaz-lang`), Topaz can detect policy conflicts before deployment. Topaz handles ~1M DNS queries per second at a global CDN, dynamically deciding addresses for millions of names on six continents. We evaluate Topaz and show that the latency overheads it introduces are acceptable.

CCS CONCEPTS

• **Networks → Naming and addressing**; **Programmable networks**; • **Software and its engineering → Formal software verification**.

KEYWORDS

authoritative DNS, CDN, formal verification, declarative, network policies

1 INTRODUCTION

Authoritative DNS nameservers originally executed a single function: Given a query for a server's domain name, return the server's

---

*Authors worked on this project during PhD research internships at Cloudflare, Inc.

IP addresses. Early on, server IP addresses rarely changed, so each nameserver maintained a static map from domain name to IP. To make changes, operators modified the nameserver's static name-to-IP assignments.

Domain names and nameservers have since evolved away from their initial designs. Instead of representing physical servers, domain names now represent application or service endpoints. Instead of using nameservers that statically map domains to server IP, many applications today employ CDNs [23], whose nameservers return *CDN* IP addresses when queried for application domains, allowing the CDN to intercede on application traffic. When assigning an IP address, CDN nameservers often consider the context of the query, not just the queried name. For instance, CDN nameservers may assign IP adresses to balance load [8], direct end users to proximate datacenters [11, 35, 42], avoid network saturation [5, 19], route around network path issues [9, 44], or minimize energy consumption [31]. However, CDNs may also enforce internal business objectives to, for example, isolate customers in a billing tier onto a single IP prefix for network provisioning, or direct some traffic to a new service endpoint for testing.

Many CDNs, both unicast and anycast, manage their DNS objectives using centralized *assignment systems*, among other components. An assignment system is a service that periodically matches query context or metadata (e.g., name, resolver location, or time of day) with CDN IP addresses. For instance, Akamai's assignment system periodically pairs DNS resolvers with the unicast IP of the closest CDN datacenter [11, 33, 42]. Assignment systems push the resulting pairwise mappings to CDN edge nameservers, which use the mappings (and sometimes local information such as per-server load [17]) to select and return the most appropriate IP addresses when they receive a DNS query. Anycast CDNs also use assignment systems to enforce their business objectives.

Unfortunately, objectives fed into assignment systems often conflict. Two (or more) objectives conflict when they each stipulate that the same subset of traffic be assigned different IP addresses. For instance, consider two operational objectives of equal priority specified by different teams: All queries received in France datacenters should receive IP address x, and all queries received in European Union (EU) datacenters should receive IP address y. What should happen to queries received in Paris, since it is in both France and the EU? By mapping Paris traffic to either x or y, the assignment system favors one objective over another, resulting in a bug.

Our experience at a large anycast CDN has shown us that assignment systems obfuscate, rather than elucidate, conflicting objectives. Once they generate and push the query-IP mappings to edge nameservers, the reasons for generating those mappings (and ignoring certain objectives) are lost or difficult to track down. As a result, nameservers cannot be debugged directly, since they just enforce

behavior decided imperatively long before. Furthermore, to the best of our knowledge, no CDN assignment system automatically detects objective conflicts at assignment time.

In this paper, we design and deploy Topaz, a new CDN architecture for managing authoritative DNS objectives declaratively. Using Topaz, a CDN's DNS objectives previously fed into imperative query-IP assignment systems are instead expressed by operators as named, modular programs called *policies*. Policies are executed directly by edge nameservers in response to live DNS queries.

Topaz's declarative design has three key features. First, compared with imperative mapping systems, executing declarative objectives directly makes DNS behavior easier to understand and modify. In Topaz, all DNS behavior is specified in a single file that can be easily understood and changed by operators (often non-experts), and changes to this file are deployed globally in minutes. Second, because nameservers execute this file directly, operators can debug nameserver behavior by reading or executing this policy file locally, using logged or sample queries as input. Finally, Topaz forces operators to explicitly and unambiguously specify the priority of new policies by positioning them appropriately in the ordered policy file—making it the single source of truth. Strict ordering encourages policy authors to consider how their policy interacts (or conflicts) with existing policies. For example, in the aforementioned France or EU case, the 'winning' policy is whichever is ordered first.

In addition, by encoding DNS behavior as a list of programs, Topaz can formally verify that the programs (and thus DNS behavior) are correct, and that policies do not conflict. To accomplish this, operators express Topaz policies in a domain-specific language called topaz-lang, which is accompanied by a formal model checker. The model checker, called Topaz verifier, can detect when one policy would nullify another (e.g., if the EU policy is ordered before the France policy). This property is called *reachability*—that there exists a DNS query to trigger every policy, given all policies ordered before it. Beyond reachability, Topaz also verifies *satisfiability* and *exclusivity*.

We believe that much can be learned from Topaz's design. However, we note that Topaz was designed for a global *anycast* CDN. In contrast to unicast or regional anycast systems, the target CDN does not use DNS for proximal routing or load balancing. Instead, it relies exclusively on BGP anycast for proximal routing, and balances load by moving application traffic between datacenters at layers 3 and 4 [15, 49] *after* it reaches an edge server. As a result, the CDN uses DNS to: (i) ensure traffic for customers using dedicated or static IP addresses get their reserved IPs; (ii) distribute remaining traffic according to customer service-level agreement (SLA) requirements; and (iii) satisfy an ever-changing and ever-growing list of business exceptions and edge-cases, e.g., split-testing new services. Topaz was built to service the latter two use cases. For more context, all CDN edge servers execute a single software stack, which means the default deployment model is that every program is executed on every server.

At time of writing, Topaz is deployed and operators are using it to incrementally replace the relevant parts of the CDN's legacy assignment system. Topaz currently receives ~1M DNS queries per second (qps). There are seven active, and at least 10 planned, deployed policies. topaz-lang and the formal verifier have also

been fully deployed, and are currently used by non-experts to write and verify policies.

To summarize, this paper makes the following contributions:
(1) We present Topaz, a new architecture for declaratively deploying authoritative DNS objectives at CDN-scale. We show how Topaz both improves transparency and enables new kinds of objectives that were not possible or very difficult to deploy with an assignment system.
(2) We apply the match/action pattern found in many network-policy languages [3, 10, 13, 24, 43] to specifying authoritative DNS behavior at a large-scale CDN.
(3) We present topaz-lang, a domain-specific language (DSL) designed for expressing and formally verifying DNS objectives. We describe how topaz-lang makes changing DNS objectives fast and safe.

This work does not raise any ethical concerns.

Outline. We proceed with additional background (§2) before describing the Topaz architecture (§3). We then present representative examples of policies deployed at a CDN (§4), which motivate our DSL, topaz-lang (§5), and its formal verification (§6). We then evaluate Topaz (§7) and discuss deployment experience (§8) before related work (§9) and concluding remarks (§10).

2 MOTIVATION & BACKGROUND

Our work is motivated by operator experience at a reverse-proxy anycast CDN that manages millions of customer websites. The CDN operates a global edge network with datacenters in 310+ cities across 120+ countries. In this section, we describe DNS at CDNs before discussing the problems with existing CDN nameserver architectures.

2.1 DNS at CDNs

Many web applications today rely on third-party services for their authoritative DNS. Such services run authoritative nameservers that receive A or AAAA queries for customer domains, and return the customer's preset IP(s) in response.

Reverse proxy CDNs also provide authoritative DNS, but their nameservers respond with CDN (rather than application) IP addresses to provide caching, security, and other services on behalf of the customer (see Figure 1). Customers delegate their domains to the CDN's nameservers like any other authoritative DNS service.

Each CDN, then, must determine how to best map incoming DNS queries for customer domains across their global IP space. Often, the CDN chooses these IP addresses in order to, for instance, maximize performance, minimize utilization, or both. For instance, unicast CDN nameservers may return the IPs of the CDN datacenter topologically closest to the resolver that made the query [11, 35, 42], regardless of the domain requested. In contrast, anycast CDNs (which get proximal request-routing "for free" [9]) may assign queries to distinct "rings" of anycast IPs to balance load [19]. Indeed, to illustrate to what extent CDNs can decouple names from IPs, it has been shown that 20M domains can be served from a single IP, or an IP selected at random [18].

Our target CDN uses DNS primarily to satisfy internal and external service-level objectives, rather than for proximal routing or load balancing. The CDN uses anycast globally, which means it
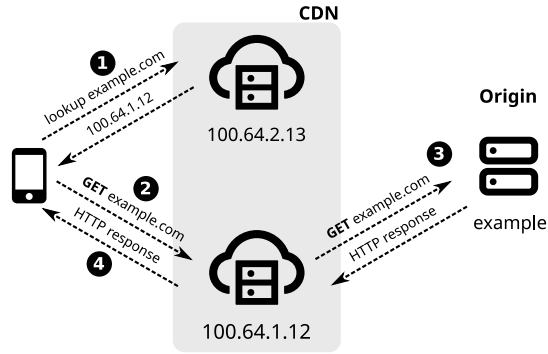
Figure 1: DNS at a reverse-proxy CDN. ❶ CDN authoritative nameservers respond to resolver queries for customer domains (e.g., example.com) with *the CDN's own addresses.* ❷ Client requests for the domain are routed to a CDN edge node for caching and security services; the CDN's authoritative nameserver and HTTP server can coexist and be co-located. ❸ Requests may be forwarded to the customer's origin server before the ❹ response is sent to the client.

relies on BGP to route client requests to the proximal datacenter. Furthermore, it balances application traffic between datacenters using a layer 3/4 load-balancer [15, 49] . But the CDN has many different teams, each with different DNS-related needs. As a result, it uses DNS to ensure customer traffic is given addresses (and thus priority) corresponding to to their service level agreements (SLAs).

2.2 A Priori Assignment System Limitations

Many CDNs achieve some of their objectives using an *assignment system* that ingests both CDN objectives and the set of CDN IPs, and then outputs mappings from DNS query metadata to IP address(es). Query metadata may include the queried name (e.g., `example.com`) and other information, such as a (resolver) location or internal metadata attached by the CDN (e.g., customer priority). The assignment system distributes the resulting mappings to edge authoritative nameservers. When an edge nameserver receives a query, it uses that query to look up the corresponding IPs in the static mappings, which it returns to the resolver. Before Topaz, our target CDN employed an assignment system that mapped customer domains to the IPs matching the customer's SLA. As the business objectives grew in number and diversity, so did the system's limitations.

Unfortunately, different operator objectives can conflict and lead to bugs. An objective conflict occurs when different objectives operate on overlapping classes of DNS queries. This often happens because different objectives are dictated by different teams (some CDNs have thousands of employees) that have not manually checked whether their policies conflict. For instance, consider two different objectives that each requires that all queries received in the a certain location receive certain (different) IP addresses. Assignment systems are forced to make a zero-sum choice: one of those objectives must win and the other must lose, since queries in that location can only be given one set of IPs. This choice is embedded into the resulting static assignments distributed to the edge.

Objective conflicts can be difficult to detect and debug for multiple reasons. First, the assignment system itself cannot determine if,

```
1  def match(query, config) {
2    return query.Name == config.Name
3  }
4  def response(query, config) {
5    return (config.IPv4s, config.IPv6s, config.TTL)
6  }
```

(a) DNS lookup expressed as policy match and response functions.

```
1  trivial-policy:
2    Name: "example.com"
3    IPv4s: ["100.64.0.1", "100.64.1.1"]
4    IPv6s: ["2001:0:0ea2::0001", "2001:0:0ea3::0001"]
5    TTL: 300
```

(b) Conventional DNS mapping expressed as a policy configuration.

Figure 2: A conventional DNS lookup expressed as a Topaz policy has two components: (a) the lookup expressed by match and response functions, and (b) the policy's parameterized configuration.

or to what degree, the objectives overlap, so operators receive no feedback upon entering their objectives. Second, the place at which authoritative DNS decisions are made (the assignment system) is far from where they are enforced (the edge nameservers). The gap makes it difficult both to verify that nameservers are behaving as expected, and to debug nameservers when a problem arises.

At the target CDN, before Topaz, DNS-driven business objectives frequently conflicted with one another, leading the assignment system to silently ignore certain objectives. As a result, operators were forced to manually find and arbitrate the objective conflicts caused by different stakeholders. This both burdened developers with resolving contradictory business objectives and caused frequent frustration when conflicts were not caught and certain objectives were ignored. As the sets of stakeholders and objectives at a CDN grows, so does the complexity of managing and understanding its authoritative DNS behavior. In describing the existing assignment system, one engineer declared: "I don't want this system to exist anymore."

3 INTRODUCING TOPAZ

We developed Topaz to make authoritative DNS behavior at CDNs more transparent and understandable. To achieve these goals, Topaz encodes each distinct stakeholder objective as a program called a Topaz *policy*. Topaz policies are executed directly in response to DNS queries by Topaz-edge, an edge service called by CDN nameservers. Policies are submitted to, and distributed to Topaz-edge by, a centralized service called Topaz-core. The two services together comprise ~8K lines of Go code. The overall architecture is depicted in Figure 3 and described below in detail.

3.1 Topaz Policies

We designed Topaz to execute DNS objectives encoded as programs called policies. A Topaz policy consists of a **match** and **response** function, which both operate over incoming DNS queries. The match function returns a Boolean indicating whether or not the policy should execute on the received query. The response function returns a 3-tuple consisting of an array of IPv4 addresses, an array of IPv6 addresses, and a time-to-live (TTL) value. Broadly, then, a Topaz policy is a program that encodes (i) a class or set of DNS queries identified by metadata; and (ii) how DNS response values

should be selected for that set of queries. This stateless match/response paradigm has not only proven comprehensible for software engineers, but also facilitates formal policy verification (§6).

We illustrate the simplest Topaz policy, which emulates traditional name-IP addressing, in Figure 2a. This policy *match*es all queries for the parameterized domain name and, if a match is found, *responds* with parameterized address(es) and TTL. During execution, Topaz-edge retrieves policy-specific parameters from that policy's corresponding (YAML) *configuration*, shown in Figure 2b. When operators need to change the domain name used, they modify the policy's configuration without having to change the policy's code.

3.2 Topaz-edge

Policies are executed directly on the edge by a service called Topaz-edge. When a CDN edge nameserver receives a DNS query, it attaches customer-specific metadata (e.g., an ID or SLA), then forwards the query to Topaz-edge.[1] Topaz-edge executes the ordered list of policies, with the received query, until a match is found. Upon finding a match, Topaz-edge calls that policy's response function and returns the resulting tuple to the nameserver. The nameserver wraps the resulting data in a valid DNS record and returns it to the resolver. If no match is found, Topaz-edge notifies the nameserver to fall back to a default set of static assignments.

Topaz-edge was consciously implemented as a separate service to isolate the deployment paths of our experimental service from the production nameserver. The choice balanced organizational and performance tradeoffs: Nameserver developers can focus on DNS protocol correctness and performance, while Topaz operators can focus on expressing and deploying business logic (objectives). However, this separation does introduce inter-process communication (IPC) overhead, which we evaluate in §7.

3.3 Topaz-core

Operators first submit policy changes to a separate, centralized service called Topaz-core. Topaz-core maintains a YAML file which includes all deployed policies and their respective configurations. To make changes to authoritative DNS behavior, operators edit (e.g., add, remove, reorder, change) policy configuration values in the list directly. Topaz-core detects changes to the configuration file, and then writes the content to a global low-latency key-value store called Quicksilver (QS) [37] maintained by the CDN. Topaz-edge periodically polls QS for policy configuration changes and, when detected, updates its list of policy configuration values accordingly. QS ensures that changes are transactional—Topaz-edge will either retrieve the complete old or complete new configuration when polling.

The order of policy blocks in the YAML file specifies the order that Topaz-edge executes policies. We considered other data structures but found that a single ordered list achieves simplicity without ambiguity—it encourages developers to consider how adding or changing a policy may change the match-rate of subsequent policies. To guarantee that adding new policies does not 'squash' existing ones, we implemented formal verification of policy configurations (§6).
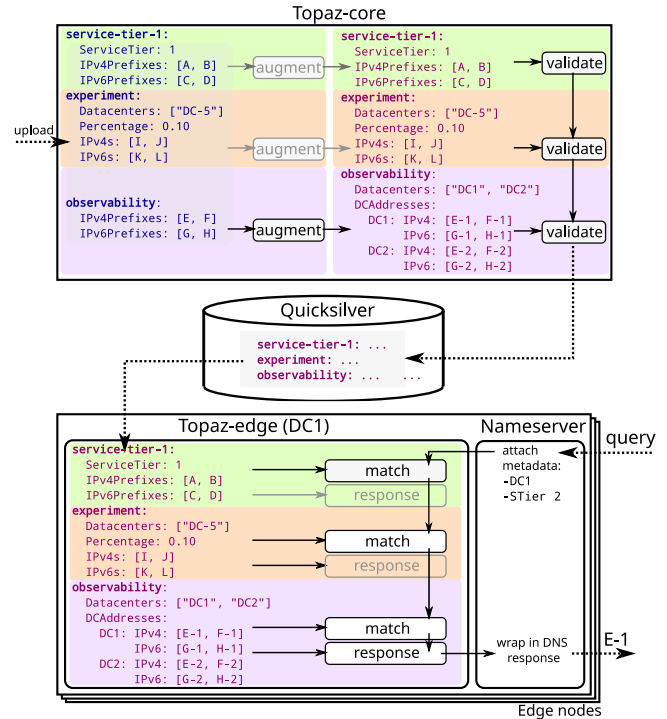
Figure 3: The Topaz architecture. Policy configurations are uploaded to Topaz-core. Each policy's configuration block is optionally augmented and validated before written to the edge-wide key-value store QS. Topaz-edge, on all edge servers, periodically polls QS for updates. Topaz-edge receives DNS queries from the authoritative nameserver, and executes every match function until one returns true, followed by that policy's response function. Values are returned to the nameserver to construct and return the query's response.

Topaz-core performs two additional steps after receiving configuration changes and before pushing those changes to the edge. First, it executes each policy's (optional) *augmentation* function. Augmentation functions supplement static policy configuration with dynamic data. For example, as shown in Figure 3, a policy's augmentation function may retrieve the set of datacenters that are active or in a region. Augmentation functions free authors from having to manually track the operational state of the global network in a policy's configuration. Second, after augmentation, Topaz-core executes each policy's (optional) *validation* function. Validation functions perform type- and syntax-checking; for example, ensuring that a percentage value in a configuration is between 0 and 1. Only after both succeed (and, as we describe later, changes have been formally verified) does Topaz-core writes the final configuration file to QS.

3.4 Code versus Configuration

As described, Topaz separates policy configuration from policy code. Each policy's match and response functions (Go code) are implemented in the Topaz-edge repository. Changes to these functions, i.e., changing the way an IP address is selected from a set,

---

[1]Edge servers in the CDN are homogeneous: each runs the same set of system services, including both Topaz-edge and the authoritative nameserver service.

```
1  def match(query, config) {
2      return config.ServiceTier == query.Customer.ServiceTier
3  }
4  def response(query, config) {
5    ipv4s = []
6    for prefix in config.IPv4Prefixes {
7      ipv4s.push(prefix.select(hash(query.Name)))
8    }
9    ipv6s = []
10   for prefix in config.IPv6Prefixes {
11     ipv6s.push(prefix.select(hash(query.Name)))
12   }
13   return (ipv4s, ipv6s, config.TTL)
14 }
```

(a) Service tier isolation policy.

```
1  def match(query, config) {
2    r = randGen(query.Time)
3    return config.Datacenters.includes(query.Datacenter)
4      && r.sample(0, 1) <= config.Percentage
5  }
6  def response(query, config) {
7    return (config.IPv4s, config.IPv6s, config.TTL)
8  }
```

(b) Feature experimentation policy.

```
1  def match(query, config) {
2    return config.Datacenters.includes(query.Datacenter)
3  }
4  def response(query, config) {
5    ipv4s = config.DCAddresses[query.Datacenter].IPv4s
6    ipv6s = config.DCAddresses[query.Datacenter].IPv6s
7    return (ipv4s, ipv6s, config.TTL)
8  }
```

(c) Observability policy.

Figure 4: Examples of Topaz policies in use at a global CDN.

```
1  service-tier-1:
2    ServiceTier: 1
3    IPv4Prefixes:
4      - "192.0.2.0/24"
5      - "198.51.100.0/24"
6    IPv6Prefixes:
7      - "2001:db8:a1:/48"
8      - "2001:db8:a2::/48"
9    TTL: 300
10 experiment:
11   Datacenters: ["DC-5"]
12   Percentage: 0.10
13   IPv4s: ["203.0.113.1", "203.0.113.2"]
14   IPv6s: ["2001:db8:ab:1::", "2001:db8:ab:2::"]
15   TTL: 300
16 observability:
17   Datacenters: [] # Calculated at augmentation
18   IPv4Prefixes: ["100.64.0.0/16", "100.65.0.0/16"]
19   IPv6Prefixes:
20     - "2001:db8:a3::/48"
21     - "2001:db8:a4::/48"
22   DCAddresses: {} # Calculated at augmentation
23   TTL: 300
```

Figure 5: Configuration for the policies shown in Figure 4

4.1 Policy: Service Differentiation

Large networks like CDNs must ensure that customers with certain service-level agreements (SLAs) receive the quality-of-service they pay for. To do this, CDNs may reserve different physical resources (e.g., servers or datacenters) for different SLAs, and then assign these resources distinct IP prefixes [19]. They can then map customer traffic to the appropriate resources via the IP addresses provisioned accordingly—for example, to prioritize higher SLAs over lower ones. This use case motivates our first policy.

Concretely, we consider the following objective: All traffic for a particular service tier should receive IP addresses from that tier's specified prefixes. We show this policy's match and response functions in Figure 4a. The match function checks whether the queried customer's service tier matches the policy's specified `ServiceTier` in the policy's configuration shown in Figure 5 (`service-tier-1`). If the service tiers match, the response function deterministically calculates IP addresses from the configured prefixes by appending a hash of the queried name to each prefix; the hash length is equal to the host portion length (e.g., 8 bits when using a /24 prefix).

Introducing new service tiers. Consider the introduction of a new service tier with its own IP prefixes. The new policy is identical in structure, so authors can duplicate the existing policy configuration and code. In the copy, authors change the policy name to indicate the new service tier, change the `ServiceTier` value, and change the IP prefixes.

Changing addresses. Operators may need to change prefixes to route traffic for this service-tier to different datacenters or to re-partition the CDN's address space. Using Topaz, these migrations are simple—operators simply change `IPv4Prefixes` and `IPv6Prefixes` in the policy configuration (Figure 5). Because the configuration is decoupled from code, changes to prefixes are fully deployed to edge servers within minutes.

4.2 Policy: Feature Experimentation

CDNs often want to safely and incrementally deploy experimental features. Our second policy, shown in Figure 4b, allows the CDN to gradually and reversibly siphon a percentage of eligible traffic

requires an entire redeployment of Topaz-edge—which can take hours or more. Furthermore, similar response functions shared by two policies must be implemented twice, which creates consistency and maintenance challenges. Policy configuration changes, on the other hand, can be deployed in minutes (via QS).

We pursued this initial design so developers could write policies in a familiar language (Go), while ensuring operators could rapidly modify (certain) policy behavior. In the next section, we describe Topaz policies using this initial system design. However, as Topaz evolved, we closed the gap between code and configuration. We pushed *all* policy behavior, including match and response functions, into the configuration file. To do this, we developed a custom domain-specific language, which we describe in §5.

4 TOPAZ POLICIES

In this section we examine Topaz policies in detail using three examples. We show these policies' match and response functions in Figure 4 and corresponding configurations in Figure 5. Each example features unique matching criteria and a unique address selection mechanism. Each example closely resembles at least one of the seven policies deployed at the global, anycast CDN. Operational sensitivities preclude exact reproductions of policies in use. Collectively, however, they accurately reflect policies that decide addresses for millions of domains on six continents.

to resources (via IPs) where experimental features are deployed. The match function probabilistically returns true in select datacenters, and the response function returns the IPs of resources where the experiment is being conducted. Both the probability and datacenters are expressed as parameters in the policy's configuration (experiment) in Figure 5, as are the IP addresses returned in by the response function.

Rapid response to bugs. Consider an outage or critical bug that forces operators to disable the experiment. Using Topaz, operators can disable the experiment by changing config.Percentage to 0, so the policy's match function never returns true. Operators can gradually increase that percentage when reactivating the experiment.

Per-query match criteria. This policy decides to return true on a per-query basis. Instead, the policy could decide whether to execute on a per-domain basis by using a hash of the queried domain name (modulo 101) instead of r.sample(0, 1). Intuitively, per-domain experimentation is easier to reason about and facilitates debugging.

4.3 Policy: Internet Observability

CDNs monitor Internet topology to better understand and react to changes in routing conditions. One situation of interest is when the path from an ISP's resolver to the CDN and the path from a subscriber of that ISP to the CDN terminate in different CDN locations (recall that both CDN datacenters anycast all of the same addresses) [18].

Our final policy, shown in Figure 4c, detects this situation. The match function returns true if the query was received in one of the configured 'observability' datacenters. The policy's response function returns a unique IP address per datacenter—critically, this address is still anycasted from all datacenter locations, but only returned by Topaz from these datacenters. Thus, if the CDN observes application-layer traffic at a non-observability datacenter on that IP, the routing anomaly has occurred.

Vantage point changes. This policy illustrates how Topaz can facilitate passive measurement. The configuration (observability) in Figure 5 marks all datacenters (via augmentation, §3.3) as 'observability' datacenters, but could instead manually list a select set of vantage points.

5 A TOPAZ-SPECIFIC DSL

Configuration changes in Topaz are powerful. Many policy changes can be achieved via configuration changes only. But other changes—those that modify match and response functions—cannot be deployed rapidly because these functions are statically bundled with Topaz-edge. Operators must commit such changes to the Topaz-edge repository and deploy them to the global edge using standard release channels, which can take from hours to a day.

To close the gap between code and configuration, we developed a new domain-specific language for expressing both policy configuration and policy code called topaz-lang, which we describe in the remainder of this section. All seven policies deployed at the CDN are currently deployed using topaz-lang.

```
 1   - name: service-tier-1
 2     exclusive: false
 3     config: |
 4       (config
 5         [ServiceTier 1]
 6         [IPv4Prefixes
 7          (list (ipv4_prefix "192.0.2.0/24")
 8                (ipv4_prefix "198.51.100.0/24"))]
 9         [IPv6Prefixes
10          (list (ipv6_prefix "2001:db8:a1:/48")
11                (ipv6_prefix "2001:db8:a2::/48"))]
12         [TTL (ttl 300)])
13     match: |
14       (match
15         (= (get "ServiceTier" query) ServiceTier))
16     response: |
17       (response
18         (let ([hashed (hash (get "Name" query))]
19               [ipv4s (select IPv4Prefixes hashed)]
20               [ipv6s (select IPv6Prefixes hashed)])
21           (list ipv4s ipv6s TTL)))
22   - name: experiment
23     exclusive: true
24     config: |
25       (config
26         [Datacenters (list (datacenter "DC-5"))]
27         [Percentage (percentage 10)]
28         [IPv4s (list (ipv4_address "203.0.113.1")
29                      (ipv4_address "203.0.113.2"))]
30         [IPv6s (list (ipv6_address "2001:db8:ab:1::")
31                      (ipv6_address "2001:db8:ab:2::"))]
32         [TTL (ttl 300)])
33     match: |
34       (match
35         (let ([r (rand_gen (hash (get "Name" query)))]
36               [sampled (sample r (range 0 1))])
37           (and (in? (get "Datacenter" query) Datacenters)
38                (<= sampled Percentage))))
39     response: |
40       (response (list IPv4s IPv6s TTL))
41   - name: observability
42     exclusive: true
43     config: |
44       (config
45         [Datacenters (get_datacenters "observability")]
46         [IPv4Prefixes
47          (list (ipv4_prefix "100.64.0.0/16")
48                (ipv4_prefix "100.65.0.0/16"))]
49         [IPv6Prefixes
50          (list (ipv6_prefix "2001:db8:a3::/48")
51                (ipv6_prefix "2001:db8:a4::/48"))]
52         [DCAddresses (distribute Datacenters
53                                  IPv4Prefixes
54                                  IPv6Prefixes)]
55         [TTL (ttl 300)])
56     match: |
57       (match (in? (get "Datacenter" query) Datacenters))
58     response: |
59       (response
60         (let ([datacenter (get "Datacenter" query)]
61               [ips (get datacenter Datacenters)]
62               [ipv4s (get "IPv4s" ips)]
63               [ipv6s (get "IPv6s" ips)])
64           (list ipv4s ipv6s TTL)))
```

Figure 6: Policy configuration with topaz-lang.

We developed a new DSL, rather than use existing policy languages, for two primary reasons. First, we wanted our language to prevent certain classes of bugs and mis-configuration. topaz-lang's non-primitive types were built specifically for DNS (e.g., ipv4_prefix, ttl, datacenter) and many types make it impossible for operators to express categorically incorrect values (e.g., invalid IP prefixes, datacenters that do not exist). Second, we wanted our language to be powerful enough to express complex policies while keeping it limited enough to enable formal verification.

5.1 Deploying `topaz-lang` Policies

`topaz-lang` subsumes all policy configuration and functions. Figure 6 shows both the configuration from Figure 5 and policy functions from Figure 4 expressed in one file. To add a new policy, operators add a new YAML block of the following form to the file:

```
- name: <name>
  exclusive: <true/false>
  config: (config <bindings)
  match: (match <e1>)
  response: (response <e2>)
```

where <name> is the name of the policy, <exclusive> is the flag for exclusivity property (explained in §6), <bindings> is a variable number of configuration bindings from name to expression, <e1> is an expression that evaluate to `true` or `false`, and <e2> is an expression evaluating to the response function's 3-tuple (a `topaz-lang` list). Match and response functions take an implicit argument query, which is a `topaz-lang` map of key-value pairs.

Operators edit only the file shown in Figure 6 to modify, remove, or add a policy. Topaz-core pushes this file via QS to Topaz-edge, which stores each policy's parsed match and response functions (and configuration). Upon receiving a query, Topaz-edge executes policy match and response functions as before, except now it uses its built-in `topaz-lang` interpreter to do so. Using `topaz-lang`, *all* policy changes take minutes to deploy to the CDN's global edge network. Our prototype `topaz-lang` interpreter comprises ~5K lines of Go code.

5.2 `topaz-lang` Syntax

In service of simplicity, we made `topaz-lang` a non-Turing-complete S-expression language with a small set of built-in types. It is dynamically typed, and each expression evaluates to exactly one value. The primitive types are Booleans, numbers, and strings. There are constructors for pairs, lists, maps (associative arrays), ranges, and sets. `topaz-lang`'s other types were designed specifically for writing Topaz policies and include IPv4 prefixes and addresses, IPv6 prefixes and addresses, TTLs, datacenter IDs, random number generators, and percentages.

We summarize `topaz-lang` syntax in Figure 7. Built-in functions (indicated by f) include Boolean negation, numerical comparisons, functions to construct composite data structures, and functions to read from these data structures (e.g., get). Both control flow and the set of available functions are intentionally limited. There are `if` statements but no loops or recursion. All functions are built-in; users cannot define their own functions. All functions evaluate to exactly one value, but can throw an error if given invalid input. For instance,

```
(get "B" (map (pair "A" true)))
```

throws an error because the key "B" is not present in the given map.[2]

5.3 Augmentation and Validation

`topaz-lang`'s type system and built-in functions also replace configuration augmentation functions. Consider the observability

$$e ::= l \mid x \mid (\text{and } e_1 \ldots) \mid (\text{or } e_1 \ldots) \mid$$
$$(f\ e_1\ e_2 \ldots) \mid (\text{if } e_1\ e_2\ e_3) \mid (\text{let } ([x_1\ e_1][x_2\ e_2]\ldots)\ e_n)$$
$$l ::= \text{true} \mid \text{false} \mid \text{number} \mid \text{string}$$
$$x ::= \text{query} \mid \text{configuration identifier}$$

Figure 7: `topaz-lang` syntax description.

policy's `topaz-lang` configuration—Datacenters is a value computed from calling the built-in `get_datacenters` function that pulls the list of available experimental datacenters from an internal API. Then, the built-in function `distribute` assigns addresses from `IPv4Prefixes` and `IPv6Prefixes` to `Datacenters`. Topaz-core evaluates the expressions in `config` before writing the "final" computed configuration to QS (Topaz-core also includes the `topaz-lang` interpreter).

While not currently deployed, in the future policies will be able to define extra validation functions in `topaz-lang` using an additional `validate: (validate <e3>)` key-value pair in its YAML block, where <e3> is an expression evaluating to `true` or `false`. This allows policy authors to define optional additional checks to perform for ensuring that policies are configured correctly.

6 `topaz-lang` POLICY VERIFICATION

Our latest version of Topaz formally verifies `topaz-lang` policies to automatically detect policy match function conflicts, and other properties. Topaz's formal verifier checks the correctness of individual policies and of inter-policy interactions against a formal model of `topaz-lang`. Policies are checked following augmentation using the final configurations.

Our verification system uses theories of first-order logic to define a formal model of `topaz-lang`'s semantic behavior. This model makes it possible to articulate and check properties as logical formulas. For instance, we can express the specification "there exists a query q that matches policy A" as $\exists q[m_A(q)]$. This formula can be checked by a Satisfiability Modulo Theories (SMT) solver: the solver will search for a query q that satisfies the match function m_A of policy A.[3]

6.1 Policy Verification by Example

Before describing Topaz's logical model, we illustrate the value of policy verification by showcasing three properties we currently verify: satisfiability, reachability, and exclusivity. For exposition, we consider the policies `service-tier-one` (ST1), `experiment` (EXP), and `observability` (OBS) from Figure 6. We assume that OBS augments the `Datacenters` field to the list `["DC-1", "DC-3", "DC-5"]`.

Property 1: Satisfiability. A policy is *satisfiable* if its match function has *some* query for which it returns true. Unsatisfiable policies can never execute and are essentially "dead code". Policies may be unsatisfiable because of a bug in the policy code, or become unsatisfiable due to a regression, e.g., the match function returns true only for a retired datacenter.

To check satisfiability, the verifier searches for a DNS query q that satisfies the encoded match function formula. DNS queries are

---

[2]Errors in match or response functions halt policy execution, moving to the next policy in order.

[3]This model of the match function elides the config parameters, which we treat as inlined data.

modeled as a finite map from fields to symbolic variables. For the sake of this example, we consider queries with the following three fields: Name, ServiceTier and Datacenter (our complete model supports other fields). Our verifier is aware of all possible values these variables can take (e.g., it knows all possible datacenters currently in use by the CDN). To encode the match function, the verifier uses an equivalent symbolic version of the topaz-lang interpreter to construct a Boolean formula over q.

For example, to check if the observability policy is satisfiable, the solver checks $\exists q[m_{OBS}(q)]$, where $m_{OBS}(q)$ returns true if q's Datacenter equals DC-1, DC-3 or DC-5. The solver finds the following query:

```
(map (pair "Name" "www.example.com")
     (pair "ServiceTier" 1)
     (pair "Datacenter" "DC-1"))
```

The Topaz verifier always checks satisfiability for each policy when executed: if a policy is satisfiable, the verifier reports a matching query to the user; otherwise, if no matching query exists, the verifier returns a warning that the policy is "dead".

Property 2: Reachability. Reachability is similar to satisfiability but stronger: a policy is *reachable* if there is *some* query that will match it *given all other higher-ranked policies*. While satisfiability ensures that each policy will execute in isolation, reachability ensures that no policy prevents subsequent policies in the policy list from executing.

To illustrate this property, we check reachability of the first two policies in Figure 6. Checking whether ST1 is reachable is equivalent to checking whether it is satisfiable because there are no preceding policies. However, for the next policy, EXP, we ask the solver to satisfy the formula $\exists q[\neg m_{ST1}(q) \land m_{EXP}(q)]$. The solver finds the following query:

```
(map (pair "Name" "www.example.com")
     (pair "Datacenter" "DC-5")
     (pair "ServiceTier" 2))
```

which fails to match ST1 since its ServiceTier field is not 1, but matches EXP because it is destined for DC-5. The EXP's match function uses the sample built-in function to probabilistically choose to match routes, so, in practice, this query may or may not trigger the match. Reachability is a discrete property (either the policy has queries that match or doesn't), that the verifier checks by over-approximating comparisons against randomly-sampled variables as simply returning a Boolean decision (the sample was below the threshold).

Like satisfiability, checking reachability can warn operators when a new policy or different policy order makes a policy *unreachable* and "dead code". For instance, imagine that an operator ranked OBS below EXP. The verifier would prove that EXP is now unreachable by constructing a formula $\exists q[\neg m_{ST1}(q) \land \neg m_{OBS}(q) \land m_{EXP}(q)]$. It encodes $\neg m_{OBS}(q)$ as a conjunction of constraints with a clause (get "Datacenter" q) $\neq$ "DC-5". As $m_{EXP}(q)$ in turn has a constraint (get "Datacenter" q) = "DC-5", the entire conjunction reduces to false, making the formula unsatisfiable.

Property 3: Exclusivity. By checking reachability, operators can confirm that their policy ordering avoids the undesirable scenario where higher-ranked policies block all queries for a lower-ranked

Property	Logical formula
A is satisfiable	$\exists q[m_A(q)]$
B is reachable after A	$\exists q[\neg m_A(q) \land m_B(q)]$
A and B are exclusive	$\forall q[\neg m_A(q) \lor \neg m_B(q)]$
B matches all queries to A	$\forall q[m_A(q) \to m_B(q)]$
A and B are match-equivalent	$\forall q[m_A(q) \leftrightarrow m_B(q)]$

Table 1: Built-in Topaz-verifier properties

policy. That being said, if two policies match an *overlapping* set of queries Q, operators must choose which policy goes before the other (and handles Q). Operators may in some situations have multiple "high-priority" policies that they intend to use as the sole policies for *all* queries of a particular type. Reachability can confirm that these high-priority policies all have queries they match, but not that this set of queries is *distinct* from the set matched by another policy. To ensure that all high-priority policies have "sole control" over their matching queries, we must prove that these policies match *no* queries in common with one another, i.e., they are *exclusive*. The relative ordering of a group of exclusive policies is thus irrelevant.

To check exclusivity of two policies, the verifier checks that no query exists that matches *both* policies' match functions. For instance, consider the EXP and OBS policies. The solver checks whether $\exists q[m_{EXP}(q) \land m_{OBS}(q)]$. In this case, the solver finds the following *counterexample*, showing that these two policies are not exclusive:

```
(map (pair "Name" "www.example.com")
     (pair "Datacenter" "DC-5")
     (pair "ServiceTier" 1))
```

If operators updated OBS's augmentation to now exclude DC-5, EXP and OBS will no longer have datacenters in common. Thus, $\exists q[m_{OBS}(q) \land m_{EXP}(q)]$ will be false, so the verifier reports that these two policies are now exclusive.

To instruct the verifier to check exclusivity of a policy, operators set its *exclusive* flag to true. The verifier then performs pairwise checks between all *exclusive* policies. In practice, the first three (of our seven) production policies are marked exclusive, to check that they never conflict.

6.2 A Formal Model of Topaz Policies

We now describe the formal model of Topaz policies. Our initial set of verified properties is summarized in Table 1. Properties are expressed as logical formulas over one or more policies' match functions.

In the model, each Topaz policy's match function is a predicate over queries: For a policy A, $m_A(q)$ returns true if A matches query q, and false otherwise. Queries are associative arrays of key-value pairs, where keys identify the represented query field, e.g., the query's datacenter or service tier. The set of allowed strings for each string field is known *a priori*; for example, to identify three-character strings used by the CDN to represent particular datacenters, we assign each unique string a unique bitvector. Primitive types, such as Booleans, integers and strings, are encoded directly in logic or represented as bitvectors. Fixed width integers such as uint32 are encoded as bitvectors with the width of the

corresponding type in our Go implementation. Lists and sets of values are encoded as fixed-length lists, and maps as fixed-length lists of pairs. We inline all (augmented) configuration values into match functions prior to encoding.

The semantics of most expressions in the DSL follow the usual encoding, taking the `topaz-lang` Go interpreter as the source of truth. So far, all match and response functions, as well as built-in functions, have been compatible with our formal model. However, in practice, we did need to make two approximations to ensure that verification was decidable. First, consider that queries have a variable-length list of *flags* or labels that are attached by the DNS nameserver. To make verification decidable over this list, we assume a safe upper bound k on the length of this list (large enough to include all flags we may realistically see in practice). Also, as already mentioned, we over-approximate the semantics of randomly sampling queries, as well as from an enumerable L. [4] We use a fresh symbolic variable l constrained to be one of the elements in L to consider *every* possible choice of element in the enumerable. Doing so precludes reasoning over probabilities, which is appropriate because the primary interest is in discrete properties of match functions (e.g., if a query exists that will match).

6.3 Topaz-verifier Implementation

Topaz-verifier is implemented in 1,800 lines of the solver-aided DSL Rosette [47], a dialect of Racket [38]. The verifier's `topaz-lang` interpreter is distinct from the `topaz-lang` interpreter written in Go—we discuss the tradeoffs in §8.1.

The implementation consists of

(1) a tree-walk interpreter to encode match functions as formulas (see §6.2);
(2) a library of properties to be checked (Table 1); and
(3) a frontend to submit configurations for checking the properties.

The property functions use Rosette's `solve` and `verify` features to ask its underlying SMT solver (Z3 [16] or CVC4 [6]) if the property holds over a symbolic query. The library can be easily extended to support more properties.

7 PERFORMANCE EVALUATION

Topaz is currently deployed in production at a global anycast CDN. It has improved the transparency and agility of DNS objectives, but has performance implications. We now evaluate Topaz's performance. On balance, the CDN finds Topaz's overheads acceptable given its benefits.

7.1 Topaz-edge Overheads

We quantify the latency that Topaz-edge (and policies) add to CDN DNS queries using production latency measurements and local microbenchmarks.

Proportion of production DNS latency. We begin by calculating the proportion of time each DNS query spends contacting Topaz-edge. We instrument all nameservers in the CDN to record the total time to resolve incoming DNS queries, and also the time between forwarding queries to Topaz-edge and receiving a response. The

---

Percentile	Topaz-edge Overhead (includes IPC)	Topaz-edge Overhead (excluding IPC)
99	0.07%	0.02%
95	0.36%	0.16%
75	1.92%	0.52%
50	4.73%	0.71%
25	9.68%	1.13%

Table 2: Proportion of the full authoritative query-response time spent on Topaz-edge, including and excluding IPC.

portion of time spent on Topaz-edge both including and excluding inter-process communication (IPC) is shown in Table 2, and leads to two observations. First, Topaz-edge is understandably less susceptible to load-induced variability than the much larger authoritative DNS process. Second, IPC increases Topaz-edge overhead by a factor of ~2-9. While Topaz may be merged into the nameserver in the future to eliminate IPC, the CDN deems the current performance overheads acceptable.

Microbenchmarking `topaz-lang` policies. We also measure performance of Topaz-edge in both Go (used for earlier versions of Topaz; see §4) and `topaz-lang` (§5) as we increase the number of deployed policies. We produce these results on a development machine because we cannot artificially inflate the number of policies in production. Instead we resort to local We deploy Topaz in a Docker container that emulates the edge DNS environment, on an M1 (arm64) with 10 cores and 32GB of memory. We construct a workload consisting of 10 clients that each send 1000 queries (10K total) to Topaz over a single gRPC connection. We repeat the process for N= [1, 10, 20...100] policies, where the first N−1 policies are a modified version of the slowest deployed policy at the CDN; only the Nth policy matches the query.

We show the results (ms) in Table 3. Each cell includes one normal typeface measurement in which all policies were expressed in Go, and one boldface measurement below in which all policies were expressed in `topaz-lang`. As expected, performance variation between 25th and 99th percentiles increase with the number of policies evaluated. However, reading from left to right suggests that Topaz-edge scales well, with a small penalty for executing in `topaz-lang`.

7.2 Topaz Verifier Performance

Our policy verifier runs as part of the CI pipeline and executes whenever operators modify the policy file. The verifier checks that every policy is satisfiable and reachable; only policies marked exclusive are checked for exclusivity. The end-to-end time to build, validate, and verify (including the verifier's Docker build time) on the seven active policies is about 45 seconds. Verification takes 5–6 seconds, a small fraction of the total.

We also measure *worst-case* verification performance as the number of policies increases. We generated an artificial policy file with N= [1, 10, 20, .., 100] policies; the first $\frac{N}{2}$ policies are marked exclusive. Results are shown in Figure 8 by each verified property. The fastest property to verify is satisfiability since the verifier can determine if there exists a matching query for each policy independent of the others. Reachability must evaluate all policies

---

[4] Enumerables may be ranges of integers, lists, sets or maps.

Percentile	1	10	20	30	40	50	60	70	80	90	100
99	13.19	14.83	16.95	17.42	17.1	19.67	19.13	23.38	21.5	20.1	20.92
	14.3	**16.68**	**18.56**	**18.77**	**20.33**	**23.38**	**21.85**	**23.82**	**21.67**	**25.03**	**26.13**
95	5.96	6.58	7.83	8.71	8.48	9.6	10.15	10.42	11.64	11.77	12.82
	5.83	**7.29**	**8.12**	**9.41**	**11.34**	**12.78**	**11.83**	**13.22**	**14.31**	**14.82**	**15.04**
75	1.64	1.89	2.18	2.64	2.38	2.96	3.26	3.41	3.68	3.8	4.22
	1.58	**1.97**	**2.34**	**2.87**	**3.58**	**3.96**	**4.11**	**4.68**	**5.57**	**5.68**	**5.81**
50	0.99	1.06	1.12	1.23	1.15	1.31	1.42	1.55	1.6	1.63	1.91
	0.95	**1.08**	**1.23**	**1.38**	**1.66**	**1.78**	**1.91**	**2.11**	**2.58**	**2.56**	**2.81**
25	0.66	0.69	0.72	0.76	0.72	0.79	0.82	0.9	0.91	0.89	0.99
	0.64	**0.71**	**0.8**	**0.87**	**0.99**	**1.02**	**1.09**	**1.15**	**1.35**	**1.36**	**1.49**

Table 3: Topaz-edge microbenchmarks (in ms) with 1 to 100 policies. Values in normal typeface are measurements for policies expressed in Go. Values in boldface are measurements when all policies are expressed in `topaz-lang`.
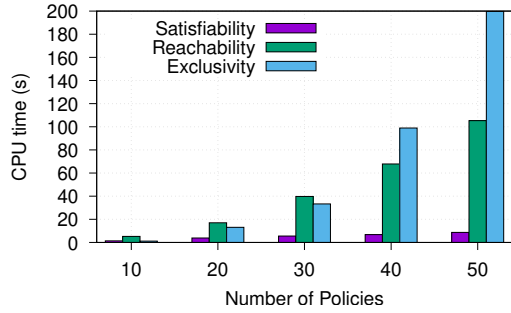
Figure 8: Verifier runtime as number of policies increase, according to property verified. Policies are modified from the slowest production policy, and only the last policy in the list matches. The first $\frac{N}{2}$ policies are marked `exclusive`.

up to and including the policy being checked. Exclusivity incurs greater penalties as exclusive policies increase in number, since comparisons are pairwise with every exclusive policy. In Figure 8 the N=40 policies file has 20 policies marked exclusive, which takes ~100s to verify—a fraction of time needed by a human to manually review and test, and who may make mistakes. Policy changes are also infrequent so, on balance, relatively longer verification times are deemed acceptable.

Certain changes may not require re-verification of all policies. For instance, certain exclusivity and reachability results can be cached. We have yet to explore these optimizations because there is currently no need.

8 EXPERIENCE WITH TOPAZ

In this section we consider the qualitative impact of Topaz at the CDN, including the lessons we learned designing and implementing Topaz, as well as things we might have done differently. We also examine specific instances in which Topaz had a major impact on operations at the CDN.

8.1 Topaz in Production

At time of writing, Topaz-edge processes (globally) ~1M queries per second (qps), from which ~100K qps trigger a policy match function. Topaz has replaced multiple functions of the CDN's legacy assignment system, and efforts are currently underway to replace more. There are seven policies active and deployed. Three policies are for service isolation (§4.1), two are for experimentation (§4.2), and two for observability (§4.3). An eighth policy was retired after completing a measurement study [20], while an additional three policies are in development. We estimate that Topaz will house about 20 policies in the medium-term.

As of late 2023, all active policies are expressed in `topaz-lang`, together comprising 265 lines of policy configuration (i.e., match, response, and parameters). Each policy's match function is on average ten lines of `topaz-lang` code, while response functions are six lines on average.

Since all policies are written in `topaz-lang`, every change to any policy is checked by the formal verifier, which executes in a CI pipeline. From the active policies, the first three are marked and checked for *exclusivity*: the verifier confirms that there exists no query that would trigger more than one of these policies—this makes the order of these policies irrelevant. The remaining four policies are non-exclusive, so their order is semantically important. After months in the CI pipeline the verifier has yet to report policy conflicts or violations. Its integration has increased confidence of engineers in the reliability of policies that are active and deployed.

8.2 Lessons Learned: Strengths & Weaknesses

We continue to learn various strengths and weaknesses of Topaz's design and deployment. We highlight both to aid future practitioners' systems with similar goals, and identify opportunities for further research.

Strength: match/action paradigm. The decision to split policies into a separate match function, a response function, and separate configuration block, has yielded significant dividends. This finding reinforces the paradigm's generality from prior network policy languages [3, 10, 13, 24, 43]. First, it makes policies easier to skim (just looking at match functions) to determine which policy will run given a query. Second, since the verifier only verifies match

functions, it only needs to be changed when a new Topaz built-in function is used in a match function. Almost all new built-in functions added to `topaz-lang` were only needed for configuration or response functions: thanks to the separation of policies into these three components, the verifier did not require any changes in these cases.

Weakness: separate model-checker. Topaz's verifier was developed as a separate tool from Topaz-edge and Topaz-core, with its own `topaz-lang` interpreter and verifier written in Rosette/Racket. To ensure semantic equivalence, engineers have to maintain the verifier alongside Topaz-edge and Topaz-core. Inconsistencies in the two implementations may cause the verifier to fail or provide incorrect results if it falls out-of-sync with the Go implementation. So far, this has not been a significant issue because 1) the `topaz-lang` language itself has not changed and is not expected to change, and 2) the suite of built-in functions used in *match* functions rarely changes, as aforementioned. In the future, we may perform differential testing of the two implementations like in Cedar [14]. Topaz makes a case for formal methods in production, but effective integration and maintenance remain a hard problem.

Strength: centralized configuration, distributed execution. Much like Software-Defined Networking systems [10], we have found that developing, maintaining, and compiling (augmenting) configuration in a central location and then distributing that configuration to servers to be a good design pattern. The control plane is simple—distribute the policy configuration file to all edge servers—but the configuration values can be constructed (via augmentation) using arbitrary data sources and computation. Furthermore, running the same policy file in every datacenter (as opposed to different policies in different datacenters) simplifies operations greatly, considering the CDN maintains a single software stack across all machines. Distributing the configuration over a fast control plane fabric (Quicksilver) also makes it easy to roll the configuration forward or backward.

Weakness: Scheme-like syntax. We chose an S-expression-based syntax because it is easy to parse and evaluate. We decided early on that this might have been the wrong choice—for those engineers that lack familiarity with Scheme/Lisp languages, the prospect of editing the policy file is that much more daunting. While an informal poll of the engineers who maintain and operate Topaz said they found it strange at first but ultimately the right tool for the job, we still believe a more approachable syntax would make Topaz more palatable to non-experts.

8.3 Case Studies

We use two case studies to illustrate how Topaz has increased the agility of authoritative DNS at the CDN.

Encrypted Client Hello (ECH). Using Topaz, CDN operators safely and globally tested an emerging TLS extension in draft at the IETF that encrypts the TLS ClientHello [40]. Servers announce ECH support to clients via the DNS, using an *ECHConfig* structure in the HTTPS resource record for the supported name. ECH establishes an anonymity set defined by all the names that share the same key information in their respective resource records. In Topaz, this makes *ECHConfig* management equivalent to address selection;

only minor additions to Topaz enabled support for ECH policies that returned common *ECHConfig* structures. The changes were ready by the time ECH had been implemented in the CDN servers.

Developers used Topaz over a three-month period to announce ECH support for an incrementally increasing number of names and datacenters. In late stages, and close to general release, a latent bug in an unrelated system caused ECH to fail in a small number of cases. Operators resolved the failure by reducing the application of the ECH policy to zero percent, which took effect at edge servers in seconds.

Mitigating upstream failure by re-addressing. Topaz can help operators mitigate upstream failures. In one example, a misconfiguration in a neighboring network caused a large routing failure in a small number of datacenters, affecting a small number of the CDN's anycasted prefixes. Since the upstream network is connected to the CDN at multiple IXPs, one obvious mitigation would have been to withdraw those prefixes via BGP from only the affected datacenters, leaving the prefixes reachable at other datacenters. However, it was unclear how these withdrawals would affect systems upstream of the failure, or if the upstream failure might also affect other datacenters.

Instead, operators recognized that the affected prefixes were employed by a single Topaz policy, and that the failure could be mitigated by changing that policy's prefixes to unaffected ones. Operators made this sweeping set of address changes with a one-line configuration edit.

9 RELATED WORK

CDN-scale network management. Taiji [12], Facebook's system for managing global user traffic, uses a constraint optimization solver to map portions of traffic from edge nodes to datacenters. Taiji maps fractions of traffic each edge node will forward to data centers based on service-level objectives specified in *policies*. Facebook has another system, Edge Fabric [41], designed for optimized routing of *egress* traffic from a given datacenter.

AT&T uses CORNET [34] to manage change across heterogeneous network infrastructure by decomposing changes into reusable, composable "building blocks". Similarly, Facebook uses Robotron [46] to manage a large production network in top-down fashion. Developers express high-level intent using an object-relational mapping (ORM), which the system compiles to proprietary configuration formats.

Microsoft's Footprint [29] showed that making routing decisions based on information about datacenters, edge nodes, and wide-area networks can minimize congestion, among other factors. Google uses Espresso [50] to achieve the benefits of Software-Defined Networking (SDN) to their peering edge network, which runs arbitrary hardware and software.

Network programming & verification. Prior work on verifiable network programming languages [4, 7, 21, 22, 25, 26, 32, 39] has focused primarily on forwarding (the data plane) and routing (the control plane). These languages have different goals from Topaz: they analyze the network's packet forwarding behavior (data plane) or routing protocols (control plane). For example, NetKAT [4] defines a language for checking end-to-end properties of switch-level

forwarding behavior. Our decomposition of policies into match and response functions resembles the match-action rules used in forwarding and access control [26], as well as conditions and actions in IETF's Intent Based Networking [13] RFC. We believe these similarities confirm the legitimacy of our proposal. Our verification procedure follows a standard approach [32, 48] of embedding a DSL in a solver-aided or logic programming language (Rosette [47] in our case) to encode DSL programs in formal logic.

Formal Models of DNS. Researchers have also sought to formally verify properties of DNS resolution using traditional *static* DNS nameservers [27, 28, 30]. These properties may include rewrite loops, inconsistent answers or query latency. GRoot [27] and Liu and Duan et al. [30] present formal models of DNS that can find violations of these properties. These techniques can also be used for testing that DNS implementations comply to formal models—the tool SCALE [28] does so using the formal model of GRoot. These approaches target an orthogonal problem to Topaz: finding bugs in the responses of DNS infrastructure. Extending Topaz-verifier's model to check properties of DNS responses would potentially allow it to also check properties like these ones: otherwise, one could use tools like GRoot or Liu and Duan et al.'s to check that the resulting DNS behavior while using Topaz respects desirable resolution properties.

DNS scripting. Some authoritative DNS nameservers support executing scripts in response to certain queries. In LuaDNS [1] users can write zone files in standard Bind format or as Lua scripts checked into git, though there is no decomposition of scripts into distinct and verifiable policies. PowerDNS [2] is a scriptable DNS resolver (not nameserver) that also has Lua scripting. Most similar to Topaz is Bunny DNS [36], to which end-users can upload code that their nameserver executes in response to certain queries. While Topaz also executes code in response to queries, we focus on effectively and safely combining disparate policies written by different stakeholders.

10 CONCLUSION

Topaz was built because it no longer makes sense for modern CDNs, even anycast CDNs, to think about static assignments from query to IP. The anycast CDN in question manages millions of addresses and many more domain names. At that scale, it is more advantageous to group queries that share context and circumstances, rather than simply names. Topaz achieves this by classifying queries using live programs, i.e., match functions. CDNs also need more dynamic mechanisms for assigning IPs to those groups of queries—hence Topaz's response functions. Furthermore, the IPs themselves are no longer consequential, since they can be trivially rotated with configuration changes.

We are optimistic about Topaz's impact as its adoption and development continue. Many CDN engineers are optimistic about guarantees provided formal verification because making changes to DNS often causes large-scale outages [45, 51]. No doubt verification will not prevent all problems—as one engineer pointed out, verification tools can themselves cause bugs—but we look forward to reporting on the challenges and successes of formal verification at scale in future work.

11 ACKNOWLEDGEMENTS

REFERENCES

[1] LuaDNS. https://www.luadns.com/.
[2] PowerDNS. https://doc.powerdns.com/recursor/index.html.
[3] Policy-based control for cloud native environments, 2021.
[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. *ACM SIGPLAN Notices*, 49(1):113–126, 2014.
[5] Abbie Barbir, Brad Cain, Raj Nair, and Oliver Spatscheck. Known Content Network (CN) Request-Routing Mechanisms. Technical report, 2003.
[6] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.
[7] 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.
[8] Thomas P. Brisco. DNS Support for Load Balancing. RFC 1794, April 1995.
[9] Matt Calder, Ashley Flavel, Ethan Katz-Bassett, Ratul Mahajan, and Jitendra Padhye. Analyzing the performance of an anycast cdn. In *Proceedings of the 2015 Internet Measurement Conference*, pages 531–537, 2015.
[10] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. *ACM SIGCOMM computer communication review*, 37(4):1–12, 2007.
[11] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. End-User Mapping: Next Generation Request Routing for Content Delivery. *ACM SIGCOMM Computer Communication Review*, 45(4):167–181, 2015.
[12] David Chou, Tianyin Xu, Kaushik Veeraraghavan, Andrew Newell, Sonia Margulis, Lin Xiao, Pol Mauri Ruiz, Justin Meza, Kiryong Ha, Shruti Padmanabha, et al. Taiji: Managing Global User Traffic for Large-Scale Internet Services at the Edge. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 430–446, 2019.
[13] Alexander Clemm, Laurent Ciavaglia, Lisandro Zambenedetti Granville, and Jeff Tantsura. Intent-Based Networking - Concepts and Definitions. RFC 9315, October 2022.
[14] Joseph W Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John Kastner, Anwar Mamat, et al. Cedar: A new language for expressive, fast, safe, and analyzable authorization. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):670–697, 2024.
[15] Gonçalo Grilo David Tuber, Luke Orden. How Cloudflare's systems dynamically route traffic across the globe, September 2023. https://blog.cloudflare.com/meet-traffic-manager.
[16] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.
[17] John Dilley, Bruce Maggs, Jay Parikh, Harald Prokop, Ramesh Sitaraman, and Bill Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.
[18] Marwan Fayed, Lorenz Bauer, Vasileios Giotsas, Sami Kerola, Marek Majkowski, Pavel Odintsov, Jakub Sitnicki, Taejoong Chung, Dave Levin, Alan Mislove, Christopher A. Wood, and Nick Sullivan. The Ties That Un-Bind: Decoupling IP from Web Services and Sockets for Robust Addressing Agility at CDN-Scale. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 433–446, New York, NY, USA, 2021. Association for Computing Machinery.
[19] Ashley Flavel, Pradeepkumar Mani, David Maltz, Nick Holt, Jie Liu, Yingying Chen, and Oleg Surmachev. FastRoute: A Scalable Load-Aware Anycast Routing Architecture for Modern CDNs. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 381–394, 2015.
[20] Withheld for anonymous review, October 2022.
[21] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. *ACM Sigplan Notices*, 46(9):279–291, 2011.
[22] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. Nv: An intermediate language for verification of network control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 958–973, 2020.
[23] Petros Gigis, Matt Calder, Lefteris Manassakis, George Nomikos, Vasileios Kotronis, Xenofontas Dimitropoulos, Ethan Katz-Bassett, and Georgios Smaragdakis. Seven Years in the Life of Hypergiants' Off-Nets. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 516–533, 2021.
[24] Timothy L Hinrichs, Natasha S Gude, Martin Casado, John C Mitchell, and Scott Shenker. Practical declarative network management. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 1–10, 2009.
[25] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, Praveen Tammana, and David Walker. Contra: A programmable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020*, 2020.
[26] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. Automated analysis and debugging of network connectivity policies. *Microsoft Research*, pages 1–11, 2014.

[27] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. Groot: Proactive verification of dns configurations. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 310–328, 2020.

[28] Siva Kesava Reddy Kakarla, Ryan Beckett, Todd Millstein, and George Varghese. SCALE: Automatically Finding RFC Compliance Bugs in DNS Nameservers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 307–323, 2022.

[29] Hongqiang Harry Liu, Raajay Viswanathan, Matt Calder, Aditya Akella, Ratul Mahajan, Jitendra Padhye, and Ming Zhang. Efficiently Delivering Online Services over Integrated Infrastructure. page 15, 2013.

[30] Si Liu, Huayi Duan, Lukas Heimes, Marco Bearzi, Jodok Vieli, David Basin, and Adrian Perrig. A formal framework for end-to-end dns resolution. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 932–949, 2023.

[31] Zhenhua Liu, Minghong Lin, Adam Wierman, Steven H Low, and Lachlan LH Andrew. Greening Geographical Load Balancing. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):193–204, 2011.

[32] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 499–512, 2015.

[33] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.

[34] Ajay Mahimkar, Carlos Eduardo de Andrade, Rakesh Sinha, and Giritharan Rana. A Composition Framework for Change Management. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 788–806. ACM, August 2021.

[35] Erik Nygren, Ramesh K Sitaraman, and Jennifer Sun. The Akamai Network: A Platform for High-Performance Internet Applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[36] Dejan Grofelnik Pelzel. We're transforming internet routing: Introducing Bunny DNS!, March 2022. https://bunny.net/blog/transforming-internet-routing-introducing-bunny-dns/.

[37] Geoffrey Plouvier. Introducing Quicksilver: Configuration Distribution at Internet Scale, March 2020. https://blog.cloudflare.com/introducing-quicksilver-configuration-distribution-at-internet-scale/.

[38] Racket. The Racket programming language, 2023.

[39] Mark Reitblatt, Marco Canini, Arjun Guha, and Nate Foster. FatTire: Declarative fault tolerance for software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, page 109–114, New York, NY, USA, 2013. Association for Computing Machinery.

[40] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. TLS Encrypted Client Hello. Internet-Draft draft-ietf-tls-esni, Internet Engineering Task Force, October 2023. Work in Progress.

[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, 2017.

[42] Kyle Schomp, Onkar Bhardwaj, Eymen Kurdoglu, Mashooq Muhaimen, and Ramesh K Sitaraman. Akamai DNS: Providing Authoritative Answers to the World's Queries. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 465–478, 2020.

[43] OASIS Standard. extensible access control markup language (xacml) version 3.0. *A:(22 January 2013). URl: http://docs. oasis-open. org/xacml/3.0/xacml-3.0-core-spec-os-en. html*, 2013.

[44] Ao-Jan Su, David R Choffnes, Aleksandar Kuzmanovic, and Fabian E Bustamante. Drafting Behind Akamai: Inferring Network Conditions Based on CDN Redirections. *IEEE/ACM transactions on networking*, 17(6):1752–1765, 2009.

[45] Mani Sundaram. Akamai Summarizes Service Disruption (RESOLVED), 2021. https://www.akamai.com/blog/news/akamai-summarizes-service-disruption-resolved.

[46] 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*, pages 426–439. ACM, August 2016.

[47] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. *ACM SIGPLAN Notices*, 49(6):530–541, 2014.

[48] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an smt solver. In *OOPSLA*, page 765–780, New York, NY, USA, 2016. Association for Computing Machinery.

[49] David Wragg. Unimog - Cloudflare's edge load balancer, September 2020. https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer.

[50] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 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, Los Angeles CA USA, August 2017. ACM.

[51] Ólafur Guðmundsson. 1.1.1.1 lookup failures on October 4th, 2023, 2023. https://blog.cloudflare.com/1-1-1-1-lookup-failures-on-october-4th-2023/.