

# SCULPTOR Has Your Back(up Path): Carving Interdomain Routes to Services

Paper #156, 12 pages body, **TBD: 19** pages total

## Abstract

Large cloud and content (service) providers help serve an expanding suite of applications that are increasingly integrated with our lives, but have to contend with a dynamic public Internet to route user traffic. To enhance reliability to dynamic events such as failure and DDoS attacks, large service providers overprovision to accommodate peak loads and reactively activate emergency systems for shifting excess traffic. We take a different approach with SCULPTOR, which proactively installs routes so that Service Providers can use their global resources to handle dynamic scenarios without excessive overprovisioning. SCULPTOR ensures users have many interdomain routes to the Service Provider, and moves excess traffic to backup paths to avoid overloading and optimize general objectives. SCULPTOR models Internet routing to solve a large integer optimization problem at scale using gradient descent. We prototyped SCULPTOR on a global public cloud and tested it in real Internet conditions, demonstrating that SCULPTOR handles dynamic loads 28% larger than other solutions using existing Service Provider infrastructure, reduces overloading on links during site failures by up to 40%, and enables service providers to achieve low latency objectives for up to 17% more traffic.

## 1 Introduction

Compiled on 2024/09/12 at 10:39:53

Cloud and content providers (hereafter Service Providers) enable diverse Internet applications used daily by billions of users, and those applications have diverse requirements. For example, enterprise services have tight reliability requirements [50, 47], whereas new applications such as virtual reality require  $\leq 10$  ms round trip latency [71] and  $\leq 3$  ms jitter [93]. It may therefore make sense for different Service Providers to allocate resources in different ways to meet these requirements to varying degrees.

Service Providers steer traffic over different end-to-end paths through the public Internet to meet these goals. For

example, Service Providers shift egress (out of the Service Provider’s network) demands across links to meet performance objectives [86, 103, 37, 54, 90]. However, it is difficult to modify the path traffic takes from users to the Service Provider (*i.e.*, ingress traffic) with the same precision as in the egress case since that path can only be controlled indirectly, by conducting BGP announcements and directing user traffic to different prefixes.

Complicating matters, Service Providers must meet these requirements subject to changing conditions such as peering link/site failures [30, 66], DDoS attacks [74, 101, 82], flash crowds [42, 55, 58], new applications/business priorities (LLMs), and route changes [72, 64, 36, 65, 76, 66]. Critically, such changes can cause *overload* if the service cannot handle new traffic volumes induced by the change. This overload can lead to degraded service for users, hurting reliability [73, 80, 51, 47].

To meet different goals and respond to changing conditions, Service Providers propose solutions that are either (a) too conservative, (b) too specific, or (c) too reactive. For example, Service Providers proactively overprovision resources [94, 1, 60], but we demonstrate using traces that bursty traffic patterns can require conservative overprovisioning rates as high as 70% (§2.3). Systems such as AnyOpt and PAINTER proactively set up routes to optimize specific objectives such as steady-state latency [104, 50], but those approaches have not been demonstrated to optimize other important objectives. FastRoute and TIPSy respond to changing loads to retain reliability [30, 66], but only do so reactively leading to short-term degraded performance. It is also unclear both how to combine these approaches (*e.g.*, retaining low latency under changing traffic loads) and how to apply approaches from one domain to another (*e.g.*, applying egress traffic cost reduction systems to ingress traffic [90]).

We present Service Providers with a flexible framework, SCULPTOR (Scouring Configurations for Utilization-Loss-and-Performance-aware Traffic Optimization & Routing), which accepts as input cost, performance, and reliability objectives and outputs BGP advertisements and traffic allocations that

help achieve those desired objectives. Service Providers can specify constraints such as link and site capacities, and optimize objectives such as maximum link utilization, transit cost, latency, and latency under partial (*e.g.*, single link) failures, and combinations thereof. SCULPTOR allows Service Providers to specify traffic patterns or failure scenarios to consider (with varying importances), but good performance also generalizes to unseen scenarios.

We provide evidence that SCULPTOR works for objectives that can be met in intradomain settings such as latency reduction, cost reduction, and maximum link utilization, and works for constraints such as link and site capacities, all at the scale of today’s Service Providers. SCULPTOR could cover a broader set of objectives, constraints, and problem sizes but we leave such formal characterization to future work.

To solve each optimization problem, SCULPTOR efficiently searches over the large BGP advertisement search space ( $> 2^{10,000}$  possibilities) by modeling how different strategies perform, without having to predict the vast majority of actual paths taken under different configurations since predicting interdomain paths is hard and measuring them is slow (§3.3). SCULPTOR then optimizes these (modeled) performance metrics using gradient descent, which is appropriate in our setting due to the high dimension of the problem and the parallelism that gradient descent admits (§3.4). This modeling enables SCULPTOR to assess 13M configurations (6,000 $\times$  more than other solutions) while only measuring tens in the Internet (§5.5).

We prototype and evaluate our framework at Internet scale using the PEERING testbed [85] (§4), which is now deployed at 32 Vultr cloud locations [98]. Vultr is a global public cloud that allows our prototype to issue BGP advertisements via more than 10,000 peerings. We evaluate our framework on a specific optimization problem that proactively provides cost and latency benefits, even under failure scenarios, demonstrating the framework’s general utility.

We compare SCULPTOR’s performance on this specific problem to that of an unreasonably expensive “optimal” solution (computing the actual optimal is infeasible). We found that, compared to other approaches, SCULPTOR increases the amount of traffic within 10 ms of the optimal by 3% in steady-state (§5.2), by 11% during link failure, and by 17% during site failure. SCULPTOR also reduces overloading on links during site failures by 17% (on average over all tested scenarios), giving Service Providers more confidence that services will still be available during partial failure (§5.3).

Providing good backup paths to handle changing conditions improves more than just latency — we find that by load balancing traffic on backup paths during peak times, we can satisfy very high peak demands with the same infrastructure. SCULPTOR can handle flash crowds (DDoS attacks, for example) at more than 3 $\times$  expected traffic volume, drastically reducing the amount of overprovisioning that Service Providers need, thus reducing costs. SCULPTOR can also han-

dle large diurnal swings of almost 2 $\times$  expected load (§5.4). Hence, SCULPTOR can serve more traffic with less overloading and do so with almost the same latency that an optimal (but unreasonably expensive) solution can.

In an era of complex, global, distributed systems, Service Providers want unified frameworks especially if those frameworks provide better results. SCULPTOR provides Service Providers with a unified framework to proactively configure good routes to optimize the routing objectives that matter to them, preparing Service Providers to provide the increasingly reliable, performant service that our applications need.

## 2 Background and Motivation

**Setting:** Service Providers offer their services from tens to hundreds of geo-distributed sites [68]. The sites for a particular service can serve any user, but users benefit from reaching a low latency site for performance. Sites consist of sets of servers which have an aggregate capacity. Sites can forward requests to other sites via private WANs, but such forwarding is undesirable since it uses valuable private WAN capacity [47]. Service Providers also connect to other networks at sites via dedicated links or shared IXP fabrics. Each such link also has a capacity. When utilization of a site or link nears/exceeds the capacity, performance suffers, so Service Providers strive to avoid very high utilization [30, 16]. Resources can also fail completely due to, for example, physical failure and misconfiguration.

Upgrading capacity has fixed costs (*e.g.*, fiber, router backplane bandwidth, line card upgrades, CDN servers near peering routers) and variable costs that scale with demand (*e.g.*, power, transit), both of which Service Providers strive to keep low [78, 1, 16, 90]. We model deployment cost as correlated with the peering capacity over all links/sites, even though the actual relationship may be complex.

### 2.1 Variable, Evolving Goals

Evolving Internet use cases are pushing Service Providers to meet diverse requirements for their applications. For example, Service Providers increasingly host mission-critical services such as enterprise solutions [50], which require very high reliability and are predicted to be a \$60B industry by 2027 [41]. Gaming is a similarly important industry generating more revenue than the music and movie industries combined [5], but instead requires latency within 50 ms [71]. Service Providers are also expanding the set of services they provide — for example, CDNs who traditionally hosted static content are pivoting to offering services like general compute [20, 28, 27].

As use cases evolve, Service Providers increasingly need to meet performance requirements for ingress traffic. As Service Providers move away from CDN business models, ingress traffic moves from simply TCP acknowledgements to critical, performance-sensitive traffic such as player movements in

real-time games, voice and video in enterprise conferences, and video/image uploads for AI processing in the cloud [18, 69].

Despite Service Providers’ efforts to meet variable service requirements [66, 46, 35, 53, 70, 50, 60], meeting requirements is still challenging often due to unexpected changes *outside* Service Provider networks. For example, peering disputes can lead to congestion on interdomain links and inflated paths [30, 22, 25], and DDoS attacks still bring down sites/services [82, 73, 80, 51], despite the considerable effort in mitigating DDoS attack effects [74, 101]. Moreover, recent work shows that user traffic demands are highly variable due to flash crowds and path changes and so are much harder to plan for than inter-datacenter demands [58, 76, 16]. Operators regularly report these traffic surges on social media and blog posts [42, 55, 72, 64, 36, 65].

## 2.2 Routing Traffic to Service Providers

A way to protect against these changes is to route traffic over different interdomain paths thus avoiding overwhelmed sites/links and balancing load. Existing systems balance traffic over egress interdomain paths [86, 103, 37, 54], but it remains unclear how to configure and balance traffic across a good set of ingress interdomain paths.

**Ingress Path Model:** Service Providers lack full control of which interdomain path traffic takes since BGP, the Internet’s interdomain routing protocol, computes paths in a distributed fashion, giving each intermediate network a say in which paths are chosen and which are communicated to other networks. Service Providers can, however, advertise their reachability to peers/providers (*i.e.*, “expose paths”) in different ways to increase the chance of there being good paths for users. Service Providers can then balance traffic across paths using traffic engineering (§3.2.1).

We model interdomain paths from users to the Service Provider as non overlapping, except at the peering link through which each path ingresses to the Service Providers’ deployment. We assume the peering link is the bottleneck of each path, as the actual bottleneck is difficult to measure, and that users share the capacity of those peering links. (We leave relaxing these assumptions for future work — see the remark at the end of this subsection). This link is one of many at a particular site. Hence a user may have many paths to a site, each with different capacities and latencies. A path to a given prefix will be through one of the corresponding ingress links over which that prefix is advertised.

**Current Ingress Routing Approaches:** Today, most Service Providers either use *anycast* prefix advertisements to provide relatively low latency and high availability at the expense of some control [8, 49, 108, 107], or *unicast* prefix advertisements to direct users to specific sites [52, 86, 13].

*Anycast*, where Service Providers advertise a single prefix to all peers/providers at all sites, offers some natural availabil-

ity following failures since BGP automatically reroutes most traffic to avoid the failure after tens of seconds [108]. Prior work shows that this availability comes with higher latency in some cases [59, 49]. *Unicast* can give clients lower latency than *anycast* by advertising a unique prefix at each site [13], but can suffer from reliability concerns potentially taking as much as an hour to shift traffic away from bad routes [50].

**Remark on Our Path Model:** Although we do not explicitly account for the possibility of bottlenecks in the public Internet, we implicitly account for it by presenting a framework that gives good routes even under changing conditions. Paths with unexpectedly low bottleneck capacities can be seen as a “changing condition”, and one could use congestion-detection mechanisms along paths to trigger traffic shifts to other, healthier (backup) paths. Other prior work on ingress [30, 104, 50] and egress traffic engineering [103, 37, 86, 54] also does not explicitly address this problem.

## 2.3 Limitations of Current Approaches

**Too Specific:** Some recent work has noted that specific objectives may be better met by offering better interdomain routes, achieving better latency and/or reliability [30, 8, 104, 107, 50]. These solutions advertise different prefixes to subsets of all peers/providers to offer users more paths. However, these solutions optimize specific objectives, mostly steady-state latency, and in Section 3.4 we discuss why it is challenging to extend these approaches to consider other objectives.

Figure 1 shows how this specific focus on a single objective could lead to reliability issues, *e.g.*, during a deployment failure. In normal operation, user traffic is split evenly across two prefixes and achieves low latency (Fig. 1a). However when site B fails, BGP chooses the route through Provider 1 for all traffic, causing link overutilization and subsequent poor performance for all users (Fig. 1b). In Section 5 we show that this overload happens in practice for systems that provide very low (steady-state) latency from users to Service Provider networks such as *AnyOpt* [104] and *PAINTER* [50], since those systems optimize for specific objectives.

**Too Reactive:** To enhance reliability, some Service Providers propose systems that react to changing conditions — for example, *FastRoute* and *TIPSY* drain traffic from overloaded links/sites. However, reactivity still requires time to change to a new state, leading to short-term service degradation and additional uncertainty at a time where unexpected things are already occurring. Reacting by modifying BGP announcements or DNS configurations as *TIPSY* and *FastRoute* do is also undesirable since these actions are known to lead to outages [6, 44, 10, 60].

**Too Conservative:** Another common approach is to over-provision resources to handle transient peak loads [94, 1, 60], but Figure 2 demonstrates that doing so can incur excessive costs. Figure 2 computes differences in peak utilization on links between different successive time periods, using longi-

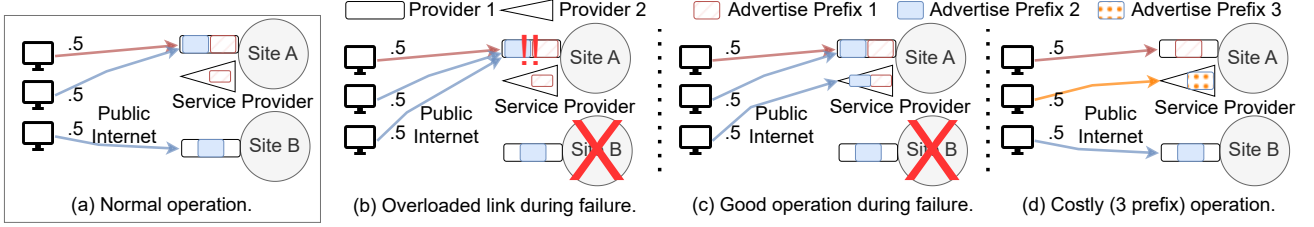


Figure 1: In normal operation traffic is split between two sites by directing half the traffic to each prefix (a). When site B fails, there is enough global capacity to serve all traffic (each link can handle 1 unit) but no way to split traffic across multiple providers given available paths leading to link overload (b). A *resilient* solution is to advertise prefix 2 to an additional provider at site A, allowing traffic splitting across the two links (c). A simpler but prohibitively expensive solution is to advertise one prefix per peering (d).

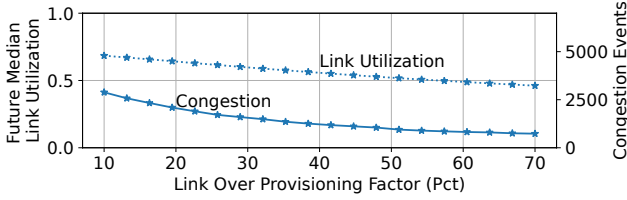


Figure 2: Planning for peak loads to avoid overloading requires inefficient overprovisioning.

tudinal link utilization data from OVH cloud [77].

To generate Figure 2 we first split the dataset into successive, non-overlapping 120-day planning periods and compute the 95<sup>th</sup> percentile link utilization (“near-peak load”) for each link and for each period. The dataset reports link utilizations over approximately 5-minute intervals. We then simulate assigning future capacities from period to period by setting each link’s capacity for the next period as the near-peak load in the current period multiplied by some overprovisioning factor (varying the factor on the X axis). We then compute the average link utilizations in the next period and the total number of links on which we see overloading ( $\geq 100\%$  utilization in at least one time interval).

Figure 2 plots the median utilization across links and periods and the number of congestive events as we vary the overprovisioning factor. Figure 2 shows that overprovisioning to accommodate peak loads introduces a tradeoff between inefficient utilization and overloading. Low overprovisioning factors between 10% and 30% lead to more efficient utilization (60%-70%) but lead to thousands of congestive events. High overprovisioning factors lead to far less overloading, but only roughly 50% utilization. Microsoft and Meta also stated that solely installing extra capacity without shifting traffic is not always a feasible solution [30, 86].

**Efficient, Proactive Planning is Possible:** Despite these limitations, Figure 1c shows that by advertising prefix 2 to provider 2 at site A a priori, the Service Provider can split traffic between the two links during failure avoiding overloading without (a) reacting to the failure and (b) upgrading deployment capacity.

## 2.4 Key Challenges

Since a Service Provider has lots of global capacity, dynamically placing traffic on paths to optimize performance objectives subject to capacity constraints would therefore be simple if all the paths to the Service Provider were always available to all users as in Figure 1d. However, making paths available to all users as in Figure 1d. However, making paths available uses IPv4 prefixes, which are monetarily expensive and pollute BGP routing tables. The solution in Figure 1d uses 50% more prefixes than the solution in Figure 1c and, at \$20k per prefix for 10,000 peerings would cost \$200M [75]. IPv6 is not a good alternative as IPv6 is not universally supported and IPv6 routing entries take 8 $\times$  the amount of memory to store in a router so would pollute global routing tables even more. Prior work noted the same but found that advertising around 50 prefixes was acceptable [104, 50], and most Service Providers advertise fewer than 50 today according to RIPE RIS BGP data [92]. We also verified with engineers at six large service providers that this was a valid limitation.

Since we cannot expose all the paths by advertising a unique prefix to each connected network, we must find some subset of paths to expose.

Finding that right subset of paths to expose that satisfies performance objectives, however, is hard since there are exponentially many subsets to consider and each subset needs to be tested (*i.e.*, advertised via BGP) to see how it performs. The number of subsets is on the order of  $2^{\text{number of peers/providers}}$ . As measuring this many advertisements is intractable, we have to predict how different subsets of paths perform which is challenging since inter-domain routing is difficult to model, and since there are too many possible changing conditions (failures, attacks) over which to assess these predictions.

## 3 Methodology

### 3.1 SCULPTOR Overview

SCULPTOR’s goal is to find an advertisement strategy that gives users good interdomain paths (relative to some objective function) and a traffic allocation to those paths. SCULPTOR should

also allocate good backup paths to handle non steady-state conditions. For objective functions we consider, such optimization problems are often large with more than 100M constraints and 2M decision variables (§3.2.2).

Minimizing an objective function requires evaluating it with several different inputs, but performing such measurements (*i.e.*, advertising prefixes) could take years at our problem size (§3.3.1), and so is not scalable. Instead, we estimate the objective by modeling paths probabilistically (§3.3.2), and update this model over time using a small number of measurements in the Internet (§3.3.3).

We then minimize the objective function using gradient descent which, at each iteration, requires solving millions of sub-problems (§3.4). To avoid expending the resources to solve all these sub-problems, we also implement efficient heuristics that find good approximate solutions (appendix A.4), although these heuristics are specific to the objective function we use in our evaluations. These heuristics speed computation at the cost of some convergence (§5.5), but are not necessary.

SCULPTOR’s power comes from its precise formulation of a global traffic engineering problem and the local approximations to this problem it makes to form a tractable solution. These approximations enable SCULPTOR to compare millions of possible advertisement strategies at milliseconds per computation, but only conduct tens in the actual Internet.

## 3.2 Problem Setup and Definitions

### 3.2.1 Goal and Setting

We aim to advertise prefixes to connected networks to optimize an objective function subject to constraints such as link capacity. Adding capacity constraints on sites (*e.g.*, servers) in addition to links would be a straightforward extension.

Service Providers connect to peers/providers at sites via physical connections we call peering links. Users route to the deployment through the public Internet to a prefix, over one of the peering links via which that prefix is advertised. The path to a prefix (and therefore peering link) is chosen via BGP. We fix the maximum number of allowable prefixes according to the Service Provider’s budget, and is generally much less than the number of peerings.

We consider users at the granularity of user groups (UGs), which generally refer to user networks that route to the Service Provider similarly, but could mean different things (*e.g.*, /24 IPv4 prefixes, metros). UGs generate steady-state traffic volumes,  $v_{UG}$ , and the Service Provider provisions capacity at links/sites to accommodate this load. A system run by the Service Provider measures latency from UGs to Service Provider peering links  $l$ , which is a reasonable assumption [9, 13, 24, 34, 104, 50].

We assume that the Service Provider has some technology for directing traffic towards prefixes. Examples include DNS [13, 8, 52], multipath transport [81, 23], or control-points

at/near user networks [84, 50]. DNS offers slow redirection due to caching [50] but is the most readily deployable by the largest number of Service Providers, whereas Service Provider-controlled appliances offer precise control but may not be a feasible option for some Service Providers. Service Providers with stronger incentives to provide the best service to users will invest in better options with more control, and eventually multipath transport will see wide enough deployment to be used by all Service Providers. Today, MPTCP is enabled by default in iOS [3] and Ubuntu 22 [31], and MPQUIC can be used in any application [106]. SCULPTOR assumes that a UG can be precisely directed to each prefix.

### 3.2.2 General Formulation

We cast the problem of finding advertisement configurations that admits good assignments of user traffic to paths as a multivariate optimization over both configurations and traffic assignments.

We represent an advertisement configuration  $A$  as a binary vector. Each entry indicates whether we advertise/do not advertise a particular prefix to a particular peer/provider (similar to prior work [104, 50]). Implementing this advertisement configuration (*i.e.*, advertising prefixes via BGP sessions) results in routes from users to the Service Provider. The assignment of traffic to resulting routes from this advertisement configuration is given by the nonnegative real-valued vector  $w$ , whose entries specify traffic allocation for each user along each route.

Although we do not know exactly which routes will result from a given configuration, assume that this mapping is well defined. Let  $R$  be the mapping from advertisement configurations to functions that map  $\langle \text{prefix}, \text{UG} \rangle$  pairs to links. For example, say  $R(A) = f$ , and  $f(p, \text{UG}) = l$  — this notation means that the output of an advertisement configuration  $A$  under routing  $R$  is a function  $f$  that maps prefixes  $p$  and UGs to a link,  $l$ . It could be that a configuration leads to no route for some UG to some prefix. We define a function  $e$  such that  $e(R(A)(p, \text{UG})) = 1$  when there is some route for UG to prefix  $p$  under configuration  $A$ , and 0 otherwise.

Now suppose the overall metric we want to minimize is  $G$  where  $G$  is a function of both configurations and traffic assignments. Examples include traffic cost, average latency, maximum latency, and their combinations (see appendix A.3 for a discussion of which metrics work better than others). The joint minimization over configurations and traffic assignments can then be expressed as the following.

$$\begin{aligned}
& \min_{A, w} G(R(A), w) \\
& \text{s.t. } w(p, \text{UG}) \geq 0 \\
& A(p, l) \in \{0, 1\} \quad \forall p, l \\
& \sum_p w(p, \text{UG}) e(R(A)(p, \text{UG})) = v(\text{UG}) \quad \forall \text{UG} \\
& \sum_{R(A)(p, \text{UG})=l} w(p, \text{UG}) \leq c(l) \quad \forall l
\end{aligned} \tag{1}$$

The first constraint requires that traffic assignments be non-negative and the second requires that configurations are binary. The third constraint requires that all user traffic  $v(\text{UG})$  is assigned, and none is assigned to nonexistent paths. The fourth constraint specifies that total traffic from all users towards a link,  $l$ , does not exceed the link's capacity,  $c(l)$ . Considering other capacity-related constraints (or at least the ones we could think of, like site capacity) is straightforward.

### 3.2.3 Specific Objectives

In our evaluations we focus on two specific objectives: (1) minimizing latency and maximum link utilization in unseen scenarios, and (2) optimally routing different traffic classes. One could choose any objective that can be met in intra-domain settings. With our interdomain path model (§2.2), path latency for a  $\text{UG}$  to a prefix is uniquely determined by  $\text{UG}$  and the corresponding peering link over which the  $\text{UG}$  ingresses. Let the latency for a user  $\text{UG}$  via a link  $l$  be  $\mathcal{L}(\text{UG}, l)$ .  $G$  is then given by the following.

$$\begin{aligned}
G(R(A), w) &= \sum_{p, \text{UG}} \mathcal{L}(\text{UG}, R(A)(p, \text{UG})) w(p, \text{UG}) + \beta M \\
&\text{s.t. } \frac{\sum_{R(A)(p, \text{UG})=l} w(p, \text{UG})}{c(l)} \leq M \quad \forall l
\end{aligned} \tag{2}$$

Constraining maximum link utilization,  $M$ , to be at least as much as the utilization of each link and then minimizing a sum including it forces  $M$  to be the maximum link utilization. The sum of average latency and maximum link utilization is weighted by a parameter,  $\beta$ , which represents a tradeoff between using uncongested links/sites and low propagation delay and is set by the Service Provider based on their goals. We first solve Equation (2) with  $M = 1$  to see if we can allocate traffic to paths with zero overloading. However such overload-free solutions may not exist for all  $A$ .

**Unseen Scenarios** To encourage good solutions to Equation (1) not only in steady-state but also in unseen conditions (failures, shifting traffic distributions), we add a regularizer to  $G$  given by  $\sum_l \alpha_l G(R(A * F_l), w)$ .  $F_l$  is the binary vector where  $F_{l,i} = 0$  if index  $i$  of  $A$  corresponds to advertising a prefix through link  $i$  and 1 otherwise. The optional weights  $\alpha_i$  allow operators to specify which failure events may be

more impactful or likely. In other words, multiplying by  $F_l$  simulates a failure, since it effectively withdraws all advertisements through link  $l$ , blackholing that traffic. This regularizer intuitively encourages advertisement solutions to provide  $\text{UGs}$  good primary and backup paths, thus preparing for dynamic conditions. In Appendix A.2 we discuss why this regularizer helps us find good global minima, despite Equation (1)'s non-convexity, by drawing analogies to the deep learning literature.

**Different Traffic Classes** We consider a separate problem where we try to route traffic with different performance requirements. One traffic class (latency-sensitive) should be routed with low latency, and the other (bulk) should not congest the latency-sensitive traffic. The objective function is a weighted combination of the average latency of latency-sensitive traffic and the amount of latency-sensitive traffic that is congested. We also constrain the maximum bulk overprovisioning on any link.

## 3.3 Predicting Interdomain Routes

Solving Equation (1) is challenging because we need to compute  $G(R(A), w)$  but computing  $R(A)$  requires advertising prefixes in the Internet which can only be done infrequently to avoid route-flap-dampening. In our optimization (detailed below) we compute  $R(A)$  for millions of different  $A$  which could take tens to hundreds of years at a rate of advertising one strategy per 20 minutes. Hence we model, instead of measure,  $\text{UG}$  paths and improve this model over time through relatively few measurements.

### 3.3.1 Initialization and Measuring $R(A)$

We initialize our strategy to be anycast on one prefix, and unicast on remaining prefixes (*i.e.*, one prefix per site). If we have more prefixes in our budget, we advertise those prefixes to randomly chosen peers/providers.

During optimization we periodically measure  $R(A)$  by advertising the corresponding prefixes via peerings as specified by  $A$  and measuring routes taken by  $\text{UGs}$  to each prefix. The time between successive advertisements is between 20 minutes and 2 hours to avoid route flap dampening.

### 3.3.2 Probabilistic $\text{UG}$ Paths

We model the routing function, and therefore our objective functions, probabilistically and update our probabilistic model over time as we measure how  $\text{UGs}$  route to the deployment, adapting methods from prior work [104, 88, 50]. When, during optimization, we require a value for  $G(R(A), w)$ , we compute its expected value given our current probabilistic model. We compute this expected value either approximately via Monte-Carlo methods, or exactly if  $G$ 's structure admits efficient computation (appendix A.4).





Figure 3: With no knowledge of routing preferences, we estimate latency from this UG to both the red and blue prefixes with the average over possible ingress latencies (3a). A priori, no prefix is advertised. We then advertise the red prefix and measure the path towards the red prefix (*e.g.*, using `traceroute`) and learn that the first ingress has higher preference than the third and fourth ingresses (3b). We use this information to refine our latency estimate towards the blue prefix (since the third ingress is no longer a possible option) without advertising the blue prefix, saving time.

Our probabilistic model assumes a priori, for a given UG towards a given prefix, that all ingress options that prefix is advertised to for a UG are equally likely. Upon learning that one ingress is preferred over the other, we exclude that less-preferred ingress as an option for that UG in all future calculations for all prefixes for which both ingresses are an option.

A useful property of many choices of objective function  $G$  is that we do not need to approximate the routing function perfectly to solve Equation (1). For example, most UGs have similar latency via their many paths to a single site, so when optimizing latency as in Equation (2) we effectively only need to predict the site the user ingresses at. As we exclude more options of where UGs could possibly ingress for each prefix, our objective function’s distribution on unmeasured scenarios converges to the true value.

An example of this process is shown in Figure 3, where we refine a latency estimate towards an unmeasured prefix (blue) using measurements towards other prefixes (red). Even though we only have 50% confidence in exactly which path the user takes (*i.e.*, in  $R(A)$ ), we have high confidence that the user’s latency towards the blue prefix will be about 22.5 ms.

### 3.3.3 Exploration to Refine $R(A)$

As, a priori, we do not know UG preferences, estimates of  $G$  could be very noisy until we learn preferences. To bootstrap our model, we periodically measure  $R(A)$  on “adjacent” advertisement configurations to our current scheme and look for strategies for which we are very uncertain about whether it will be better or worse. By adjacent schemes, we mean configurations that differ from  $A$  by one entry, or configurations representing failures ( $A \times F_l$ ). One could use various measures of uncertainty — we choose entropy. Measuring  $R(A)$  in the Internet takes time (§3.3.1), but we observed that it improves convergence.

## 3.4 A Two Pronged Approach

Even with our probabilistic model, we cannot solve Equation (1) directly/exhaustively since it is a mixed-integer program with millions of constraints which is too large for off-the-shelf solvers. Greedy [50] and random [104] solvers find good solutions for simple objectives such as steady-state latency, but do not generalize to other objectives with constraints (§5).

Instead, we split Equation (1) into an outer and inner optimization, solving for configurations,  $A$ , on the outer loop using gradient descent and traffic assignments,  $w$ , on the inner loop using a general-purpose solver. Intuitively, this approach works because it breaks up a challenging optimization into many sub-problems that can be solved in parallel, and is guaranteed to push us towards  $a$  (perhaps locally optimal) solution.

### 3.4.1 Outer Loop: Gradient Descent

To apply gradient descent to our integer configuration variable, we extend  $A$  to be a continuous variable with entries between 0 and 1 and threshold its entries at 0.5 to determine if a prefix is advertised/not advertised to a certain peer/provider. We approximate gradients between adjacent advertisements  $A_i$  and  $A_j$  by computing the expected value of  $G(R(A), w)$  at each advertisement and continuously interpolate  $G$  at intermediate configurations using sigmoids. We do not compute all entries of the gradient of Equation (1) since there are too many to compute. Instead, we use subsample entries of the gradient and track the largest entries. These continuous extension and dimension reduction methods are also used in other settings [89, 56].

**Scaling:** Minimization in the outer loop scales linearly with the number of UGs, quadratically with the number of peers/providers, and linearly with the number of prefixes. Despite this high complexity, in practice the implementation runs quickly (minutes per gradient step) relative to the rate at which we can advertise prefixes (once an hour). Gradient entry computations are parallelizable so this loop scales horizontally.

**Convergence:** Gradient descent converges to a local minima for bounded objectives [57]. Our evaluations show that `SCULPTOR` finds good solutions over a wide range of simulated topologies, and converges quickly with thousands of UGs and  $\langle \text{peering}, \text{prefix} \rangle$  pairs (§5). We comment on which problem properties will likely lead to better convergence in Appendix A.2 by drawing analogies to the deep learning literature, but find that allocating higher prefix budgets and adding richer advertisement capabilities (*e.g.*, BGP community tagging) can improve convergence, the latter of which is an area for future work.

### 3.4.2 Inner Loop: General Purpose

Each iteration of the outer loop requires solving for traffic allocations on advertisements corresponding to the gradient

entries that we wish to compute. We refer to the process of solving for these traffic allocations as an inner loop.

**Scaling:** In the case where we use Monte Carlo methods to approximate  $G(R(A), w)$ , we solve for allocations given several randomly generated  $R(A)$ . Hence the number of iterations in the inner loop scales with the product of the number of gradient computations and the number of Monte Carlo simulations. Scaling properties for solving each traffic allocation depends on the objective,  $G$ .

**Convergence** depends on the objective,  $G$ , and the solver. Many useful objectives are expressed as linear programs which are convex (e.g., our choices of  $G$ ), so solvers can find globally optimal solutions. Useful, non-convex objectives such as Cascara’s sometimes have approximately optimal solutions [90].

## 4 Implementation

We prototype SCULPTOR on the PEERING testbed [85], which is now available at 32 Vultr cloud sites. We describe how we built SCULPTOR on the real Internet and how we *emulate* a Service Provider including their clients, traffic volumes, and resource capacities. (We are not a Service Provider and so could not obtain actual volumes/capacities, but our extensive evaluations (§5) demonstrate SCULPTOR’s potential in an actual Service Provider and our open/reproducible methodology provides value to the community.)

### 4.1 Simulating Clients and Traffic Volumes

To simulate client performances, we measured actual latency from IP addresses to our PEERING prototype as in prior work [50], and selected targets according to assumptions about Vultr cloud’s client base.

We first tabulate a list of 5M IPv4 targets that respond to ping via exhaustively probing each /24. Vultr informs cloud customers of which prefixes are reachable via which peers, and we use this information to tabulate a list of peers and clients reachable through those peers.

After tabulating peers, we then measure latency from all clients to each peer individually by advertising a prefix solely to that peer using Vultr’s BGP action communities and ping-ing clients from Vultr. We also measure performance from all clients to all providers individually, as providers provide global reachability.

In our evaluations, we limit our focus to clients who had a route through at least one of Vultr’s direct peers (we exclude route server peers [12]). Vultr likely peers with networks with which it exchanges a significant amount of traffic [86], so clients with routes through those peers are more likely to be “important” to Vultr. We found 700k /24s with routes through 1086 of Vultr’s direct peers. In an effort to focus on interesting optimization cases, we removed clients whose lowest latency to Vultr was 1 ms or less, as these were assumed to be

addresses related to infrastructure, leaving us with measurements from 666k /24s to 825 Vultr ingresses.

As we do not have client traffic volume data, we simulate traffic volumes in an attempt to both balance load across the deployment but also encourage some diversity in which clients have the most traffic. To simulate client traffic volumes, we first randomly choose the total traffic volume of a site as a number between 1 and 10 and then divide that volume up randomly among clients that *anycast* routes to that site. Client volumes in a site are chosen to be within one order of magnitude of each other.

Although these traffic volumes are possibly not realistic, in demonstrating the efficacy of SCULPTOR over a wide range of subsets of sites and simulated client traffic volumes, we demonstrate that SCULPTOR’s benefits are not tied to any specific choice of sites or traffic pattern within those sites.

### 4.2 Deployments

We use a combination of real experiments and simulations to evaluate SCULPTOR. Both cases use simulated client traffic volumes, but our real experiments measure real paths using RIPE Atlas probes, while our simulations use simulated paths.

We implement SCULPTOR with Nesterov’s Accelerated Gradient Descent in Python [? ]. We set  $\alpha_l = 4.0$  and set the learning rate to 0.01 with decay over iterations. We solve Equation (2) with Gurobi.

**Experiments in the Internet** We assess how SCULPTOR performs on the Internet using RIPE Atlas probes [91], which represent a subset of all clients. RIPE Atlas allows us to measure paths (and thus ingress links) to prefixes we announce from PEERING, which SCULPTOR needs to refine its model (§3). However, RIPE Atlas does not have large coverage, as probes are only in 3,000 networks, and we are limited by RIPE Atlas probing rate constraints (15k traceroutes/day).

**Simulations** We also evaluate SCULPTOR by simulating user paths which allows us to conduct more extensive evaluations, as experiments take less time and use clients in more networks. To limit computational complexity, we select 100 clients with a route through each peer. Service Providers could similarly limit computational complexity by optimizing over, for example, a certain percent of the traffic which is a common practice in the networking optimization literature [8, 62, 50, 76]. (Service Providers could also scale compute.)

### 4.3 Setting Resource Capacities

We assume that resource capacities are overprovisioned proportional to their usual load. However, we do not know the usual load of Vultr links and cannot even determine which peering link that traffic to one of our prefixes arrives on, as Vultr does not give us this information. (This limitation only exists since we are not a Service Provider, as a Service Provider could measure this using IPFIX, for example.) We



overcome this limitation using two methods corresponding to our two deployments in Section 4.2, each with their pros and cons.

For our first method of inferring client ingress links, we advertise prefixes into the Internet using the PEERING testbed [85], and measure actual ingress links to those prefixes using traceroutes from RIPE Atlas probes [91]. Specifically, we perform IP to AS mappings and identify the previous AS in the path to Vultr. This approach has limited evaluation coverage, as RIPE Atlas probes are only in a few thousand networks. In cases where we cannot infer the ingress link even from a traceroute, we use the closest-matching latency from the traceroute to the clients’ (known) possible ingresses. For example, if an uninformative traceroute’s latency was 40 ms to Vultr’s Atlanta site and a client was known to have a 40 ms path through AS1299 at that site, we would say the ingress link was AS1299 at Atlanta.

The second method we use to infer ingress links is simulating user paths by assuming we know all user routing preference models (§3.2). We use a preference model where clients prefer peers over providers, and clients have a preferred provider. When choosing among multiple ingresses for the same peer/provider, clients prefer the lowest-latency option. We also add in random violations of the model. This second approach allows us to evaluate our model on all client networks but may not represent actual routing conditions. However, we found that our key evaluation results (§5) hold regardless of how we simulated routing conditions (we also tried completely random preference assignments), suggesting that our methodology is robust to such assumptions. Prior work also found the preference model to be valid in 90% of cases they studied [104].

Given either method of inferring client ingress links (RIPE Atlas/simulations), we then measure paths to an *anycast* prefix and assign resource capacities as some overprovisioned percentage of this catchment. (Discussions with operators from Service Providers suggested that they overprovision using this principle.) We report results for an overprovisioning rate of 30%, but find similar takeaways for 10% through 50%.

## 5 Evaluation

We evaluate SCULPTOR on two objective functions (§5.1). In optimizing the first objective (§5.2), SCULPTOR yields advertisement strategies that give clients lower latency paths and fewer congested links than other solutions during steady-state (§5.2), failure (§5.3), and conditions (§5.4). In optimizing the second objective, SCULPTOR successfully routes latency-sensitive traffic and excessive amounts of lower priority bulk traffic.

### 5.1 General Evaluation Setting

We compare SCULPTOR to other solutions.

**anycast:** A single prefix announcement to all peers/providers at all sites, which is a common strategy used by Service Providers today [33, 17, 8, 97, 79, 102, 107].

**unicast:** A single prefix announcement to all peers/providers at each site (one per site). Another common strategy used in today’s deployments [52, 13]. Unicast also provides an upper bound on FastRoute’s performance.

**AnyOpt/ PAINTER:** Two proposed strategies for reducing steady-state latency compared to *anycast* [104, 50].

**One-per-Peering:** A unique prefix advertisement to each peer/provider, so many possible paths are always available from users. This solution serves as our performance upper-bound, even though it is prohibitively expensive. (We do not know an optimal solution with fewer prefixes.)

For the public Internet implementation (§4.2), we limit the scale of our deployment to 10 large sites to avoid reaching RIPE Atlas daily probing limits. These 10 sites were Miami, New York, Chicago, Dallas, Atlanta, Paris, London, Stockholm, Sao Paulo, and Madrid. We do not compute the solution for AnyOpt for our evaluations on the Internet since AnyOpt did not perform well compared to any other solution in our simulations, and since computing AnyOpt takes a long time. We use 12 prefixes, which is low, since we do not have many prefixes. Choosing RIPE Atlas probes to maximize network coverage and geographic diversity, we select probes from 972 networks in 38 countries which have paths to 484 ingresses.

For our simulated evaluations (§4.2), we compute solutions over many random routing preferences, demands, and subsets of sites to demonstrate that SCULPTOR’s benefits are not limited to a specific deployment property. We evaluate SCULPTOR over deployments of size 3, 5, 10, 15, 20, 25, and 32 sites. Each size deployment was run at least 10 times with random subsets of UGs, UG demands, and routing preferences. We use one tenth of the number of peers/providers as the number of prefixes in our budget for all solutions (except **One-per-Peering**). For larger deployments (20-32 sites) we use approximately 60 prefixes, and for smaller ones (3-15 sites) we use between 10 and 30. Prior work found that using this many prefixes to improve performance was a reasonable cost [104, 50]. Over all studied deployments, we consider paths from clients in 42k /24s to 868 peers/providers.

In solving for traffic allocations during failure scenarios, links may be overloaded. We say all traffic arriving on a congested link is congested and do not include this traffic in latency comparisons (congested traffic latency would be a complicated function of congestion control protocols and queueing behavior). We comment separately on how much traffic is congested.

We compute both average overall latency and the fraction of traffic within 10 ms (very little routing inefficiency), 50 ms (some routing inefficiency), and 100 ms (lots of routing inefficiency) of the *One-per-Peering* solution for each advertisement strategy.

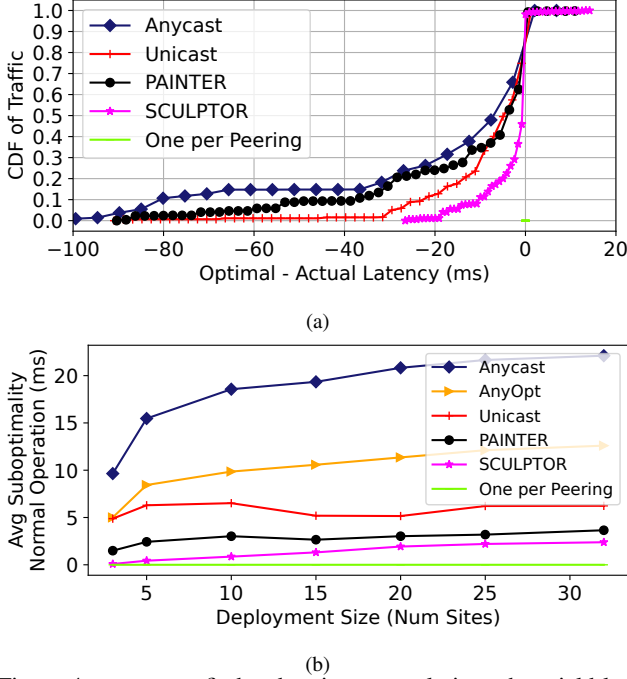


Figure 4: SCULPTOR finds advertisement solutions that yield low steady-state latency subject to capacity constraints both on the Internet (4a) and in simulation (4b).

## 5.2 Handling Unseen Conditions

### 5.2.1 Lower Latency in Steady-State

**Internet Deployment** We compute the latency that each advertisement solution achieves compared to the One-per-Peering solution for each UG. Figure 4a shows a CDF of these differences weighted by traffic, demonstrating that SCULPTOR outperforms other solutions with realistic routing. SCULPTOR is only 3.0 ms worse than (the unreasonably expensive) One-per-Peering solution whereas the next best solution, unicast, is 8.4 ms worse. SCULPTOR also serves 88.8% of traffic within 10 ms of One-per-Peering, whereas the same is true for only 69.5% of traffic with a unicast solution.

PAINTER does not do as well as unicast here (but does in simulation), which may be because of our relatively low budget of 12 prefixes. We use more prefixes in our simulations, and prior work noted that the most significant gains when using roughly 40 prefixes [50].

**Simulations** Figure 4b shows the average latency compared to the One-per-Peering solution over all simulated deployments at each deployment size. Average latency for SCULPTOR ranges from 0.1 to 2.5 ms worse than (the undeployable) One-per-Peering. The next-best solution (PAINTER) is on average 1.2 ms and 2.4 ms worse than SCULPTOR.

anycast comparatively performs the worst with UGs being on average 17.9 ms worse than One-per-Peering whereas SCULPTOR performs the best with UGs being 1.3 ms

worse than One-per-Peering. Interestingly, unicast (6.1 ms worse than One-per-Peering) actually performs better than AnyOpt (10.1 ms worse than One-per-Peering) which could be due to the different setting AnyOpt was designed for (AnyOpt was designed to optimize latency without capacity constraints over a small number of provider connections which does not capture the realities of many Service Providers).

These average latency improvements translate into quantifiable routing inefficiency for different fractions of traffic (we include full results in Appendix B.1). SCULPTOR has on average 94.8% traffic within 10 ms of the One-per-Peering solution, 99.2% within 50 ms, and 99.9% within 100 ms. These percentages compare to the next-best solution, PAINTER, which has on average 91.7% traffic within 10 ms of the One-per-Peering solution, 97.4% within 50 ms, and 99.2% within 100 ms.

### 5.2.2 Better Resilience to Failure

We next assess SCULPTOR’s ability to provide resilience to ingress failures and site failures. Examples of such failures include excessive DDoS traffic on the link/site (thus using the link/site as a sink for the bogus traffic), physical failure, draining sites, and/or planned maintenance. Figure 5 demonstrates that SCULPTOR shifts traffic without overloading alternate links/sites more effectively than any other solution *without BGP announcements/withdrawals* which could cause further failure (§2.3).

In these evaluations, we fail each ingress/site once and compute traffic allocations. For each advertisement strategy and failed component, we compute the difference between achieved latency and One-per-Peering latency for UGs who use that component in steady-state. For example, if the Tokyo site fails, we report on the post-failure latency of UGs served from Tokyo before the failure and not of other UGs.

We separately note the fraction of traffic that lands on congested links for each solution. We do not include congested traffic in average latency computations but say such traffic does *not* satisfy 10 ms, 50 ms, or 100 ms objectives.

**Internet Deployment** Figure 5a demonstrates that SCULPTOR offers lower latency paths for more UGs during single link failure in realistic routing conditions. On average, SCULPTOR is only 7.2 ms higher latency than One-per-Peering while the next best solution, unicast, is 12.6 ms worse than One-per-Peering. PAINTER struggles to find sufficient capacity for UGs, with 40.8% overloading.

Single site failures (shown in Appendix B.2) show similar results. On average, SCULPTOR is 18.8 ms higher latency than One-per-Peering while the next best solution, unicast, has 16% overloading and PAINTER has 86% overloading during site failure.



Figure 5: SCULPTOR improves resilience to failure both on the Internet (5a) and in simulation (5b, 5c).

**Simulations** We show the fraction of traffic within 50 ms of the One-per-Peering solution for link and site failures in Figure 5b and Figure 5c (further results are in Appendix B.2).

For single-link failures, anycast comparatively performs the worst with UGs being 55.4 ms worse than One-per-Peering on average, whereas SCULPTOR ranges from 1.4 ms and 9.8 ms worse than One-per-Peering on average. SCULPTOR also avoids more overloading, with only 0.6% of traffic being congested on average while PAINTER (the next-best solution) leads to 2.3% of traffic being congested on average. Single-site failure exhibits similar trends where SCULPTOR is 10.5 ms worse than One-per-Peering and has 22.9% traffic overloading, on average, while PAINTER leads to 19.5 ms worse than One-per-Peering and 39.9% overloading on average.

When looking at routing inefficiency, SCULPTOR has 78.2% of traffic within 10 ms of the One-per-Peering solution on average, 93.0% within 50 ms, and 97.2% within 100 ms during link failure. These statistics compare to the next-best solution, PAINTER, which has 66.9% within 10 ms, 83.4% within 50 ms, and 90.4% within 100 ms. Site failures show similar trends (full results are in Appendix B.2).

Differences quoted above may seem small (for example, 11.3% in the amount of traffic within 10 ms of One-per-Peering between SCULPTOR and PAINTER), but represent potential fractions of *trillions* of requests per day made across the whole world [86]. Large Service Providers recently emphasized that these seemingly small gains make large differences [29, 100].

### 5.2.3 Efficient Infrastructure Utilization

Figure 6 shows that installing more capacity to handle peak loads is not always necessary with better routing — SCULPTOR finds ways to distribute load over existing infrastructure to accommodate increased demand without adversely affecting latency. We quantify this improved infrastructure utilization under two realistic traffic patterns: flash crowds and diurnal effects.

**Methodology** We define a flash crowd as a transient traffic increase at a single site. Examples of flash crowds include content releases (games, software updates) that spur downloads in a particular region. Since increased demand is highly

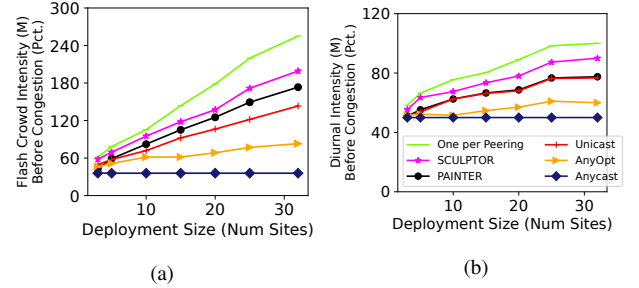


Figure 6: SCULPTOR improves infrastructure utilization under flash crowds and diurnal traffic patterns so that Service Providers can underprovision compared to peak loads.

localized, we can spread excess demand to other sites if paths to those sites exist.

To generate Figure 6, we identify each client with a single “region” corresponding to the site at which they have the lowest possible latency ingress link. For each region individually, we then scale each client’s traffic in that region by  $M\%$  and compute traffic to path allocations. If there are  $S$  sites in the deployment, we thus compute  $S$  separate allocations per  $M$  value where each allocation assumes inflated traffic in exactly one region. We increase  $M$  until any link experiences overloading when we (separately) scale volume in any region. For example, if Atlanta experiences overloading with a 60% increase in traffic for Atlanta clients, but no other region does, we call  $M = 60\%$  the critical value of  $M$ .

Our diurnal analysis in Figure 6b uses a similar methodology. We define a diurnal effect as a traffic pattern that changes volume according to the time of day. Diurnal effects might be different for different Service Providers, but a prior study from a large Service Provider demonstrates that these effects can cause large differences between peak and mean site volume [16]. We sample diurnal patterns from that publication and apply them to our own traffic. We group sites in the same time zone and assign traffic in different time zones different multipliers — in “peak” time zones we assign a multiplier of  $M$  and in “trough” time zones a multiplier of  $0.1M$ . The full curve is shown in Appendix B.3. Similar to our flash crowd analysis, we increase  $M$  until at least one link experiences overloading at least one hour of the day.

**Results** Figure 6 plots the average over simulations of critical  $M$  values that cause overloading for each deployment size under flash crowds and diurnal effects computing using