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

Paper #950, 12 pages body, 21 pages total

ABSTRACT

Large cloud/content (service) providers 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 DDoS attacks, Service Providers overprovision to accommodate peak loads and activate emergency systems for shifting excess traffic. We take a different approach with SCULPTOR, a framework that proactively optimizes BGP configurations to satisfy diverse ingress routing objectives-including latency, cost, and capacity constraints-simultaneously. SCULPTOR models Internet routing to approximately 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 route high-priority traffic with a 50% reduction in overloading.

1 INTRODUCTION

Cloud/content providers (hereafter Service Providers) enable diverse Internet applications used daily by billions of users. Traditionally, Service Providers used DNS and static BGP advertisements to define a single path from a user to a service, leaving the specific path up to the Internet to determine.

Increasingly, however, these services have diverse requirements that can be challenging to meet with a single path that Service Providers have little control over. For example, enterprise services have tight reliability requirements [**?** **?**], and new applications such as virtual reality require $\leq$ 10 ms round trip latency [**?**] and $\leq$ 3 ms jitter [**?**]. Complicating matters, Service Providers must meet these requirements subject to changing conditions such as peering link/site failures [**?** **?**] and DDoS attacks [**?** **?** **?**]. Critically, such changes can cause *overload* if the Service Provider cannot handle new traffic volumes induced by the change. This overload can lead to degraded service for users, hurting reliability [**?** **?** **?** **?**], and may require manual intervention.

To try to meet different goals and respond to changing conditions, Service Providers adopt solutions that are either (a) too costly, (b) too reactive, or (c) too specific. For example, Service Providers proactively overprovision resources [**?** **?** **?**], but we demonstrate using traces that bursty traffic patterns can require costly overprovisioning rates as high as 70% despite sufficient global capacity (§2.3). TIPSY responds to changing loads to retain reliability [**?**], but only does so reactively, which could lead to short-term degraded performance. AnyOpt and PAINTER proactively set up routes to minimize specific objectives such as steady-state latency [**?** **?**], but it is challenging to extend those approaches to other objectives (§5.4). 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).

We present Service Providers with a flexible framework, SCULPTOR (**S**couring **C**onfigurations for **U**tilization-**L**oss-and **P**erformance-aware **T**raffic **O**ptimization & **R**outing), which accepts as input cost, performance, and reliability objectives and outputs BGP advertisements and traffic allocations that help achieve those desired objectives. SCULPTOR is the first system that finds good solutions for ingress interdomain routing objectives such as maximum link utilization, transit cost, and latency for *interdomain* traffic (existing systems work for intradomain traffic, *e.g.,* [**?** **?** **?** **?**]). SCULPTOR computes BGP advertisements proactively, and controls which subset of addresses handles active traffic. SCULPTOR only places live traffic on routes after convergence, and hence does not run the risk of outages (§4).

To solve each traffic engineering problem, SCULPTOR efficiently searches over the large BGP advertisement search space ($> 2^{10,000}$ possibilities, *i.e.,* from > 10,000 peerings) 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 on the Internet is slow (§3.3). SCULPTOR then finds configurations using gradient descent, which is appropriate in our setting due to the horizontal scaling that gradient descent admits (§3.4). This modeling enables SCULPTOR to assess > 1M configurations per measurement on the Internet (§5.4), 1,000× more than other solutions. Like other work that uses gradient descent with success (*e.g.,* deep learning), we sacrifice the ability to provide a formal characterization of which objective functions are possible for an approach that lets us approximately optimize multiple criteria (§5)

We prototype and evaluate our framework at Internet scale using the PEERING testbed [**?**] (§4), which is now deployed at 32 Vultr cloud locations [**?**]. Vultr is a global public cloud that allows our prototype to issue BGP advertisements via more than 10,000 peerings. To demonstrate SCULPTOR's utility across different objectives, we evaluate our framework on three specific objectives (computed separately): (a) minimizing latency under unseen traffic conditions, (b) routing latency-sensitive and bulk traffic classes, and (c) directing

traffic to lower-cost sites. We compare SCULPTOR's performance on these problems to that of an unreasonably expensive "optimal" solution (computing the actual optimal is infeasible). We make an anonymized implementation of SCULPTOR available at https://anonymous.4open.science/r/sculptor_stripped-FE7C and will release all code upon acceptance.

For the first objective, we found that, compared to other approaches, SCULPTOR increases the amount of traffic within 10 ms of the optimal by 19.3% in steady-state (meeting that target for 95% of traffic) (§5.2.1), by 11% during link failure, and by 17% during site failure. SCULPTOR also reduces overloading on links during site failures by up to 41% over PAINTER, giving Service Providers more confidence that services will still be available during partial failure (§5.2.1). We also find that, by load balancing traffic on backup paths during peak times, we can satisfy high peak demands with the same infrastructure. SCULPTOR can handle flash crowds (*e.g.,* DDoS attacks) at more than 3× expected traffic volume, reducing the amount of overprovisioning that Service Providers need, thus reducing costs.

For the second objective, SCULPTOR routes bulk low-priority traffic in ways that avoid congesting high-priority traffic, achieving near-optimal latency and reducing congestion by up to 2× (§5.2.2).

Finally, for the third objective, SCULPTOR routes traffic with lower latency and to lower-cost sites than all other solutions which has implications for power savings, carbon savings, and resilience (§5.3).

After decades of intradomain traffic engineering using programmable networking primitives such as virtual output queueing for differentiated service [**?**] (and others, §7), SCULPTOR brings such control closer to realization in the interdomain setting with a unilaterally deployable framework. Service Providers can use flexible frameworks such as SCULPTOR to help bring us the resilient, performant service that our diverse applications increasingly need.

Ethics: This work raises no ethical issues.

2 MOTIVATION

2.1 Variable, Evolving Goals

Evolving Internet use cases are pushing Service Providers to meet diverse requirements. For example, Service Providers increasingly host mission-critical services such as enterprise solutions [**?**], which are predicted to be a $60B industry by 2027 [**?**]. These enterprise solutions require high reliability, whereas low latency is a more important metric for gaming [**?**], which generates more revenue than the music and movie industries combined [**?**]. Service Providers are also expanding the set of services they provide—for example, CDNs that traditionally hosted static content are pivoting to offering services like compute [**? ? ?**].

Moreover, as use cases evolve, Service Providers increasingly need to meet performance requirements for *ingress* traffic since that traffic includes, for example, player movements in real-time games, voice and video in enterprise conferences, and video/image uploads for AI processing in the cloud [**? ?**]. This reality is a departure from traditional CDN traffic patterns in which ingress traffic was primarily small requests and TCP acknowledgments.

Despite efforts to meet variable service requirements [**? ? ? ? ? ? ?**], doing so is still challenging due to issues *outside* Service Provider control.

First, Service Providers lack control over which ingress path traffic takes since BGP, the Internet's interdomain routing protocol, computes paths in a distributed fashion, giving each intermediate network (notably the source network) a say in the ingress path. BGP chooses a single path which may not be the best one for meeting a given requirement. Service Providers cannot even unilaterally control the part of the path closest to them—upgrading peering capacity requires coordination among multiple parties [**?**].

Second, dynamic factors outside Service Provider control make satisfying requirements even harder. Peering disputes can lead to congestion on interdomain links and inflated paths [**? ? ?**], and DDoS attacks still bring down sites/services [**? ? ? ? ?**], despite considerable effort in mitigating attacks [**? ?**]. Moreover, recent work [**? ? ?**] and blog posts [**? ? ? ? ? ? ?**] show that user traffic demands are highly variable due to flash crowds and path changes and so can be hard to plan for.

2.2 Existing Techniques and Limitations

Today, most Service Providers either use anycast prefix advertisements to provide low latency and high availability at the expense of some control [**? ? ? ? ?**], or unicast prefix advertisements to direct users to specific sites [**? ? ? ? ?**]. Anycast, where Service Providers advertise a single prefix to all peers/providers at all/many sites, offers some natural availability following failures, since BGP automatically reroutes most traffic to avoid the failure after tens of seconds [**?**]. Prior work shows that this availability comes with higher latency in some cases [**? ?**]. Unicast gives clients lower latency than anycast by advertising a unique prefix at each site [**? ?**], but can suffer from reliability concerns, taking as much as an hour to shift traffic away from bad routes due to stale DNS records [**?**].

2.3 Limitations of ITE Systems

In response to some of the basic limitations of anycast and unicast, others have proposed specialized ITE systems. However, these solutions come with the following three key limitations.

Too Specific: Some have noted that, by intelligently advertising different prefixes to subsets of peers/providers to offer users more paths, Service Providers can achieve better performance [**? ? ? ? ? ?**]. However, these solutions rely

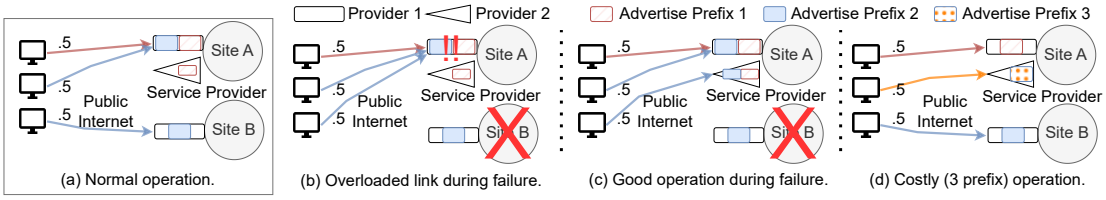


Figure 1: In normal operation traffic is split between two sites by directing half the traffic to each prefix using DNS (a). Even though there is enough global capacity to serve all traffic when site B fails, there is no way to split traffic across multiple providers given available paths, leading to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

on objective-specific solution methods that do not easily generalize to multi-constraint optimization problems (§5.4).

Figure 1 shows how this specific focus on a single objective could lead to reliability issues, *e.g.,* during a site 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 state-of-the-art systems that provide low (steady-state) latency from users to Service Provider networks such as `AnyOpt` [**?**] and `PAINTER` [**?**].

Too Reactive: To enhance reliability, some Service Providers propose `ITE` systems that change BGP advertisements when conditions change—for example, `TIPSY` drains traffic from overloaded links/sites by withdrawing announcements [**?**]. However, changing the set of paths is known to lead to outages [**? ? ? ?**], and can take time to converge. We would ideally only change announcements carrying live traffic as a last resort.

Too Costly: As sudden changes in traffic can cause congestion and thus disrupt services, Service Providers often overprovision resources to handle transient peak loads [**? ? ?**]. However, this approach can incur excessive costs, as Figure 2 demonstrates using link utilization data (per 5 minutes) from OVH cloud [**?**].

To generate Figure 2 we first split the dataset into successive, non-overlapping 120-day planning periods and compute the 95^{th} percentile ingress link utilization ("near-peak load") for each link/period. We then assign each link's capacity for the next period as the near-peak load in the current period multiplied by an overprovisioning factor (X-axis variable). We then compute the average link utilization in the next period and the total number of links on which we see $\geq 100\%$ utilization in at least one 5-minute 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,

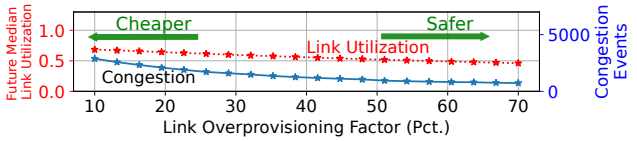


Figure 2: Planning for peak loads to avoid overloading requires inefficient overprovisioning.

but only 50% utilization. The congestive events span 41 OVH sites and 1,650 peering links, including provider and cloud links, indicating that the challenge is not isolated and simple to address. Considering larger planning periods did not significantly change results, and considering smaller ones led to more congestion at higher overprovisioning factors.

An important note on cost: We model deployment cost as correlated with the peering capacity over all links/sites, even though the actual relationship may be complex and additional peering capacity itself is not prohibitively expensive. Upgrading capacity has fixed costs (*e.g.,* fiber, router backplane bandwidth, line card upgrades, CDN servers near peering routers) and variable costs (*e.g.,* power, transit), both of which scale with demand, and Service Providers strive to keep them low [**? ? ? ?**]. Hence the high "Link Overprovisioning Factor" to ensure low congestion should be extrapolated out to the general "cost" of upgrading Service Provider serving infrastructure, rather than just a peering link.

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 satisfy 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 uses IPv4 prefixes, which are expensive and contribute to the global routing table's memory footprint. 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 [?]. IPv6 is not a good alternative as it is not supported by 60% of users according to APNIC [?]. Moreover, adoption is slowing, with adoption in the US stagnating at 50% for a few years, and full adoption possibly taking until 2045 according to APNIC [?] and Google [?]. Prior work noted the same but found that advertising around 50 prefixes was acceptable [? ?], and most Service Providers advertise fewer than 50 today according to RIPE RIS BGP data [?]. We also verified with engineers at six Service Providers that this challenge is important to address.

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 currently needs to be tested (*i.e.,* advertised via BGP) to see how it performs. The number of subsets is on the order of 2 to the number of ingresses Service Providers have, which, for some Service Providers (including Vultr, which we measure) [?], is $> 2^{10,000}$. As measuring this many advertisements is intractable, we have to predict how different subsets of paths perform, which is challenging since interdomain routing is difficult to model [? ?].

3 METHODOLOGY

3.1 SCULPTOR Overview

SCULPTOR tries to find an advertisement strategy that gives users good interdomain paths (relative to general objectives) and a traffic allocation to those paths. SCULPTOR computes BGP advertisements proactively, only placing live traffic on them after computation (§4). For objectives we consider, resulting optimization problems are large with more than 100M constraints and 2M decision variables (§3.2.2). Our framework breaks this challenging problem into manageable components. Figure 3 shows interactions among these components.

Minimizing an objective function requires evaluating it with many different inputs, but performing these assessments by advertising prefixes and measuring the results could take years at our problem size (§3.3.1) and so is not scalable. Instead, we estimate the objective using a "Probabilistic Routing Model" (§3.3.2) and update this model over time by "Advertising & Measuring" a small number of advertisements in the Internet using entropy-based "Exploration" (§3.3.3). The waiting time between advertisements is SCULPTOR's key bottleneck (§3.3.1), so advertising as little as possible is important.

We then minimize the objective function using gradient descent ("∇G") which, at each iteration, requires solving millions of sub-problems for traffic allocations ("Inner Loop Workers", §3.4.1). These sub-problems can either be solved exactly with a "General Purpose Solver" (§3.4.2) or approximately with efficient heuristics (Appx. C).

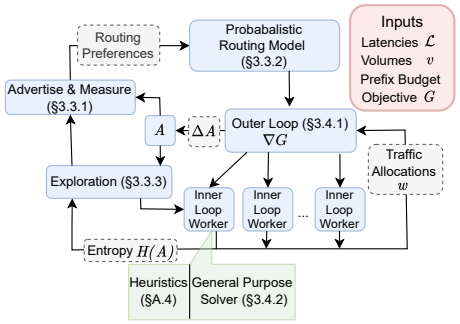


Figure 3: SCULPTOR advertisement computation overview.

3.2 Problem Setup and Definitions

3.2.1 Setting

Service Providers offer their services from tens to hundreds of geo-distributed sites [?]. 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 that have an aggregate capacity. Service Providers also connect to other networks at sites via dedicated links or shared IXP fabrics which we call peering links. 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 high utilization [? ?]. Resources can also fail due to, for example, physical failure and misconfiguration.

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, which is generally much less than the number of peerings.

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. This model is different from considering paths as equivalent to inter-AS adjacencies since many ASes peer with Service Providers at multiple locations.

We assume the peering link is the bottleneck of each path, for two reasons. First, this link increasingly represents the most important part of the path for Service Providers due to Internet flattening [?]. Second, other parts of the path are less interesting for traffic engineering—last-mile bottlenecks are common to all paths for a user, and existing systems perform traffic engineering in intermediate networks. Handling unexpected bottlenecks (*i.e.,* in a peering network) can also be viewed as a path failure, which we evaluate in Section 3.2.3, *demonstrating that* SCULPTOR *handles violations of this assumption.*

We consider users at the granularity of user groups (UGs), where a UG refers to user networks that route to the Service Provider similarly, but could mean different things in different instantiations of our system (*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 and traffic volumes from UGs to Service Provider peering links l, which is a reasonable assumption [? ? ? ? ? ?]. SCULPTOR is resilient to changes in latency (§5.2.1) and traffic (§5.2.1) over time.

We assume SCULPTOR has a technology for directing traffic towards prefixes. Examples include DNS [? ? ? ?], multipath transport [? ? ?], or control-points at/near user networks [? ?]. DNS offers slow redirection due to caching [?] but is the most deployable, whereas Service Provider-controlled appliances offer precise control but may be hard to deploy. Multipath transport will eventually see wide enough deployment to be used by all Service Providers. Today, MPTCP is installed in iOS [?] and Ubuntu 22 [?]; all applications can use MPQUIC [?]. As noted in PAINTER [?], Service Providers with more incentives can invest in more effective direction technology (*e.g.,* developing MPQUIC tooling). Moreover, not every objective function needs fine-grained redirection.

3.2.2 General Formulation

Finding advertisement configurations that admit good assignments of user traffic to paths is a multivariate optimization problem over both configurations and traffic assignments. Appendix A tabulates variables and expressions.

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 at a particular site (similar to prior work [? ?]). Implementing this configuration (*i.e.,* advertising prefixes via BGP sessions) results in routes from UGs to the Service Provider. UGs may have their traffic fractionally allocated to different prefixes (*i.e.,* routes). The assignment of traffic to resulting routes from this configuration is given by the nonnegative real-valued vector w, whose entries specify traffic allocation for each UG along each route.

Announcing a configuration will result in some set of routes (although knowing exactly which routes a priori is challenging), and these routes define which ingress link a UG uses for a prefix. We can think of this process as a routing function R that takes an advertisement configuration A and outputs a map from $\langle \texttt{prefix, UG} \rangle$ pairs to ingress links. For example, say $R(A) = f_A$, and $f_A(p, \mathsf{UG}) = l$ — this notation means that the output of an advertisement configuration A under routing R is a function f_A that tells us that users UG reach prefix p via 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 $e(R(A)(p, \mathsf{UG})) = 1$ 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 which is a function of both configurations and traffic assignments. Examples include traffic cost, average latency, maximum latency, and their combinations (§3.4). The joint minimization over configurations and traffic assignments can then be expressed as the following.

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

The first constraint requires that traffic assignments be nonnegative and that configurations are binary. The second constraint requires that all user traffic $v(\mathsf{UG})$ is assigned, and none is assigned to nonexistent paths. Capacity constraints depend on the choice of G.

3.2.3 Specific Objectives

In our evaluations we focus on some specific objectives that are commonly targeted in other settings: (1) minimizing latency and maximum link utilization in unseen scenarios, (2) performing traffic engineering over different traffic classes, and (3) minimizing traffic cost over sites. The framework likely accommodates objectives that have been met in other networking settings [? ? ? ? ? ? ? ? ? ?] but, like other work that has used gradient descent with success, we sacrifice the ability to provide a formal characterization of which objective functions are possible for an approach that lets us find useful solutions.

With our interdomain path model (§3.2.1), steady-state path latency for a UG to a prefix is uniquely determined by UG and the corresponding peering link over which the UG ingresses (but latency may change, which we evaluate in Section 5.2.1. Let the latency for a user UG via a link l be $\mathcal{L}(\mathsf{UG}, l)$. G is then given by the following.

$$\begin{aligned} G(R(A), w) = \frac{1}{\sum_{\mathsf{UG}} v(\mathsf{UG})} \sum_{p,\mathsf{UG}} \mathcal{L}(\mathsf{UG}, R(A)(p, \mathsf{UG})) w(p, \mathsf{UG}) \;+\; \beta M \\ \text{s.t.} \quad \frac{\sum_{R(A)(p,\mathsf{UG})=l} w(p, \mathsf{UG})}{c(l)} \leq M \quad \forall\, l \end{aligned} \tag{2}$$

The capacity for link l is $c(l)$. 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, β, 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, which may not be possible for all A.

Unseen Scenario Regularizer To encourage good solutions to Equation (1) not only in steady-state but also in unseen conditions (failures, shifting traffic distributions), we add a term to G given by $\sum_l \alpha_l G(R(A * F_l), w)$. The vectors F_l are binary vectors defined so that multiplying advertisements, A, by these vectors simulates withdrawal/failure on a link/site (*i.e.,* zeroing out the corresponding components).

This term (*i.e.,* regularizer) intuitively encourages advertisement solutions to provide UGs good primary and backup paths, thus preparing for dynamic conditions.

Different Traffic Classes To demonstrate that SCULPTOR works with other useful objectives, we consider a problem where we try to route traffic with different performance requirements (*i.e.,* classes). As in the private WAN setting where Service Providers already perform multi-class traffic engineering [**? ?**], we consider a multi-traffic class problem. We consider a scenario where one traffic class (high-priority) should be routed with low latency, and the other (low-priority) should not congest the high-priority traffic. An additional challenge in the interdomain setting is that we cannot use priority queueing to ensure that high-priority traffic is not congested since we do not control queue behavior on the path. The objective is a weighted combination of the average latency of high-priority traffic and the amount of high-priority traffic that is congested. We also constrain the maximum low-priority overprovisioning on any link so that not all low-priority traffic lands on one link, as this solution would lead to poor low-priority goodput.

Minimizing Traffic Cost Over Sites We also consider the scenario where each site has a cost proportional to the traffic volume arriving at the site. This objective models scenarios where Service Providers want to minimize power/carbon/peering infrastructure cost (which can vary by region) or minimize the use of unreliable sites (*e.g.,* dirty power). We minimize the weighted sum of average latency and average site cost.

3.3 Predicting Interdomain Routes

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

3.3.1 Initializing A 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 randomly do/do not advertise those prefixes via random ingresses. We do not use a completely random initialization since anycast and unicast have their benefits (§2.2).

During optimization SCULPTOR measures $R(A)$ by occasionally advertising the corresponding prefixes via peerings as specified by A and measuring routes taken by UGs to each prefix. SCULPTOR alternates between measuring advertisements it thinks are good (§3.4) and ones it thinks offer useful information (§3.3.3). SCULPTOR waits between advertising configurations to avoid route-flap-dampening which is the key optimization bottleneck. Hence the convergence time is essentially the number of measurements multiplied by the waiting period (for us set to one hour).

3.3.2 Probabilistic UG Paths

We model the routing function, and therefore our objective functions, probabilistically and update our probabilistic model over time as we measure how UGs route to the deployment. 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 (Appx. C).

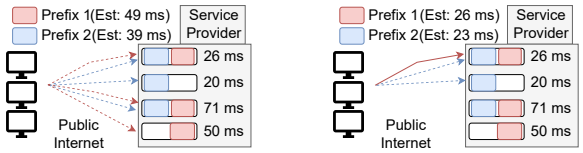
Our probabilistic model assumes a priori, for a given UG towards a given prefix, that all ingress link options that prefix is advertised to that are reachable from a UG are equally likely, assuming the UG has *any* route through that ingress. Providers can discern the set of possible ingresses for a UG with high reliability based on which ingresses the UG is reachable from. One could incorporate prior information about ingress preferences (from economic/routing models, for example), but such complexity is unnecessary for good performance. We instead learn preferences over time.

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. Prior work used a similar routing model [**? ? ?**] but did not extend it to deal with general objectives nor did it treat routing probabilistically.

A useful property of many choices of G is that we do not need to perfectly predict all the routes to solve Equation (1). For example, most UGs have similar latency via their many paths to a single site [**?**], so when minimizing latency as in Equation (2) we effectively need to predict the site the user ingresses at. As we exclude more options that 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 4, where SCULPTOR refines a latency estimate towards an unmeasured prefix (blue) using measurements towards other prefixes (red). Even though SCULPTOR has 50% confidence in which path the user takes (*i.e.,* in $R(A)$), SCULPTOR knows that the user's latency towards the blue prefix will be about 23 ms (*i.e.,* either 26 ms or 20 ms) and so SCULPTOR obtains a better estimate of the objective function.

3.3.3 Exploration to Refine $R(A)$

Probabilistic estimates of G could be inaccurate until we learn UG preferences. To refine its model, SCULPTOR periodically measures $R(A)$ on an adjacent configuration to the configuration at the current optimization step (§3.4). By adjacent configurations, we mean ones that differ from the current configuration, A, by one entry, or configurations representing a single failure ($A \times F_l$). SCULPTOR measures adjacent



(a) (b)

Figure 4: SCULPTOR initially estimates latency from this UG to both the red and blue prefixes with the average over possible ingress latencies (4a). A priori, no prefix is advertised. SCULPTOR then advertises the red prefix and learns that the first ingress has higher preference than the third and fourth (4b). SCULPTOR uses this information to refine its latency estimate towards the blue prefix (since the third ingress is no longer a possible option) without advertising the blue prefix, saving time.

configurations since it uses a descent-based optimization (§3.4) which benefits from knowing whether nearby strategies are better. Of the adjacent options, we select the one with the most uncertainty about its benefit.

Intuitively, a configuration has high uncertainty if SCULPTOR knows little about what the resulting routes will be and if the different potential routes will lead to large differences in the objective function. Specifically, for each adjacent configuration, SCULPTOR estimates the objective function's distribution at that configuration using Monte Carlo methods (as in Section 3.3.2) and computes the Shannon entropy [**?**] of the estimated distribution. (One could use other measures to determine how to explore, but entropy works well). By advertising a configuration with high uncertainty and learning the resulting routes, SCULPTOR resolves the uncertainty and refines its model in a way that it can apply to other configurations.

3.4 A Two Pronged Approach

Even with our probabilistic model, we cannot solve Equation (1) directly since it is a mixed-integer program with millions of constraints which is too large for general-purpose solvers. Greedy [**?**] and random [**?**] approaches find good solutions in this setting for simple objectives such as steady-state latency, but, without more intelligent search through the large space, those approaches can converge to poor solutions on other objectives (§5.4).

Instead, we split Equation (1) into an outer and inner optimization, solving for configurations, A, in the outer loop using gradient descent, and traffic assignments, w, in the inner loop using a general-purpose solver. Traffic assignments are continuous optimization problems and so are solved tractably with general-purpose solvers.

3.4.1 Outer Loop: Gradient Descent

To apply gradient descent to our problem, we create a real-valued auxiliary variable to A, , with entries between 0 and 1 representing "confidence" values that an advertisement to the corresponding ingress will be beneficial. We define gradients of our objective function with respect to this auxiliary variable by (a) computing the expected value of $G(R(), w)$ at *integer* entries of and (b) interpolating G (and thus its gradient) at non-integer values using sigmoids (as in prior work [**?**]). Since there are too many gradients to compute, we subsample gradient entries and track the largest ones (as in prior work [**?**]). We then threshold at 0.5 to translate real-valued confidences to whether or not we should advertise a prefix at an ingress.

Scaling: Minimization in the outer loop scales with the product of the number of ingresses and number of prefixes, but is completely parallelizable. Hence, the bottleneck is effectively the prefix/path measurement period (20 minutes - 2 hours per iteration).

Convergence: Gradient descent converges to a local minimum for bounded objectives [**?**]. Our evaluations show that SCULPTOR finds good solutions over a wide range of emulated topologies (§4.1), and converges quickly with thousands of UGs and ⟨`peering, prefix`⟩ pairs (§5). Allocating higher prefix budgets and adding richer advertisement capabilities (*e.g.,* BGP community tagging) can lead to convergence to a better minimum, 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 Monte Carlo simulations and the scaling behavior of each traffic allocation, which depends on the objective.

Convergence depends on the objective, G. Our objectives (§3.2.3) and many others can be expressed as linear programs, so solvers can find globally optimal solutions. Non-convex objectives such as Cascara's sometimes have efficiently computable solutions [**?**].

4 IMPLEMENTATION

We prototype SCULPTOR on the PEERING testbed [**?**], 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.)

In practice, Service Providers can use SCULPTOR *safely* without changing BGP announcements on production traffic and integrate SCULPTOR into automated frameworks. More details about how a Service Provider may do so are in Appendix B.

4.1 Emulating Client Traffic

We measure actual client latency from IP addresses to our PEERING prototype as in prior work [**?**], selecting targets according to assumptions about Vultr cloud's client base.

We first tabulate a list of 5M IPv4 targets that respond to ping via 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. 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 pinging 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 [**?**]). Vultr likely peers with networks with which it exchanges a significant amount of traffic [**?**], 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 ingresses. In an effort to focus on interesting 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 volume data, we emulate traffic volumes in two ways: first using randomly generated traffic and second using APNIC data [**?**].

To emulate client traffic volumes, we first randomly choose the total traffic volume of a site as a number between 1 and 10 and then divide volume up randomly among clients that `anycast` routes to that site. Client volumes in a site are set 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 many subsets of sites and emulated client traffic volumes, we demonstrate that SCULPTOR's benefits are not tied to any specific choice of sites or traffic pattern within those sites (thus demonstrating SCULPTOR can be applied to many Service Providers). Choosing a wider range of volumes or Zipf-like distributions leads to trivial solutions since a small fraction of sites/clients contain all of the volume (*i.e.,* importance).

A more realistic set of traffic volumes assigns clients an amount of traffic proportional to that client's AS population according to APNIC [**?**], which were shown to be representative of Service Provider traffic volumes [**?**]. Specifically, we assign the aggregate of all clients in an AS their APNIC client population (0 if the AS is not present), and divide volume evenly among the clients.

We found that the high-level results shown in Section 5 were the same using both emulated and APNIC volumes, and so report results for emulated traffic.

4.2 Deployments

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

We implement SCULPTOR with Nesterov's Accelerated Gradient Descent (§3.4.1) in Python [**?**]. We set $\alpha_l = 4.0$ and set the learning rate to 0.01 with decay over iterations. We solve traffic allocations (§3.4.2) with Gurobi [**?**].

Experiments in the Internet We assess how SCULPTOR performs on the Internet using RIPE Atlas probes [**?**], 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). We limit the scale of our deployment to 10 sites to avoid reaching RIPE Atlas daily probing limits (15k traceroutes/day). We choose a deployment with high geographic density rather than greater geographic coverage, as we believe the proximity creates a more interesting routing surface to solve (differences from `unicast` could be smaller, for example). These 10 sites were Miami, New York, Chicago, Dallas, Atlanta, Paris, London, Stockholm, Sao Paulo, and Madrid. 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 unique ingresses. Each probe has paths via approximately 60 ingresses. We use 12 prefixes.

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. 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. We evaluate SCULPTOR over deployments of size 3, 5, 10, 15, 20, 25, and 32 sites. Each size deployment is run at least 10 times with random subsets of UGs, UG demands, and routing preferences. The distribution of the number of /24s per peer is Pareto-like, so we consider random subsets of /24s through each ingress in a way that balances the number of unique /24s per ingress. Over all scenarios, we consider paths from clients in 52k prefixes (representing 31% of APNIC population [**?**]) to 873 ingresses. We use one tenth of the number of ingresses as the number of prefixes in our budget (10 prefixes for 3 site deployments, 60 prefixes for 32 site deployments). Prior work found that using tens of prefixes to improve performance was a reasonable cost [**? ?**].

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 exists

since we are not a Service Provider, but 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.

Experiments in the Internet For our first method of inferring client ingress links, we advertise prefixes into the Internet using the PEERING testbed [**?**], and measure actual ingress links to those prefixes using traceroutes from RIPE Atlas probes [**?**]. 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 in a few thousand networks. In cases where we cannot infer the ingress link from a traceroute, we use the closest 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 say the ingress link was AS1299 at Atlanta.

Simulations 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, though prior work found it held in 90% of the cases they studied. We conducted a sensitivity analysis on our preference models and found that SCULPTOR's performance gains are robust to variations to this specific model (e.g., random preferences).

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

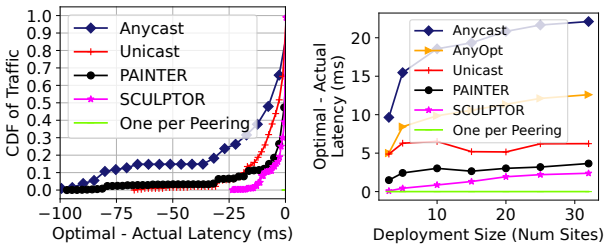
5.1 General Evaluation Setting

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 [**? ? ? ? ? ? ? ?**].

unicast: A single prefix announcement to all peers/providers at each site (one per site). Another common strategy used in today's deployments [**? ? ?**].

AnyOpt/ PAINTER: Two proposed strategies for reducing steady-state latency compared to anycast [**? ?**]. AnyOpt was one of the first works to systematically optimize ITE, while PAINTER built on that effort. (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 AnyOpt takes a long time to compute.)



(a) (b)

Figure 5: SCULPTOR provides the lowest steady-state latency both on the Internet (5a) and in simulation (5b).

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.)

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, as these statistics provide a more informative measure of latency improvement than averages.

5.2 Different Objectives Evaluation

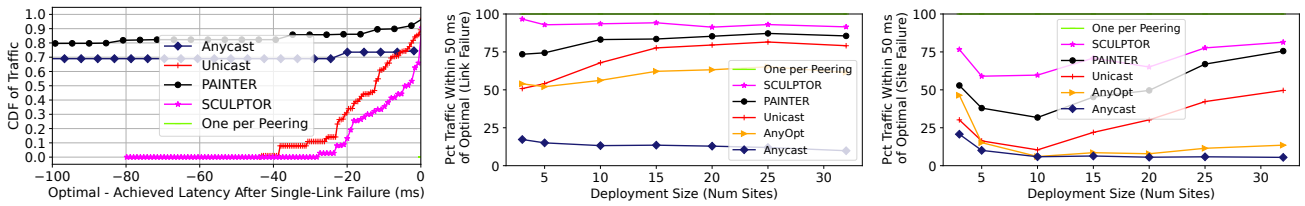
5.2.1 Handling Unseen Conditions

In minimizing user latency (§3.2.3), SCULPTOR achieves that objective both during steady-state and also during failures and conditions unseen during learning. Failure here could represent actual site failure or a heavily congested link along the path.

For context, at the scale of Service Providers that serve trillions of requests per day, improving a few percent of traffic by tens of milliseconds represents a significant improvement. Service Providers recently emphasized that small percentage gains are important [**? ?**].

Lower Latency in Steady-State. **Internet Deployment** Figure 5a shows a CDF of the difference in latency between each solution and One-per-Peering for all UGs, weighted by traffic. SCULPTOR outperforms other solutions and is only 2.0 ms worse than (the unreasonably expensive) One-per-Peering solution on average, whereas the next best solution, PAINTER, is 5.5 ms worse. SCULPTOR also serves 91.8% of traffic within 10 ms of the latency with which One-per-Peering serves it, whereas the same is true for only 88% of traffic for PAINTER.

Simulations Figure 5b shows the average latency compared to the One-per-Peering solution over all emulated deployments at each deployment size. Average latency for SCULPTOR ranges from 0.1 to 2.4 ms worse than One-per-Peering. The next-best solution (PAINTER) is on average 1.1 ms and 2.2 ms worse than SCULPTOR. Interestingly, unicast (5.7 ms worse than One-per-Peering) performs better than AnyOpt (10.0 ms



(a) **(b)** **(c)**

Figure 6: SCULPTOR improves resilience to failure both on the Internet (6a) and in simulation (6b, 6c).

worse than One-per-Peering), which could be due to the different setting AnyOpt was designed for. AnyOpt was designed to minimize latency without capacity constraints over provider connections, which does not capture that many Service Providers have many peers and limited capacity [**?**].

These average latency improvements translate into quantifiable routing inefficiency for different fractions of traffic (we include full results in Section D.1). SCULPTOR has on average 94.9% traffic within 10 ms of the One-per-Peering solution, 99.2% within 50 ms, and 99.9% within 100 ms. These percentages compare favorably to the next-best solution, PAINTER, which has on average 92.3% traffic within 10 ms of the One-per-Peering solution, 97.7% within 50 ms, and 99.2% within 100 ms.

Better Resilience to Failure. We next assess SCULPTOR's ability to gracefully handle a type of *unseen* condition—ingress failures and site failures. Examples include excessive DDoS traffic on the link/site (thus using the link/site as a sink for the bogus traffic), physical failure, resource draining, changes in latency on a path, and planned maintenance. Figure 6 demonstrates that SCULPTOR shifts traffic without overloading alternate links/sites more effectively than any other solution, *without reactive BGP changes* which could cause further failure (§2.3).

Here, 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 that use that component in steady-state. For example, if the Tokyo site fails, we report on the post-failure latency of UGs that were served from Tokyo before the failure and not of other UGs.

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

Internet Deployment Figure 6a demonstrates that SCULPTOR offers lower latency paths for more UGs during single link failure in realistic routing conditions. On average, SCULPTOR is 7.9 ms higher latency than One-per-Peering, compared to unicast which is 14.2 ms higher than One-per-Peering. PAINTER struggles to find sufficient capacity for UGs, overloading 69.6% of traffic.

Site failures (shown in Section D.2) show similar results. On average, SCULPTOR is 13.1 ms higher latency than One-per-Peering, while the next-best solution, unicast, is 25.1 ms higher. PAINTER again performs poorly, with 95% of traffic overloaded during site failure.

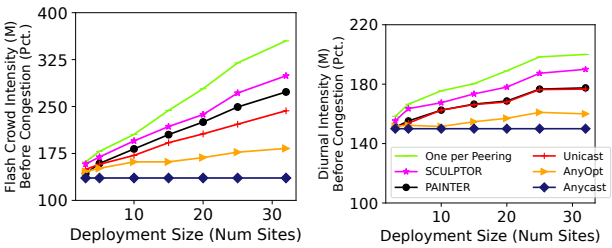
Simulations We show the fraction of traffic within 50 ms of the One-per-Peering solution for link and site failures in Figure 6b and Figure 6c (further results are in Section D.2).

For single-link failures, SCULPTOR ranges from 0.7 ms and 9.6 ms worse than One-per-Peering on average. SCULPTOR also avoids more overloading, with only 1.3% of traffic being congested on average, while PAINTER (the next-best solution) leads to 3.7% of traffic being congested on average. Single-site failure exhibits similar trends where SCULPTOR is 8.7 ms worse than One-per-Peering's latency and has 18.5% traffic overloaded, on average, while PAINTER leads to 14.7 ms worse latency than One-per-Peering's latency and 34.8% overloading on average.

SCULPTOR also has 78.9% of traffic within 10 ms of One-per-Peering's latency on average during link failure, 93.3% within 50 ms, and 97.3% within 100 ms. The next-best solution, PAINTER, only has 66.1% within 10 ms, 81.8% within 50 ms, and 89.3% within 100 ms. Site failures show similar trends (full results are in Section D.2).

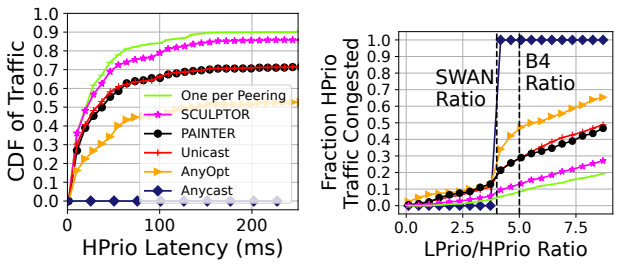
Efficient Infrastructure Utilization. Figure 7 shows that installing more capacity to handle peak loads during unseen scenarios is not always necessary with better routing — SCULPTOR finds ways to distribute load over existing infrastructure to accommodate increased demand. We quantify this improved infrastructure utilization under two realistic traffic patterns that SCULPTOR *did not explicitly consider in its learning process* — flash crowds and diurnal effects.

Methodology We define a flash crowd as a transient traffic increase for users in a region. Examples include content releases that spur downloads in a particular region, and localized DDoS attacks. Since increased demand is localized, we can spread excess demand to other sites, which is a cheaper option than installing more capacity (see Section 3.2.1 for our cost model). Links are still provisioned 30% higher than anycast traffic volume as in the rest of Section 5.2.1.



(a) (b)

Figure 7: SCULPTOR improves infrastructure utilization under flash crowds and diurnal traffic patterns so that Service Providers can underprovision compared to peak loads.



(a) (b)

Figure 8: SCULPTOR routes high-priority traffic with low latency (8a) and minimal congestion from low-priority traffic (8b). We inferred LPrio/HPrio ratios for SWAN [?] and B4 [?] from those papers.

To generate Figure 7, 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, but all the allocations are across a single set of routes calculated based on the original (non-flash) traffic. For a target region, we increase M until any link experiences overloading, then find the lowest such M value across regions. For example, if a 60% increase in traffic for Atlanta clients creates overload (and no values < 60% did for any region), we call $M = 160$% the critical value of M.

Our diurnal analysis in Figure 7b 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 Service Provider demonstrates that these effects can cause large differences between peak and mean site volume [?]. 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 Section D.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 7 plots the average over simulations of critical M values that cause overloading for each deployment size under flash crowds and diurnal effects computed using emulated deployments.

Figure 7a shows that SCULPTOR finds ways to route more traffic during a flash crowd without overloading than other solutions. For deployments with 32 sites, SCULPTOR can handle flash crowds at 2× more than the expected volume, creating a 3× savings in provisioning costs compared to anycast, and 26% more savings than PAINTER. Figure 7b shows that SCULPTOR also handles more intense diurnal traffic swings, allocating traffic to paths without overloading for 24% more intense swings than both PAINTER and unicast with 32 sites.

Hence, instead of scaling deployment capacity to accommodate peak time-of-day traffic, Service Providers can redistribute traffic to sites in off-peak time zones.

Using backup paths does not imply reduced performance as Service Providers can move less latency-sensitive traffic onto these backup paths so as to not impact high-level applications. We explore this idea in Section 5.2.2.

5.2.2 Handling Multiple Traffic Classes

We also evaluate SCULPTOR's ability to satisfy multiple traffic classes. We split traffic into high and low-priority. The objective is to route high-priority traffic to low-latency routes while limiting congestion on those routes from low-priority traffic. We do not penalize cases where low-priority is congested, but do limit the maximum amount of any traffic on a link to 10× the capacity of the link to avoid solutions where all low-priority is placed on one link, as this solution would lead to low goodput for low-priority traffic (in practice, UGs would lower sending rates in response to congestion). We solve SCULPTOR on a single emulated 32 site deployment.

In Figure 8b we vary the amount of low-priority traffic as a multiple of high-priority traffic volume and compute the fraction of high-priority traffic congested. Links are provisioned to 5× the high-priority traffic volume (different from Section 5.2.1), as that is roughly the LPrio/HPrio ratio reported in prior work [? ?]. There is insufficient global capacity to route all traffic for all LPrio/HPrio over 4.0, thus the jump in anycast congestion in Figure 8b. SCULPTOR finds strategies that allow us to route more low-priority traffic with less congestion than all other approaches. For example, when LPrio/HPrio = 5, SCULPTOR achieves half as much congestion as PAINTER and unicast.

Figure 8a also shows that SCULPTOR routes traffic with lower latency than other solutions (intercepts at the right of the graph show fractions of high-priority traffic not congested). Figure 8a uses LPrio/HPrio = 4, a midpoint between the ratios seen in SWAN [?] and B4 [?].

5.3 Minimizing Per-Site Traffic Costs

We also evaluate SCULPTOR's ability to minimize both latency and per-site traffic cost. Per-site traffic cost takes into account

that different sites may cost more or less to serve traffic due to, for example, the availability of power. We solve SCULPTOR on multiple emulated **TBD: 25** site deployment with both (a) completely random per-site costs, (b) a model of expensive vs cheaper sites where half the sites were 1.2× the cost of cheap sites, and (c) a model of per-site carbon cost taken from prior work **TBD: cite**. We do not claim to have a realistic model of site cost, but use many different site-cost functions to demonstrate that SCULPTOR's benefits are not specific to any specific assumption.

In all of our emulations, SCULPTOR found solutions near the One-per-Peering one, much closer than all other approaches. For example, for scenario (b) above SCULPTOR found solutions **TBD: X** percent lower cost and **TBD: y** ms less latency on average than the next-best solution, PAINTER. For scenario (c), SCULPTOR finds solutions that use **TBD: X** less carbon-cost and **TBD: z** ms less latency on average than the next-best solution, PAINTER.

5.4 Why SCULPTOR Works

Comparing More Options First, SCULPTOR compares far more advertisement strategies than the other solutions, so it has potentially better options to choose from. In our 32 site deployments, over 200 gradient steps SCULPTOR estimates latencies in approximately 20M scenarios across every UG. PAINTER only considers thousands (2k), and AnyOpt considers 1k (a configurable number, but the approach would not scale close to the numbers that SCULPTOR tries). SCULPTOR could estimate even more with more compute budget, while PAINTER and AnyOpt are fundamentally limited.

Conducting Fewer Advertisements SCULPTOR only conducts advertisements for exploration (§3.3.3) or for strategies that SCULPTOR thinks will yield good performance (§3.4). Since the key limited resource in finding good advertisements is time (§3.3.1), it is essential that SCULPTOR is confident that its gradient descent leads it to good performing advertisements, and so its exploration makes actual Internet announcements for configurations where its model is highly uncertain, improving the model such that future iterations of modeled announcements are more likely to yield accurate predictions and lead its search towards good configurations.

PAINTER measures advertisements that it thinks might be good, but its greedy approach means that it only considers a single scenario per advertisement iteration. AnyOpt spends time measuring potentially uninformative advertisements. Hence, both AnyOpt and PAINTER conduct a large number of advertisements on the Internet relative to the configurations they estimate, limiting the number of configurations they can explore. Figure 9a shows how the maximum entropy of the distribution of G on unmeasured advertisements (§3.3.3) exponentially decreases as SCULPTOR makes these advertisements on the Internet, and so it quickly grows confident that it does not need to issue more advertisements to find good

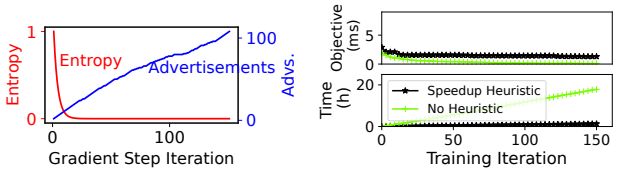


Figure 9: SCULPTOR's model entropy decreases exponentially with few path measurements (9a). Heuristic speedups can also speed computation, but may sacrifice convergence (9b).

strategies. This quick convergence manifests since we only have to predict the objective, G, not the paths (§3.3.2).

Horizontal Scaling SCULPTOR converges faster given more compute since gradient computations are parallelizable. PAINTER does not scale horizontally since it uses greedy search [?]. Some objectives additionally admit heuristics for finding approximately optimal traffic allocation solutions (*e.g.,* Appendix C). Other work similarly uses heuristics to quickly solve challenging problems [? ? ?]. Figure 9b shows that our heuristic (Appendix C) marginally hurts SCULPTOR's convergence (2 ms average latency difference) but requires drastically less compute (50× speedup).

6 RELATED WORK

Egress Traffic Engineering Prior work noted that traffic from Service Providers to users sometimes experienced suboptimal performance due to BGP's limitations [? ? ?]. SCULPTOR works in tandem with these systems, similarly working with BGP, but to engineer ingress traffic. Other work also shifts traffic to other links/sites to lower cost [? ? ?]. SCULPTOR adapts this idea in a new way but instead uses the public Internet to carry traffic to lower costs.

Ingress Traffic Engineering We compare SCULPTOR to other work on ingress traffic engineering [? ? ? ? ? ? ?] both in evaluation (§5) and discussion (§2.3). FastRoute coarsely balances users across anycast rings to respond to changing conditions [?], giving Service Providers much less control over the specific path users take. PECAN [?] and Tango [?] exhaustively expose paths between endpoints and so may provide resilience to changing conditions if the right paths were exposed. However, exposing all the paths does not scale to our setting since there are too many paths to measure. Recent work balances ingress traffic across sites during failure [?], but it is unclear how to extend this methodology to other objectives.

Other prior work built a BGP playbook to mitigate DDoS attacks [?], but it is unclear if those strategies scale to larger Service Providers (they tested on a few sites/providers). SCULPTOR works for Service Providers with thousands of peering links. Other work/companies create overlay networks and balance load through paths in these overlay networks to satisfy latency requirements [? ? ? ? ?]. SCULPTOR works alongside these overlay networks by advertising the external reachability of these nodes to create better paths through the overlay structure.

Intradomain Failure Planning Service Providers have shown interest in reducing the frequency/impact of failures in their global networks [**?** **?** **?** **?**]. SCULPTOR works alongside these systems by, for example, enabling Service Providers to shift traffic away from failed components/regions while still retaining good performance. Prior work also shifted traffic during peak times to limit cost/congestion [**?**], but did so using a private backbone. SCULPTOR's benefits are orthogonal to this prior work and useful for Service Providers without private backbones, as they use the public Internet to realize the same result. Other prior work tried to plan intradomain routes to minimize the impact of k-component failures [**?** **?** **?** **?** **?**]. SCULPTOR solves different challenges that arise due to the lack of visibility/control into potential paths and their capacities in the interdomain setting.

7 DISCUSSION & CONCLUSIONS

Unilateral control over intradomain traffic has enabled technology that improves performance, reliability, and flexibility, including fast reroute for quick recovery from failure [**?**], virtual routing and forwarding for flexible traffic engineering [**?**], and virtual output queueing for differentiated service [**?**]. Researchers have proposed making similar functionality available in interdomain settings, but, after decades of little adoption, it seems unlikely that such technologies will be widespread enough for Service Provider use. For example, intserv/diffserv [**?**] and L4S [**?**] are proposed frameworks for achieving differentiated service but show no signs of deployment. MIRO is a flexible interdomain multipath routing protocol that similarly shows no signs of deployment [**?**].

Rather than require any changes to the Internet, SCULPTOR uses a simple, black-box model of interdomain routing, BGP's flexibility, and the unused capacity of a Service Provider's global resources to give operators the benefits of those aforementioned technologies in the interdomain setting. For example, SCULPTOR can configure multiple paths per user to enable traffic failover and can set up different routes for different traffic classes to enable virtual output queueing despite lack of control on the interdomain path. SCULPTOR is a step towards providing Service Providers with programmable interdomain networking primitives so that they can give us the performance that our services increasingly need.

A DEFINITION OF TERMS

Table 1 includes a table of common terms and expressions used in Section 3, for ease of reference.

B CONSIDERATIONS FOR REAL-WORLD DEPLOYMENTS

We give a high-level overview of how a real cloud may practically, safely implement SCULPTOR. The measurement, advertisement, and learning process described in Section 3 is meant to be carried out before SCULPTOR serves any production traffic, as in prior work [**?** **?** **?**].

SCULPTOR requires latency measurements to each ingress. To obtain these, the Service Provider should direct clients to measure latency towards a test prefix using a measurement system similar to those found in the literature [**?** **?** **?** **?**]. Note that this process *does not involve production traffic*. The Service Provider should then advertise the test prefix to one ingress at a time and record client latency to ingress mappings.

With these initial measurements, Service Providers can initialize the learning process described in Section 3.3.3 using test prefixes and client-orchestrated measurements. These measurements also do not involve production traffic. Prior work also shows that BGP routing changes minimally disrupt routers [**?**], so SCULPTOR's learning process does not affect production traffic.

After convergence, SCULPTOR can slowly shift production traffic over to the optimal learned configuration. Note that, since most traffic is already routed along a good path [**?** **?** **?** **?**], SCULPTOR may only need to shift a small fraction of traffic or do so infrequently [**?**], mitigating the blast radius of configuration changes. The learning process is very infrequent, as prior work found that advertisements do not need to be updated even after a month [**?**].

C HEURISTICS FOR FAST COMPUTATION

With sufficient resources, solving Equation (1) using the two pronged approach outlined in Section 3.4 is feasible. Given our advertisement rate limitations (§3.3.1) we wish to compute gradients in tens of minutes. With 100 Monte Carlo simulations per gradient and tens of thousands of gradients to compute, we thus need to solve millions of linear programs at each gradient step. Meta recently demonstrated that they could solve millions of linear programs in minutes using their infrastructure [**?**].

As we do not have these resources, we instead use efficient heuristics to approximately solve each linear program. Our heuristics depend on the structure of our objective function, and so are specific to our choice of average latency (Eq. (2)). During optimization we observed that these heuristics tended to yield accurate estimates for G despite inaccuracies in predicting any individual UG's latency or link's utilization.

Variable	
A	Advertisement strategy over peers and prefixes
w	Traffic allocation to routes
R	Routing function, which maps advertisements to routes. Unknown at first but updated according to measurements
p	Prefix, one of many in a prefix budget (limited due to expenses)
l	Ingress link, Service Providers often have thousands
$c(l)$	Link l capacity
UG	User group, users that route similarly. *e.g.,*/24's or metros
$w(p, \text{UG})$	Amount of traffic from users in UG we direct towards prefix p
G	Objective function, *e.g.,* average latency
$v(\text{UG})$	Traffic volume for users UG
F_l	Failure on link l
β	Tradeoff parameter between minimizing latency and maximum link utilization
M	Maximum link utilization
α_l	Importance parameter for failure on link l, *e.g.,* if one failure is more likely than the other
Expression	
$l = R(A)(p, \text{UG})$	Route (ingress link l) resulting from A, for UG towards p
$\mathcal{L}(\text{UG}, l)$	Latency for UG across l
$G(R(A), w)$	Objective function value under A and w

Table 1: Table of common terms and expressions.

C.1 Capacity-Free UG Latency Calculation

Minimizers of Equation (2) balance traffic across links to satisfy capacity constraints. We instead temporarily ignore capacity constraints and assign each UG to their lowest-latency path. This approximation solves two problems at once. First, we do not need to solve a linear program to compute traffic assignments. Second, we can now analytically compute the distribution of our objective function, G, over all possible realizations of $R(A)$. Analytically computing this distribution is useful, since it lets us compute entropy (§3.3.3) and expected value (§3.3.2) without Monte Carlo methods.

To see why we can compute the distribution exactly, notice that the objective function is a composition of functions of the form $min(X, Y)$ and $X + Y$ since we choose the minimum latency path across prefixes for each UG and average these minimums. We compute UG latency distributions for each prefix exactly using our routing preference model, and use analytical methods to efficiently compute the distributions of the corresponding minimums and averages.

C.2 Imposing Capacity Violation Penalties

It could be that such minimum latency assignments lead to congested links. We would like to penalize such advertisement scenarios, and favor those that distribute load more evenly without solving linear programs. Hence, before computing the expected latency, we first compute the probability that links are congested by computing the distribution of link utilizations from the distribution of UG assignments to paths. We then inflate latency for UGs on likely overutilized links.

That is, for each possible outcome of UG assignments to links, we compute link utilizations and note the probability those UGs reach each link. We then accumulate the probability a link is congested as the total probability over all possible scenarios that lead to overutilization. We discourage UGs from choosing paths that are likely congested using a heuristic — we emulate the impact of overloading by inflating latency in this calculation for UGs on those paths proportional to the overutilization factor. After emulating the effect of this overloading, we recompute the expected average UG latency *without changing* UG *decisions*. We do not change UG decisions as doing so could induce an infinite calculation loop if new decisions also lead to overloading. This heuristic penalizes advertisement strategies that would lead to many overutilized links if every user were to choose their lowest latency option. We show an example in Figure 10.

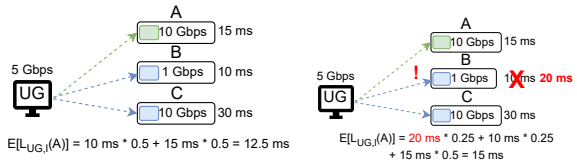
D FURTHER RESULTS

D.1 Steady-State Latency

In addition to quantifying average latency during steady-state, we also compute the fraction of traffic within 10 ms, 50 ms, and 100 ms of the `One-per-Peering` solution during steady-state. Figure 11 shows how SCULPTOR compares more favorably to the `One-per-Peering` solution than other advertisement strategies.

D.2 Latency During Link/Site Failure

We show additional evaluations of SCULPTOR during link and site failures. **??** shows that SCULPTOR provides good resilience



(a) **(b)**

Figure 10: A UG has a path to two prefixes, green and blue. The green prefix is only advertised via ingress A and the blue prefix is advertised via two ingresses B and C, each of which are equally likely for this UG. If the UG prefers B over C, we assign the client to the blue prefix (10 ms) while if the UG prefers C over B we will assign the client to the green prefix (15 ms). Hence the initial expected latency is the average of 10 ms and 15 ms corresponding to whether this UG prefers ingress B or C (10a). However, ingress B does not have sufficient capacity to handle this UG's traffic and so is congested with 50% probability corresponding to the 50% probability that this user prefers ingress B. We artificially inflate this UG's expected latency (and all other UG's using ingress B) to reflect this possible overutilization (10b).

to site failures on the Internet, and Figure 12 shows similar results for link and site failures in our simulations.

D.2.1 Internet Deployment

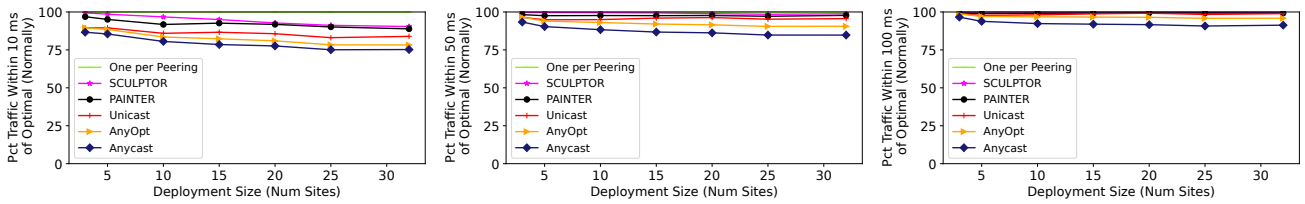
SCULPTOR provides better resilience than other solutions to site failures on the Internet, yielding no congested traffic and on average 18.1 ms worse than `One-per-Peering`. The next-best solution, `unicast`, yields 16.1% overloading and, for uncongested traffic, 28.1 ms worse latency than `One-per-Peering` on average.

D.2.2 Simulations

SCULPTOR provides good performance in simulation.

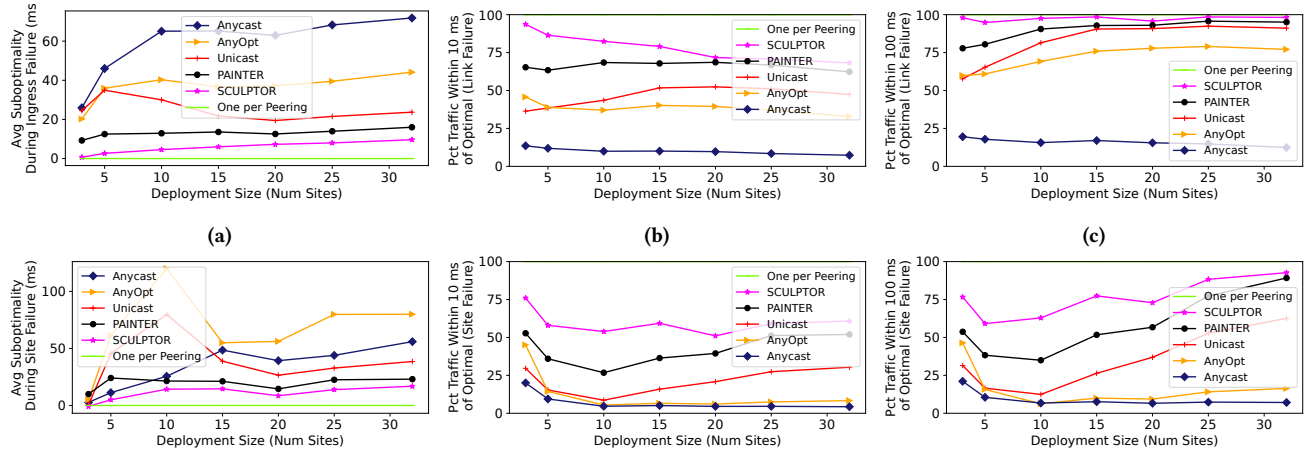
D.3 Infrastructure Utilization Further Results

In **??** we show the shape of our diurnal curve taken from observed diurnal traffic patterns at a Service Provider [**?**]. We use this diurnal traffic pattern to evaluate how well balances traffic across sites (§5.2.1). Specifically, we linearly interpolate 6 points on the purple curve (Edge Node 1) in Figure 1 of that paper, capturing the essential peaks and troughs.



(a) (b) (c)

Figure 11: SCULPTOR lowers average latency and increases the fraction of traffic within critical thresholds of their optimal latency during normal operation.



(a) (b) (c)

(d) (e) (f)

Figure 12: SCULPTOR lowers average latency and increases the fraction of traffic within critical thresholds of One-per-Peering latency during both link and site failures over many emulated deployments.