Opportunities and Challenges in Service Layer Traffic Engineering

Gangmuk Lim[*]
UIUC

Aditya Prerepa[*]
UIUC

Brighten Godfrey
UIUC and Broadcom

Radhika Mittal
UIUC

ABSTRACT

Optimizing request routing in large microservice-based applications is difficult, especially when applications span multiple geo-distributed clusters. In this paper, inspired by ideas from network traffic engineering, we propose *Service Layer Traffic Engineering* (SLATE), a new framework for request routing in microservices that span multiple clusters. SLATE leverages global knowledge of cluster states and multi-hop application graphs to centrally control the flow of requests in order to optimize end-to-end application latency and cost. Realizing such a system requires tackling several technical challenges unique to service layer, such as accounting for different request traffic classes, multi-hop call trees, and application latency profiles. We identify such challenges and build a preliminary prototype that addresses some of them. Preliminary evaluations of our prototype show how SLATE outperforms the state-of-the-art global load balancing approach (used by Meta's Service Router and Google's Traffic Director) by up to 3.5× in average latency and reduces egress bandwidth cost by up to 11.6×.

CCS CONCEPTS

• **Networks** → **Overlay and other logical network structures**; *Network design principles.*

KEYWORDS

Application-level Networking, Request Routing, Traffic Engineering, Microservices, Service Mesh

[*]Co-first authors.

ACM Reference Format:
Gangmuk Lim, Aditya Prerepa, Brighten Godfrey, and Radhika Mittal. 2024. Opportunities and Challenges in Service Layer Traffic Engineering. In *The 23rd ACM Workshop on Hot Topics in Networks (HOTNETS '24), November 18–19, 2024, Irvine, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3696348.3696871

1 INTRODUCTION

Modern cloud-native applications are built as a large collection of functional modules called microservices that carry out narrow roles. Microservice-based applications may easily have tens or hundreds of individual microservices and an even larger number of endpoints within the services; Uber has ≈4,000 microservices and ≈40,000 unique RPC endpoints [17, 27]. Serving any particular request may result in a tree of tens or hundreds of endpoint API calls across many microservices, interleaving network communication and application-level computation (Fig. 1). Increasingly, these microservices are being deployed across multiple geo-distributed clusters, in order to reduce the blast radius of failures, improve latency by locating services close to users, and utilize multiple cloud providers.

Such multi-cluster deployments of microservices introduce a new dimension for optimizing performance: *which cluster should a request for a given microservice be directed to?* The default option is to use a local replica, in the same cluster where the request arrives. But there are several reasons why a given request might need to be redirected to a remote cluster, such as when the local cluster is overloaded or a service is not available in the local cluster. Today, deployed systems generally use simple rules that extend basic load balancing. The state-of-the-art systems, Google's Traffic Director [9] and Meta's Service Router [22], spill requests over to a nearby cluster when the local cluster's load exceeds a threshold.

Our starting point is the observation that making multi-cluster request routing decisions is actually a much more subtle problem than load balancing or simple modifications thereof. Optimal request routing must account for tradeoffs between multiple resource types (cluster compute and latency, and inter-cluster network latency and cost), global allocation constraints, trees of dependent microservices, and more. This paper proposes Service Layer Traffic Engineering (**SLATE**), a new architectural framework for optimizing request routing in complex microservice-based applications that span multiple clusters. Intuitively, *global request-level route optimization becomes a traffic engineering problem*, akin to network-layer TE. Unlike traditional TE, which operates at the packet processing level, SLATE controls the flow of requests that dictates service-level utilization, which leads to a very different set of considerations.

Our goal in this paper is to explore the space of *opportunities* and *challenges* that result from the above perspective. To do so, we built an initial SLATE prototype with a hierarchical design having three components: Global Controller, Cluster Controller, and SLATE-proxy (data plane). SLATE operates in the "service" layer (e.g. as part of the service mesh infrastructure) that is separate from the application.

We evaluate SLATE in a multi-cluster K8S environment with a real topology from Google Cloud Platform (GCP) and different microbenchmark applications under various deployment scenarios. The evaluation shows there are significant opportunities to optimize request routing by carefully considering how much to offload to remote clusters given the application's latency profile; which clusters to route to; where in the application's call topology to route cross-cluster; and which subset of requests is most efficient and effective to route remotely. These considerations allow the prototype to outperform the state-of-the-art approach taken by Traffic Director and Service Router by up to 3.5 times in latency and reduce egress cost by up to 11.6 times.

While our initial prototype shows the promise of our approach, several challenges remain open for future research, which are distinct to the service layer and do not appear in, e.g., network-layer TE.

- *Latency prediction:* End-to-end latency of microservice-based applications depends on not only individual services but also where the request flows in the application call tree and invocation patterns between caller and callee services.
- *Difference in requests:* Requests that flow into a single service may have very different compute and network resource usage and call graphs. This results in different implications for routing, and the need for traffic classification.
- *Robustness to misprediction:* If latency predictions or traffic classification are inaccurate, the system should realize it and react appropriately.
- *Scalability and fast reaction:* Complexity of optimization increases with the number of clusters, services, and traffic classes.
- *Interaction with other systems:* Routing requests to different clusters can affect the behavior of autoscalers. Also, ideal request routing should consider application caching layer and data locality for stateful services.

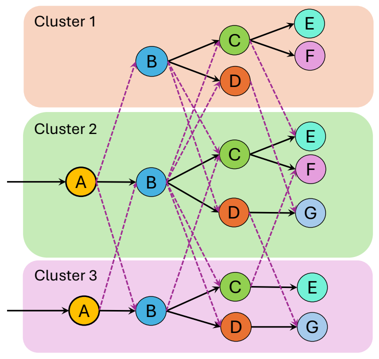


Figure 1: An example call tree of a microservice-based application deployed in multiple clusters. Nodes indicate microservices and edges indicate request flows. Some services are replicated in all clusters (e.g. B, C, D); others might be partially replicated (e.g., A, E, F, G). Solid and dotted edges indicate local and remote routing, respectively.

We discuss these in more detail in §5. In summary, we believe service layer TE offers significant opportunities for optimization and technical challenges to explore.

2 BACKGROUND & MOTIVATION

Multi-cluster Deployment of Microservices. Microservices are increasingly operated in multiple Kubernetes (K8S) clusters – ranging from tens to almost thousands of clusters [4, 6] – in many different regions and data centers. Multi-cluster deployments have several benefits: isolating failures of individual clusters; improving latency with proximity to different populations of users; reducing reliance on a single cloud provider; and taking advantage of price or feature differences across cloud providers. Microservices in a multi-cluster setup could be replicated in all clusters, or in only a subset of clusters due to various reasons such as security, data locality, or temporary service failure or decommissioning. A call tree of a single user request can cut across multiple clusters for performance reasons (e.g., if the local cluster is overloaded) or can be forced to do so due to partial replication (Fig. 1).

Cluster Autoscalers. To improve resource utilization and latency, there is an active line of work on job scheduling [10, 18] and autoscaling [2, 16, 20, 23, 24, 26], adjusting the number of service replicas or nodes as needed. While this work is making progress, it addresses resource allocation at the level of applications or containers. It has two disadvantages relevant for the focus of this paper. (1) It is too slow to react to sudden load changes that can happen at $\geq 1000\times$ faster timescales than autoscaling. Common autoscalers operate over seconds to minutes [14] including resource monitoring period, autoscaler interval period, and scaling overhead including container image pull and application initialization before starting serving requests. (2) Regardless of timescale, autoscaling has no direct control over how requests are routed among microservice instances, which in multicluster environments can significantly affect request latency and bandwidth cost. Our work deals with the complementary problem that happens at much finer granularity after provisioning the containers – the handling of individual requests.

Service Meshes and Load Balancing. As microservices result in more distributed applications, they require many network-related features – such as service discovery, encryption, telemetry, load balancing among replicas, and bridging multi-cluster deployments. Instead of implementing these features within the application, the application can utilize a service mesh such as Istio [11], Linkerd [15], or Cilium [5]. The service mesh is a separate layer from the application itself, with a centralized control plane and a distributed data plane. The data plane elements are typically "sidecar" containers (often the Envoy proxy [7]) paired with each microservice instance. Today, load balancing of requests among service replicas is done locally at each sidecar and uses relatively simple policies like round-robin, consistent hashing, or least outstanding requests. This can perform well under the assumption that all server hosts are within the same cluster. However, these assumptions do not hold in a multi-cluster environment where requests can span the clusters.

Current Global Load Balancing. Some providers and planet-scale deployments have global load balancing strategies which enable cross-region load balancing – specifically, Google's Traffic Director [9] and Meta's ServiceRouter [22]. However, both systems are manually-configured capacity-based heuristics, which make local, greedy decisions for server choice and only consider the latency and cost implications of a single RPC hop (more details about the existing global load balancing system will be explained in § 4). We find that there are a rich set of scenarios where global knowledge and optimization can yield substantially better latency and cost results.

Surveying Cluster Operators. We surveyed[1] the Istio community, one of the most widely adopted service meshes, on their Kubernetes multi-cluster deployment patterns to understand the need for optimizing request routing. Full results are available at [8]. The respondents[2] ran a median of ten to nineteen production clusters. 53% of respondents

[1]Our institution's IRB reviewed the survey and determined that it is not human subjects research and did not require IRB approval.

[2]The total number of responses is 31. Four of them were excluded since they do not run multi-cluster and have less than 10 nodes. The respondents of the survey were from a variety of internet businesses at varying scales, from 2 clusters and a few nodes to over 50 clusters and thousands of nodes.

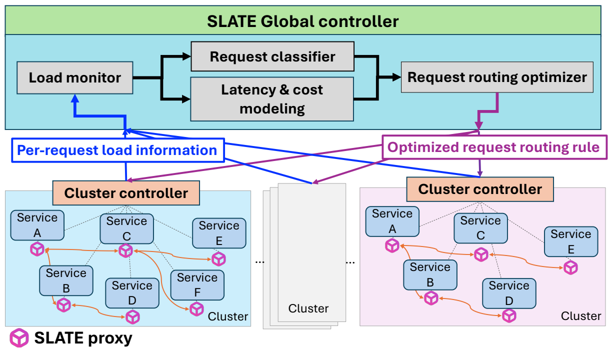


Figure 2: SLATE system architecture

deployed at least one service in multiple clusters (called a multi-cluster service). In these responses, 48% of services deployed are multi-cluster services. Among the multi-cluster service responses, most stated that they are suffering from considerable load imbalance between clusters – 50% of respondents said this occurs for hours or longer, and 20% for seconds or minutes. Out of our respondents, 81% utilized cross-cluster routing, and cited various reasons (general load balancing, minimizing latency, absence of a certain service in clusters, data locality, etc.). However, all of them only rely on simple load balancing (i.e., round robin, least response time, consistent hashing), static load distribution [13], or locality-based failover [12]. None of the respondents claim to directly optimize for request latency or cost. The large majority of the respondents (90%) reported that cross-cluster routing optimization among multi-cluster services would be useful for reasons such as optimizing application request latency (67% of respondents), reducing bandwidth costs (62%), reacting to load bursts (48%), and optimizing cloud compute costs (33%). None of the respondents claim to use any sort of global load balancing system.

3 PRELIMINARY DESIGN FOR SLATE

The overarching goal of SLATE is to globally optimize the routing of requests to minimize latency and cost. To achieve this, SLATE employs a hierarchical architecture (Fig. 2) where SLATE-proxy is running in service meshes collecting fine-grained per-request stats, which are aggregated at cluster level controller and then sent to Global Controller to run end-to-end request routing optimization.

As mentioned in §2, routing in existing systems mostly focuses on load balancing, with some enhancements, but these only begin to scratch the surface of the problem which we argue is a deeper *traffic engineering* problem, that must simultaneously handle four key aspects:

(1) *What portion of inbound requests should be routed away from the local cluster when local cluster is overloaded?*
(2) *Which cluster should we route those requests to?*

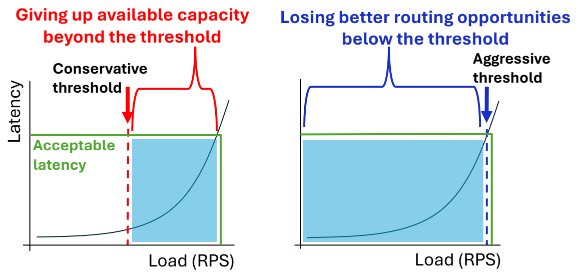

Giving up available capacity beyond the threshold
Losing better routing opportunities below the threshold
Conservative threshold
Aggressive threshold
Latency
Acceptable latency
Load (RPS)
Load (RPS)

Figure 3: Limitation in capacity-based offloading missing better load-to-latency tradeoff opportunities.

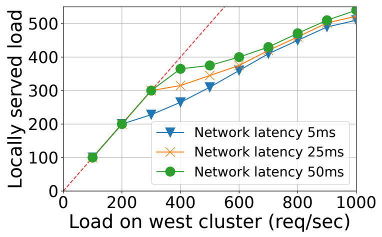


Figure 4: Empirical Cross-cluster routing threshold calculated by SLATE over different network latency and loads. The red dotted line indicates 100% local serving. Load in east cluster is constant (100 RPS).

(3) *Where in the application topology should requests make the cut when routing to remote clusters?*
(4) *Which subset (traffic classes) of requests should be routed to remote clusters?*

SLATE is designed to tackle this problem space. We describe our prototype design below, before presenting preliminary results in §4 and open technical challenges in §5.

3.1 Data Plane

The *SLATE-proxy* is the data plane element in SLATE – it is attached to each application instance as a sidecar extension as per the service mesh architecture. (Other realizations of the data plane, like a library-based sidecar within the application process, are compatible with our design.) Its two main jobs are *(1) telemetry* and *(2) request routing policy enforcement. SLATE-proxy* monitors and reports telemetry in each microservice replica to the Global Controller, including the load on the service, request specific information, latency, trace information, and request traffic classes. As a policy enforcer, it executes request routing at per-request level for each traffic class based on the policies given by Global Controller. The routing execution logic should be simple and heavily optimized since it is in the critical path of request processing.

3.2 Cluster Controller

The *Cluster Controller* acts as a metrics aggregator for a certain region, to avoid the scaling limitations of having every individual service connect to a global controller. The Cluster Controller has the responsibility of aggregating and relaying relevant per-service metrics to the Global Controller, as well as attaching the cluster ID of the metrics, as each service instance does not have the notion of which cluster it belongs to. Additionally, when the Global Controller has new rules for services in a cluster, those rules are pushed to the Cluster Controller, which then redistributes those rules to every relevant service.

3.3 Global Controller

Request routing optimizer. The *Global Controller* performs the global request routing optimization to minimize latency and cost, which is formulated as a Mixed Integer Linear Program. Our formulation uses the concept of a *traffic class*, which is a subset of requests at a service; the partitioning of requests into classes can be chosen by the controller (we return to this later). For each traffic class, the formulation specifies the load-to-latency relationship, call tree, and demand. Additionally, the formulation models clusters, inter-cluster network latency, and egress bandwidth costs. It then optimizes the specified objective, such as minimizing mean latency and/or cost, and generates optimized routing rules for each service, traffic class, and cluster, which are then pushed to the corresponding Cluster Controller. Each routing rule specifies the fraction of requests of a certain traffic class that should be sent to a certain cluster; standard load balancing will then select the server within the cluster. The execution of a rule may look something like the following: "When a request matches class X, send 60% of requests to the local cluster, 30% of requests to remote cluster B and the remaining 10% to remote cluster C". Notice how such an output addresses all four aspects of the problem space identified at the start of this section.

Deriving Classes. We define a request traffic class based on an arbitrary set of characteristics on a request. The overall goal is for traffic classes to allow more fine-grained traffic engineering decisions based on different resource implications (compute, bandwidth, and call graph) of each class. Deriving traffic classes heavily depends on the application behavior, along with the set of request attributes available to differentiate them. We find that a significant amount of differences in requests can be captured by classification using two broad request attributes: *(1) The service being called* and *(2) The action being invoked on that service (i.e. RPC name).* With the HTTP-based applications that SLATE operates on, we use the attributes of *(1) service*, *(2) HTTP Method*, and *HTTP Path.* This is not a perfect heuristic by any measure. However, applications are fundamentally potentially nondeterministic, and limiting the number of classes is required to have enough observations to accurately characterize average behavior of the class, and to scale the optimizer.

Latency Modeling. Accurately modeling application performance is extremely difficult especially at per-request level. However, with appropriate request classification, the average behavior can be predicted. Based on software architecture in common application serving frameworks, SLATE models latency of *each traffic class* in each service as a function of load with a variation of a M/M/1 queuing model.

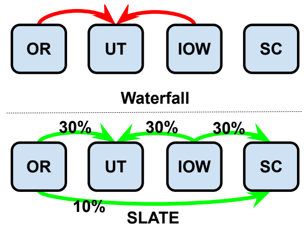
4 OPPORTUNITIES IN SLATE

We use the prototype of §3 to show there are significant opportunities for SLATE's traffic engineering perspective to improve global request routing.

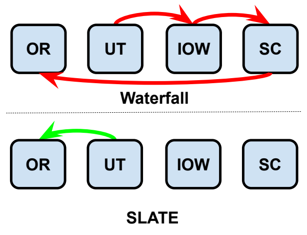
Baselines. We compare SLATE with the most advanced global load balancing systems we could find: Google's Traffic Director [9] and Meta's ServiceRouter [22], which both use a greedy, static capacity-based offloading policy. In both systems, each service has a predefined *capacity*, which is in terms of requests (of any type) per second or CPU utilization. Requests beyond this capacity are greedily offloaded to the nearest region with available capacity. We note that these systems are algorithmically roughly equivalent, depending on how they are configured; in what follows, we refer to this scheme as the *Waterfall* algorithm, using [9]'s terminology. We compare SLATE with Waterfall in the context of how well they tackle different aspects of the request routing problem space described in §3. We experimentally study these schemes, with an application composed of three microservices with ingress gateway chained linearly. Each microservice in the application used for fig. 6a, fig. 6b, and 6d performs simple file write operations. For fig. 6c, we used anomaly detection application which is described in § 4.4 in more detail. The applications are run in multi-node K8S cluster with inter-cluster network latency added using Linux's tc command to emulate the network latency.

4.1 How much to route to remote clusters?

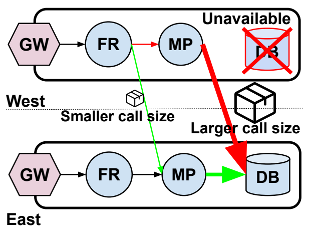
When one cluster receives more load than it can handle, *how much load needs to be routed to remote regions?* Waterfall routes all extra load beyond the configured static RPS threshold to the nearest regions with available capacity. Fig. 3 illustrates the consequences of using a static threshold. In the left graph, a conservative threshold forgoes opportunities to keep more traffic local, offloading to a remote cluster too early, paying more network latency unnecessarily. On the other hand, in the right graph, an aggressive threshold forces traffic to stay local when it may be better to offload that traffic to a remote cluster if the expected latency is lower.



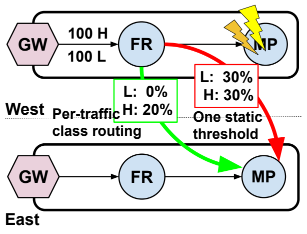
(a) How much



(b) Which cluster

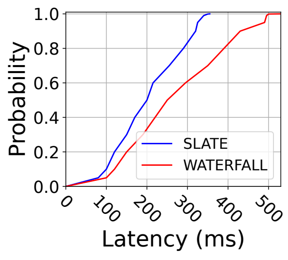


(c) Multi-hop

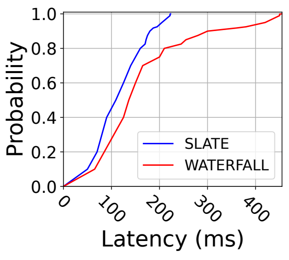


(d) Different traffic classses

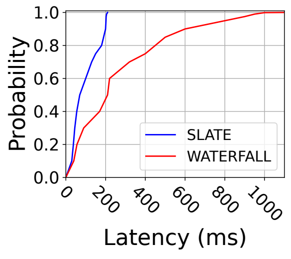
Figure 5: Example routing optimization cases in SLATE v.s. global load balancing algorithm in the existing traffic management systems (waterfall, and locality failover). We show the examples for each of the four questions but note that they are closely interlinked and should be considered together.



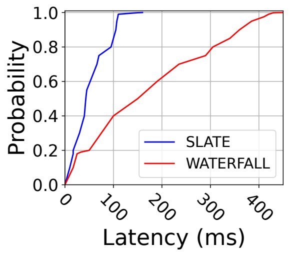
(a) How much



(b) Which cluster



(c) Multi-hop



(d) Different traffic classes

Figure 6: Prototype experiments. Each experiment uses the scenario described in § 5.

The optimal threshold changes with *(1) load conditions in every cluster*, *(2) inter-region network RTT*, and *(3) individual services' load-to-latency relationship*. Fig. 4 shows the empirical optimal routing threshold under different inter-cluster network latency and load conditions. In this scenario, there are two clusters (West and East), and the load (RPS) in West varies from 100 to 1000, while the other cluster is held at constant load (RPS=100). Fig. 6a shows the latency CDF of each system when West is overloaded. SLATE outperforms Waterfall by offloading only until it improves the latency. The optimal threshold would become more dynamic as the load in the East cluster starts to vary, the number of clusters grows, and application call tree becomes more complex. In addition, cost is another dimension that should be considered by an optimal solution. For example, if an administrator values cost over latency, an optimal request routing system (jointly optimizing latency and cost) should reflect it by keeping more traffic local.

4.2 Which clusters to route to?

In deciding which cluster to offload requests to, Waterfall uses a simple heuristic: choose the closest cluster with available capacity. However, this is greedy and prone to sub-optimality, since Waterfall does not consider load globally. In our example, we will use a real GCP topology consisting of clusters in OR (Oregon), UT (Utah), IOW (Iowa), SC (South Carolina). We have our emulator replicate the observed median latency of inter-region VM-to-VM traffic – OR-UT: 30ms, UT-IOW: 20ms, IOW-SC: 35ms, OR-SC: 66ms, and OR-IOW: 37ms. In the scenario of Fig. 5b where OR and IOW are overloaded, they greedily offload to UT, which is the closest to both overloaded regions and *technically* has available capacity; they route nothing to SC since it is located further with higher network latency than UT. However, due to this greedy decision, UT is running at capacity and thus requests incur higher average latency. A latency-optimal solution would utilize the SC cluster. One might think of tweaks to Waterfall for some cases, but fundamentally, optimal request routing involves a matching problem for which greedy algorithms can perform poorly [3]. Fig. 6b shows the latency CDF in the aforementioned application for the same load conditions where OR and IOW are overloaded.

4.3 Where in the topology to route?

Most conventional load balancing algorithms (including Waterfall) are *single-hop*, meaning the load balancing decision for a request only depends on the state of the corresponding service's replica pool. However, a single call to a microservice can spawn entire trees of subsequent calls. Therefore, the effect of a load balancing decision early in the call tree can have ripple effects throughout the rest of the call tree. Fig. 5c shows an anomaly detection application deployed in multiple clusters. *FR* is the frontend, *MP* is a metrics processor running an anomaly detection algorithm, and *DB* is a database storing metrics (e.g., Prometheus [21]). *MP* pulls large amount of metrics data from *DB*. In the us-west cluster, *DB* service is degraded or does not exist due to security constraints, regulation (e.g., GDPR), or a failure. In this scenario, Waterfall and existing service meshes will do locality failover load balancing, where requests will cross the cut between clusters at $MP \rightarrow DB$ (red arrow in Fig. 5c).

This is not optimal in terms of egress cost, since the response size for $DB \rightarrow MP$ is roughly ten times larger than $MP \rightarrow FR$. If a request routing algorithm had the foresight that this would occur, requests could have been routed earlier across clusters in the topology at $MP \rightarrow FR$ (green arrow in Fig. 5c). In our experiment, SLATE achieves 11.6× less egress cost. In addition, routing with knowledge of multiple future hops is essential for latency minimization. With multi-hop knowledge, requests destined for an overloaded service can be routed to remote clusters before they run into those services. In fig. 6c, Waterfall shows high latency since it only starts to offload at *MP* in West whereas SLATE offloads at both at *FR* and *MP*.

4.4 Which subset of requests to route?

Existing load balancers treat the inbound requests at a service as a homogeneous pool. While it is possible to make decisions for the "average" request, this ignores potentially large differences in the behavior of requests. Requests at a single service may have dramatically different load (in terms of compute, disk, or network usage), and completely different call trees which in turn cause different load on other services. SLATE's traffic class technique gives a way of defining smaller granularities which allow it to discover substantially more optimal routing opportunities.

In Fig. 5d, each service has two request classes: *L* and *H*, where *H* is significantly more expensive than *L*. The service is overloaded by high volumes of *H* requests. Waterfall evenly offloads the same fraction of requests in each request class, whereas SLATE can account for differences in computational load between the request classes and offload a smaller number of just *H* requests. Fig. 6d shows the latency result of the real application in the same set up. There are many other scenarios with even more dramatic differences – e.g., one request class could spawn sub-requests that need to query a database in the origin cluster, whereas another request class performs only stateless compute and offloading it is much cheaper and lower latency.

Summary. The examples above have some common themes – request routing is a *non-local problem* (across global clusters, and across multi-hop call graphs), and it is a *rich* problem (involving tradeoffs between service's latency profiles and network latency, request classes with diverse behavior, and more). Instead of baking heuristics into limited load balancing decisions, fundamentally viewing request routing as a *traffic engineering* problem (and explicitly modeling the optimization problem) yields significant potential benefits even with our relatively simple prototype.

5 TECHNICAL CHALLENGES

Several technical challenges remain unresolved in Service Layer Traffic Engineering. We describe them below to promote future research.

Traffic classification. Finding the right traffic classes is critical to effectively steer request traffic. The simplest option is to treat all requests homogeneously as a single class (similar to Waterfall), but as we have seen in Fig. 6d, such routing will miss large opportunities for optimization. At the other end of the spectrum, an extremely large number of classes could more accurately characterize traffic in principle, but makes it hard to get enough samples of each class to predict its behavior, and worsens performance of the centralized optimizer. Finding the right tradeoff with "just enough" meaningful classes is the key. The majority of requests in a meaningful traffic class should spawn the same child call graph and consume similar type and amount of resources in each microservice. As a potential future research direction, more advanced techniques, such as machine learning, could be applied to derive a small yet precise set of classes based on various request features, including service name, method, URL, and request headers.

Latency prediction. Our prototype showed that even a relatively simple model of latency can yield significant gain over the state of the art. A highly accurate latency model would improve SLATE's decisions, but is nontrivial due to several considerations. First, it is desirable to learn latency profiles dynamically in production, rather than profiling offline, where it may be difficult to replicate production conditions. Second, latency at a service S is a function of not only utilization at S, but also of the latency of back-end services that S depends on. This is important because SLATE's actions may modify those back-end latencies, creating complex dependencies on modeling S's latency. Third, differentiating performance by traffic class brings additional complexities: each class can have a different latency profile due to different call tree or different execution path within the same service. Fourth, latency will depend on resources on the specific machine, including its hardware and dynamic resource allocation among workloads sharing a machine.

Resilience to prediction error. SLATE leverages the predictions discussed earlier, but it must account for inherent unpredictability. Inaccuracies can stem from various sources, such as limited expressiveness in the latency model, noisy neighbors, changes in resource allocation, etc. These inaccuracies could cause SLATE to shift traffic in clearly suboptimal ways or even degrade performance rather than improve it. A promising design approach would be to use the optimizer's output as a guideline, without fully relying on it. For instance, if the optimizer suggests increasing the fraction of requests routed to a certain cluster by 50%, SLATE could implement incremental increases of, say, 10%, evaluate the system objectives (latency and cost) using real-time telemetry, and proceed only if the objectives improve as predicted. Any observed discrepancies would signal the need for reprofiling or active adjustments to the routing rules. However, how to handle such inaccuracies while still achieving the optimization objectives robustly remains an open question. Making the routing system resilient to these inaccuracies is an interesting direction for future research.

Scalability & Fast reaction. The request routing system for user-facing, latency-sensitive applications must be able to react to microbursts, so an optimization time on the order of seconds for large-scale deployments is desirable. To achieve this, two key components are required: (1) a scalable control plane and (2) a low-overhead data plane. The optimization problem run by SLATE's controller expands with the number of clusters, services, and traffic classes. Although heuristics have been developed for network-layer TE (multicommodity flow) [1, 19] and might provide useful inspiration, ours is a richer problem involving trees, multiple resource types, and nonlinear latency functions that would need a different set of acceleration techniques.

Interaction between request routing and autoscaler. Request routing decisions in the service layer can affect the autoscaler's behavior since cross-cluster request routing increases resource utilization in remote clusters. Co-designing the request routing layer and container resource allocation layer is an interesting area to explore.

Caching & data locality. Application layer caching and data locality are not explicitly considered in SLATE. Since internal application logic is not externally observable (and in particular to *SLATE-proxy*), it is hard to infer this behavior. Caching systems in microservice-based applications [25] is an orthogonal technique used to improve latency. Caching-aware request routing framework can further optimize the performance. It would be an interesting future research topic.

6 CONCLUSION

Microservice architectures effectively introduce a network among application instances. We have argued that this creates a rich problem space, elevating what was once simple load balancing into a traffic engineering problem requiring network-wide optimization. SLATE has shown, with a running implementation in the Envoy data plane, that this can bring significant benefits to applications in terms of lower latency, lower bandwidth cost, and less need for overprovisioning.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work was supported by NSF Award 2312714.

REFERENCES

[1] Abuzaid, F., Kandula, S., Arzani, B., Menache, I., Zaharia, M., and Bailis, P. Contracting wide-area network topologies to solve flow problems quickly. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (2021), pp. 175–200.

[2] Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., and Zhang, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proc. USENIX NSDI* (2017).

[3] Balkanski, E., Faenza, Y., and Périvier, N. The power of greedy for online minimum cost matching on the line. In *Proceedings of the 24th ACM Conference on Economics and Computation* (New York, NY, USA, 2023), EC '23, Association for Computing Machinery, p. 185–205.

[4] Carey, S. Why Mercedes-Benz runs on 900 Kubernetes clusters. https://www.infoworld.com/article/3664052/why-mercedes-benz-runs-on-900-kubernetes-clusters.html, 2022.

[5] Cilium. Cilium. https://cilium.io, 2021.

[6] Cloud Native Computing Foundation. CNCF Annual Survey 2021. https://www.cncf.io/reports/cncf-annual-survey-2021/, February 2022.

[7] Envoy. Envoy. https://www.envoyproxy.io/, 2021.

[8] Gangmuk Lim, Aditya Prerepa, B. G. R. M. Survey on multi-cluster deployments and cross-cluster routing in Istio community. https://servicelayernetworking.github.io/assets/images/slate/multicluster_survey_result-public.pdf, 2024.

[9] google. Google Traffic Director. https://cloud.google.com/traffic-director/docs/overview, 2019.

[10] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. Multi-resource packing for cluster schedulers. In *Proc. ACM SIGCOMM* (2014).

[11] Istio. Istio. https://istio.io, 2021.

[12] Istio. Istio locality failover load balancing. https://istio.io/latest/docs/tasks/traffic-management/locality-load-balancing/failover/, 2023.

[13] Istio. Istio locality weighted distribution load balancing. https://istio.io/latest/docs/tasks/traffic-management/locality-load-balancing/distribute/, 2023.

[14] Kubernetes. Kubernetes Cluster Autoscaler Reaction Time. https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#how-fast-is-hpa-when-combined-with-ca/, 2023.

[15] Linkerd. Linkerd. https://github.com/linkerd/linkerd2, 2021.

[16] Luo, S., Xu, H., Ye, K., Xu, G., Zhang, L., Yang, G., and Xu, C. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing* (2022), pp. 355–369.

[17] Mace, J. End-to-End Tracing: Adoption and Use Cases. Survey, Brown University, 2017.

[18] Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., and Alizadeh, M. Learning scheduling algorithms for data processing clusters. *Proc. ACM SIGCOMM* (2019), 270–288.

[19] Narayanan, D., Kazhamiaka, F., Abuzaid, F., Kraft, P., Agrawal, A., Kandula, S., Boyd, S., and Zaharia, M. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 521–537.

[20] Qiu, H., Banerjee, S. S., Jha, S., Kalbarczyk, Z. T., and Iyer, R. K. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *Proc. USENIX OSDI* (2020).

[21] Rabenstein, B., and Volz, J. Prometheus: A Next-Generation monitoring system (talk). USENIX Association.

[22] Saokar, H., Demetriou, S., Magerko, N., Kontorovich, M., Kirstein, J., Leibold, M., Skarlatos, D., Khandelwal, H., and Tang, C. ServiceRouter: Hyperscale and minimal cost service mesh at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association.

[23] Venkataraman, S., Yang, Z., Franklin, M., Recht, B., and Stoica, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proc. USENIX NSDI* (2011).

[24] Yu, G., Chen, P., and Zheng, Z. Microscaler: Automatic scaling for microservices with an online learning approach. In *2019 IEEE International Conference on Web Services (ICWS)* (2019), IEEE, pp. 68–75.

[25] Zhang, H., Kallas, K., Pavlatos, S., Alur, R., Angel, S., and Liu, V. MuCache: A general framework for caching in microservice graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)* (Santa Clara, CA, Apr. 2024), USENIX Association, pp. 221–238.

[26] Zhang, Y., Hua, W., Zhou, Z., Suh, E., and Delimitrou, C. Sinan: ML-Based and QoS-aware resource management for cloud microservices. In *Proceedings of the Twenty Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (April 2021).

[27] Zhang, Z., Ramanathan, M. K., Raj, P., Parwal, A., Sherwood, T., and Chabbi, M. CRISP: Critical path analysis of large-scale microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (2022), pp. 655–672.