SLATE: Service Layer Traffic Engineering

Gangmuk Lim[*] *UIUC*
Aditya Prerepa[*,†] *UIUC & xAI*
Brighten Godfrey *UIUC*
Radhika Mittal *UIUC*

Abstract

In microservice-based applications, requests flow between many microservice instances across potentially multiple geo-distributed clusters. Today, the routing of requests is limited to simple load balancing, or extensions that spill requests to nearby clusters. We argue that the problem is more subtle, and that there are significant opportunities for improvement by viewing microservice request routing as a global traffic engineering problem. We present *Service Layer Traffic Engineering* (SLATE), a system that optimizes request routing in microservice deployments that span multiple clusters to minimize average latency and bandwidth cost. SLATE tackles challenges unique to the service layer, including multiple request traffic classes, multi-hop call trees, and service latency profiles. To achieve this, SLATE takes a unique hybrid approach combining global optimization and local exploration. SLATE outperforms state-of-the-art global load balancing by up to 18.3× in average latency and reduces egress bandwidth cost by up to 2.64× in a Kubernetes deployment of an open-source benchmark application, and shows resilient performance against dynamic changes due to its hybrid optimization approach. Our system is completely transparent to the application and can be seamlessly plugged into existing L7 proxy deployments, specifically Envoy.

1 Introduction

Modern cloud-native applications are built as a large collection of functional modules called microservices that individually carry out narrow roles. Large-scale applications may comprise thousands of microservices with tens of thousands of endpoints—Uber, for example, uses approximately 4,000 microservices and 40,000 unique RPC endpoints [45]. Processing a single request often triggers numerous subsequent requests across multiple microservices, producing a *call tree* that combines network communication and application computation. Increasingly, these microservices are deployed across geo-distributed clusters to enhance fault tolerance, reduce latency by positioning services closer to users, and spread risk across multiple cloud providers.

Such multi-cluster deployments of microservices introduce a new dimension for optimizing performance: *For each request in a call tree, to which microservice replica (in which cluster) should a given request be directed?* The default option is to use replicas in the local cluster, i.e., the same cluster where the request originated. But there are several reasons why a given request might need to be routed to a remote cluster, such as when the local cluster is overloaded or does not host replicas of the service. Today, deployed systems generally route requests with simple rules that extend basic load balancing. The state-of-the-art systems, Google's Traffic Director [13] and Meta's Service Router [36], spill requests (whether at the gateway, or deeper in the microservice call tree) over to a nearby cluster when the local cluster's load exceeds a fixed threshold. However, making multi-cluster request routing decisions is a much more subtle problem than per-hop load balancing or simple modifications thereof, which (as we will see in § 3) can be significantly suboptimal.

This paper proposes Service Layer Traffic Engineering (**SLATE**), a new architectural framework for optimizing request routing in microservice-based applications that span multiple clusters. Conceptually, global request-level route optimization becomes a traffic engineering problem, similar to network-layer traffic engineering (TE). SLATE elevates this concept from the network layer to the service layer. Unlike traditional TE, which operates at the packet level, SLATE manages the routing of service requests.

Concepts from network-layer TE – such as centralized control using optimization software – formed our starting point for SLATE. However, we quickly found that the shift from network layer to service layer introduced a new set of technical problems, with two being particularly challenging. First, while in the network layer each packet corresponds to a clearly defined source-destination path, *the effect of each request in the service layer is potentially diverse and not clearly defined*. A single request initiates a tree of service invocations spanning multiple microservices, with different computational resource demands at each node (i.e., service invocation) in the tree and different network resources to send requests and responses. Second, traditional TE can be modeled relatively directly and accurately so that an optimizer can find solutions; but *accurately modeling TE at the service layer, particularly to capture application latency, is difficult.* Individual hops in the call tree span applications, which means latency varies with load in potentially complex ways that may change with workload, rollout of new service versions, changes in resource allocation, or transient performance degradation; and modeling how different request types affect each others' latency becomes even harder. As a result, the optimizer could be thrown off and send the system into a suboptimal state, or even harm the system.

SLATE is built with a hierarchical design, consisting of a Global Controller, per-cluster Cluster Controllers and per-container data plane elements (SLATE-proxy). SLATE-proxy

[*] Co-first authors.
[†] Work done while at UIUC.

collects detailed metrics which are aggregated by Cluster Controllers and delivered to the Global Controller. The Global Controller uses this traffic demand and performance telemetry to compute optimized routing rules. To solve the challenge of *diverse request behavior*, SLATE introduces L7 traffic classes that partition requests at each service. These classes capture predictable patterns by summarizing distributions of call graphs and expected resource utilization, despite individual request unpredictability. This approach reveals dramatic optimization opportunities that would remain hidden when treating all service requests uniformly.

To solve the challenge of *accurately modeling the optimization problem*, instead of developing an extremely detailed model, we adopt the philosophy that the model may always be somewhat inaccurate and we need to be robust to that inaccuracy. SLATE thus adopts a novel hybrid approach: the Global Controller uses both an optimizer to calculate globally-optimal TE solutions within the limits of the theoretical model, along with a dynamic exploration-based algorithm that makes local adjustments to that solution. This algorithm, Performance-Aware Policy Adjustment (PAPA), continually tests small deviations from the optimizer's solution, and keeps moving in the direction of the change when it finds actual observed performance improves.

This paper makes the following key contributions:

- The first framework to formalize and globally optimize request routing for microservice-based applications, accounting for multi-hop call trees, L7 traffic classes, application latency modeling, and bandwidth cost.
- A novel hybrid approach that combines an optimizer with a local exploration algorithm, which continually seeks to improve upon the optimizer's routing rules, increasing robustness to a potentially inaccurate or stale model.
- Implementation and evaluation of SLATE on Kubernetes clusters with up to 480 cores, extending the Envoy proxy to create SLATE-proxy. Our evaluation used three benchmark applications (Hotel Reservation [11], Online Boutique [14], and a reconstructed Alibaba trace [28]) in different replication scenarios (complete and partial replication) including scenarios where the environment dynamically changes.

SLATE outperforms existing global load balancing systems (Google's Traffic Director and Meta's Service Router) by up to 18.3× in average latency during overloaded conditions, and reduces egress costs by up to 2.64× for the benchmark applications and 11.6× for microbenchmark application without compromising latency. Our evaluation further highlights the need for our hybrid optimization approach, with the combination of the model-based optimizer and PAPA's empirical local exploration producing better results than either alone, especially for complex application performance effects.

2 Background

Multi-cluster Deployment of Microservices. Microservices are increasingly operated in multiple Kubernetes (K8S) clusters – ranging from tens to almost thousands of clusters [5, 7] – 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 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. We surveyed cluster operators to further motivate the problem and learned multi-cluster service deployments are common, but only simple load balancing is used today (see Appendix B).

Cluster Autoscalers. There is an active line of work on job scheduling [16, 30] and autoscaling [2, 29, 34, 39, 43, 44], adjusting resource allocation at container level. While this work is making progress, it addresses resource allocation at the level of applications or containers. In particular, (a) it is too slow to react to sudden load changes that can happen at $\geq 1000\times$ faster timescales than autoscaling; and (b) regardless of timescale, autoscaling has no direct control over how requests are routed among microservice instances. 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 app can utilize a service mesh such as Istio [21], Linkerd [27], or Cilium [6]. 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 [8]) 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. These can perform well if all server hosts are within the same cluster, but not in a multi-cluster environment where requests can span the clusters.

Current Global Load Balancing. Some planet-scale deployments have global cross-region load balancing strategies – specifically, Google's Traffic Director [13, 15] and Meta's ServiceRouter [36]. As these are the most advanced global load balancing systems we could find, we describe them in more detail. In both systems, each service has a predefined *capacity*, defined in terms of requests (of any type) per second (RPS). This capacity is usually defined as RPS a service can handle before average latency goes up or a certain target CPU utiliza-

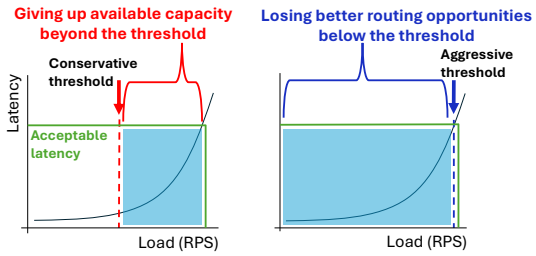


Figure 1: Limitations of static capacity-based offloading: This method misses potential load-to-latency tradeoff improvements. The left figure shows a conservative threshold, while the right depicts an aggressive one. Thresholds are marked by red and blue dashed lines. The green horizontal line indicates acceptable latency, and the blue box illustrates the missed opportunities for each threshold.

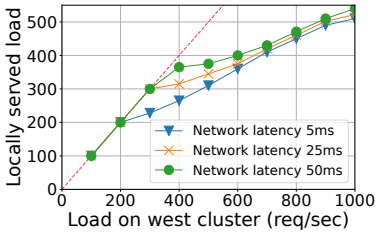


Figure 2: Empirical cross-cluster routing threshold calculated by SLATE over different network latency and loads. The red dotted line indicates 100% local serving. Right side of red dotted line means cross-cluster routing from West to East.

tion; but there is no clear rule on how it is picked. Requests beyond this capacity are offloaded to another region (cluster, in the terminology we use here). Specifically, Traffic Director uses a *Waterfall* algorithm: send to the nearest (least network latency) region with available capacity. ServiceRouter groups regions into *locality rings* and chooses a cluster in the nearest ring with available capacity; service owners configure these rings based on service-specific knowledge. These are very similar approaches depending on configuration, and the difference is not important for our purposes. In the rest of the paper, we focus on the Waterfall algorithm as a representative of these systems.

There was another work, Dealer [18] that tried to address request routing for geo-distributed multi-tier applications. For routing optimization, Dealer constructs all possible end-to-end routing paths consisting of data center choice at each service through the application. For example, FE at DC1 → BE at DC1 → DB at DC1 represents one routing path and FE at DC1 → BE at DC1 → DB at DC2 represents another path, and so on (DC: data center, FE: frontend, BE: backend, DB: database). For each path, Dealer computes the expected mean end-to-end latency. It then runs a greedy algorithm by assigning incoming traffic to these paths. It fills the lowest latency path first and continues filling the next paths to progressively higher-latency path until all the traffic is allocated. All routing decisions are made at the application entry point (e.g., the frontend service) of each data center. Based on the paper, Dealer has no coordination between data centers.

Both Traffic Director, ServiceRouter, and Dealer use manually-configured static RPS thresholds; make local, greedy routing decisions; and only consider the latency and cost implications of a single RPC hop. As we will see next, this can lead to significantly suboptimal performance.

3 Problem definition

The overarching goal of SLATE is to globally optimize the flow of requests across clusters to minimize latency and cost. In contrast, 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. In this section we outline why, by highlighting four nontrivial aspects of the problem space.

1. When the local cluster is overloaded, what portion of inbound requests should be routed away?
2. Which cluster(s) should we route those requests to?
3. Where in the application topology should requests make the cut when routing to remote clusters?
4. Which subset of requests (traffic classes) should be routed to remote clusters?

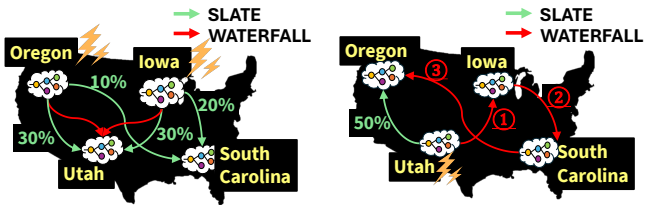
3.1 How much to route to remote clusters?

If one cluster is receiving more load than it can handle, how much load should be routed to remote regions? Figure 1 illustrates the consequences of Waterfall's use of a static threshold. In the left graph, a conservative threshold is set, aiming to distribute load early to other available clusters. However, this approach misses opportunities to fully utilize local capacity, resulting in unnecessary offloading. Offloaded traffic incurs additional network latency and egress costs, ultimately leading to higher average latency and higher expenses. In contrast, the right graph shows that an aggressive threshold might keep traffic local, even when it would be more efficient to offload that traffic to a remote cluster with lower expected latency.

The optimal threshold should be adjusted based on at least three factors: (1) the load in each cluster, (2) the inter-region network RTT, and (3) how load affects the service's latency. To illustrate it, we built a simple application consisting of three microservices; gateway, frontend, and backend, all running in K8s cluster. All three services were replicated in two different machines. Each machine was used to emulate one region and we used two regions (West and East) in this setup. Figure 2 demonstrates how the optimal routing threshold changes under different inter-cluster network latencies and load conditions. The load on East is fixed to 100 RPS while West's load varies. The optimal threshold would be even more dynamic if load in East varied as well.

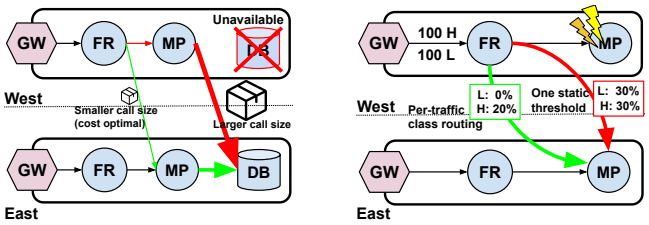
3.2 Which clusters to route to?

After choosing how much traffic to offload, where do we route it? We next show Waterfall is prone to sub-optimal routing: that is, the greedy choice (the closest currently-underutilized cluster) can be globally suboptimal. The same application from § 3.1 is used but with a real Google Cloud Platorm



(a) Latency tradeoff **(b)** Global cluster state

Figure 3: Two example use cases of SLATE for global optimization with the consideration of the network latency tradeoff. There are four clusters in different regions. In Figure 3a, Oregon and Iowa are overloaded. In Figure 3b, Utah is overloaded. The green lines indicate SLATE routing, and the red lines indicates Waterfall load balancing.



(a) Multi-hop, Egress cost **(b)** Traffic classification

Figure 4: Two example use cases of SLATE for multi-hop, egress cost, and request differentiation. The application is an anomaly detection app. *FR* is frontend, *MP* is a metrics processor running an anomaly detection algorithm, and *DB* is a datastore service storing metrics. The green lines indicates SLATE routing, and the red lines indicate Waterfall load balancing.

topology: Oregon (OR), Utah (UT), Iowa (IOW), South Carolina (SC).[1] In Figure 3a, OR and IOW are overloaded. With Waterfall (WF), they greedily offload to UT, which is the closest to both overloaded regions and *technically* has available capacity. However, due to this greedy decision, UT is running at capacity, and thus requests could incur higher average latency in UT. A latency-optimal solution would utilize the SC cluster even if it is not the closest one in proximity.

Figure 3b contains another interesting example, where some requests are pigeonholed into choosing a very distant cluster in Waterfall, which happens when there is an adversarial order of events: ① UT gets overloaded, and spills over to IOW, the nearest cluster. ② IOW gets overloaded, and spills over to SC. ③ SC gets overloaded, and chooses OR, as it is the last available cluster with capacity, even though it is also the furthest. An optimal solution would take the global knowledge of all clusters' loads into account, and UT might spill over to OR even though OR is further than IOW from UT. This optimal solution should be achieved regardless of the order of events. SLATE achieves better latency by finding the globally optimal routing solution. Figure 14a and Figure 14b in Appendix C show the latency CDF of each scenario.

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. This is well known for other matching problems [4], with somewhat different models. In Appendix A, we have formulated such a model for request routing and proved that Waterfall can (a) be exponentially worse than optimal, with certain timing of workload arrival; (b) be arbitrarily worse than optimal, with certain timing of the algorithm's events; (c) be suboptimal regardless of event ordering; and (d) exhibit *metastable failures* [20] where the algorithm is stuck in a suboptimal routing even after an initial trigger causing the problem is resolved. And there is even more complexity than a matching problem, as we will see next.

3.3 Where in the topology to route?

Most conventional load-balancing algorithms are *single-hop*, meaning the load-balancing decision for a request depends only on the state of that request's replica pool. However, in a microservice application, the effect of an early load-balancing decision can ripple through the rest of the call tree.

The application in Figure 4a pulls historical data from a database (*DB*), runs anomaly detection (*MP*), and returns the result (*FR*). $MP \rightarrow DB$ requests are ten times larger than those from $FR \rightarrow MP$ since it is fetching large amounts of metrics data from *DB*. The *DB* service in the us-west cluster is unavailable (which could occur due to security constraints, regulations like GDPR, or failure). In this scenario, Waterfall and existing service meshes (Istio) will perform locality failover load balancing, where requests cross cluster boundaries at $MP \rightarrow DB$ (red arrow in Figure 4a).

On the other hand, SLATE knows the call tree and the fact that *DB* is not running in West. Now, SLATE will route requests from West to East at $FR \rightarrow MP$, resulting in saving egress cost: \$2.21 in Waterfall vs \$0.19 in SLATE[2] per 1000 requests. Moreover, the operator can express the preference between latency and cost. If latency is the objective, SLATE will keep some requests local instead of routing all requests to East at $FR \rightarrow MP$. This achieves lower average latency by utilizing instances of *MP* in the both cluster. The corresponding latency CDF can be found in Figure 14c in Appendix C.

3.4 Which subset of requests to route?

Not all requests are the same. Requests can impose varying computation, network load, and call trees. However, conventional load balancing assumes that the inbound pool of requests is homogeneous, offloading a certain fraction of requests regardless of their type. An optimal solution should differentiate traffic classes and route them based on their characteristics.

[1] Inter-cluster VM-to-VM network RTTs are OR-UT: 30ms, UT-IOW: 20ms, IOW-SC: 35ms, OR-SC: 66ms, and OR-IOW: 37ms. The region codes in Google Cloud Provider: Oregon: us-west-1, Utah: us-west-3 , Iowa: us-central-1, South Carolina: us-east-1.

[2] The egress costs are calculated based on a standard AWS Egress rate of \$0.02 per GB between us-west-1 and us-east-1.

Feature	Waterfall	Dealer	SLATE
Multi-cluster	✓	▲	✓
Latency tradeoff	▲	▲	✓
Global view	✗	✗	✓
Egress cost	✗	✗	✓
Per-service	✓	✗	✓
Multi-hop	✗	✗	✓
Traffic classification	✗	✗	✓
Automatic adaptation	✗	✗	✓

Table 1: Comparison of Waterfall, Dealer, and SLATE. Latency tradeoff means whether the system considers tradeoff between compute latency and network latency. Global view indicates whether the system considers the load conditions of all clusters globally. Automatic adaptation refers to the ability to continuously optimize routing rules during runtime in response to real-time changes in the system. Multi-hop means whether the system considers multi-hop call tree nature of microservices or not. Traffic classification means whether the system distinguishes different classes of traffics or not. The details will be discussed later in § 4.

Figure 4b shows the effectiveness of such traffic classification. In this application, each service handles two traffic classes: Heavy (H) and Light (L), where H is significantly more compute intensive. SLATE achieves lower latency by selectively routing heavy requests to remote clusters, alleviating the local cluster effectively. However, Waterfall blindly applies a single threshold to all the requests to the service. The problem of a single threshold is that to achieve the same latency performance as SLATE it needs to set a much lower threshold, considering the worst-case scenario (all requests are of the heavy type). However, making this worst-case scenario the default requires the cluster administrator to overprovision, resulting in severe under-utilization over time. In this experiment, SLATE achieves 3.4× lower average latency (74ms) compared to Waterfall (253ms). (For the CDF, see Figure 14d in Appendix C.)

Summary: Table 1 shows what factors are considered in making routing decisions in each system. The current approach ignores many factors that affect latency and cost. We argue that a shift in viewpoint is needed, from request routing being a load balancing problem with some minor tweaks, to fundamentally a global *traffic engineering* problem. We design SLATE from this perspective.

4 SLATE

4.1 System Overview

SLATE employs a hierarchical architecture (Figure 5) with the following three key components.

Global Controller The Global Controller is the heart of SLATE's decision-making. As detailed in § 4.2, it adopts a hybrid design, combining (i) a model-based global optimizer, and (ii) an exploration based strategy (PAPA) that continually refines the routing rules generated by the optimizer.

Cluster Controller The Cluster Controller acts as an intermediary aggregation layer that addresses scaling limitations by collecting metrics from services within its cluster rather than having each service connect directly to the Global Controller. Its two key design functions are: (1) presenting a cluster-level view of load and latency (rather than individual replicas in a cluster) to the Global Controller, and (2) distributing routing rules from the Global Controller to the appropriate services within its cluster.

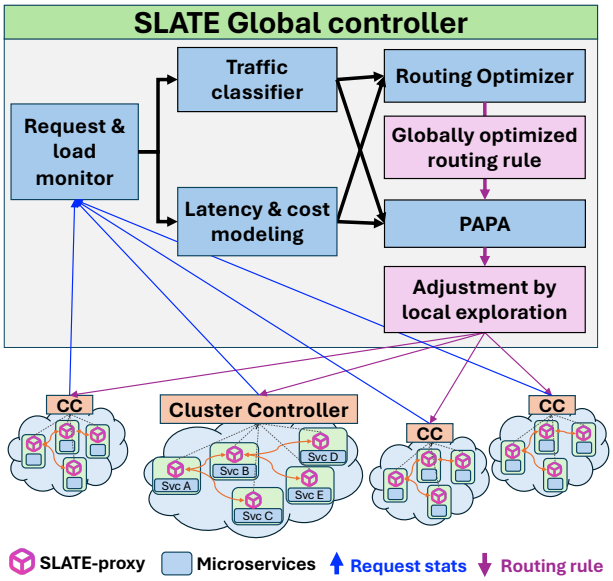


Figure 5: SLATE system architecture

Data plane SLATE-proxy functions as the data plane element, deployed as an extension to the sidecar alongside each application instance following service mesh architecture (other realizations of the data plane, like a library-based sidecar within the application process, are compatible with our design). It performs two critical functions: (1) collecting fine-grained, per-request telemetry including load metrics, request details, latency, and trace information; and (2) enforcing request routing policies for each traffic class based on directives from the Global Controller. The proxy's key design distinction is its per-request monitoring granularity, enabling precise load-to-latency mapping for each traffic class.

4.2 Global Controller

The Global Controller combines multiple components and techniques to feed both the core routing optimizer and the subsequent PAPA adjustments.

L7 Traffic Classification As shown in § 3.3 and § 3.4, request routing with traffic classification can significantly improve latency and cost. In SLATE, a traffic class (TC) is defined as the set of requests that match some arbitrary set of observable characteristics on a request, and requests in the same traffic class should exhibit similar resource and latency profiles. The simplest approach is to treat all requests homogeneously as a single TC (as Waterfall does), but this will miss large opportunities for optimization (Figure 4b, Figure 14d). At the

other end of the spectrum, an extremely large number of TCs could more accurately characterize traffic, but makes it hard to get enough samples of each class to predict its behavior, and worsens performance of the optimizer. We therefore seek a modest number of TCs, defined so that requests within each TC are as homogeneous as possible (in terms of child call tree and resource consumption).

Note that a TC is not expected to be perfectly homogeneous, but the more we are able to divide requests into distinct behaviors, the better SLATE will find ways of optimizing routes. In our current implementation, we define TCs by the combination of three features: *(1) destination service*, *(2) HTTP Method*, and *(3) HTTP Path*. This will capture differing behaviors across API endpoints. Traffic classes are initially determined during offline profiling and can be updated at runtime using online data. SLATE can perform traffic classification periodically together with continuous model update during operation. Optimizing traffic classification might further improve SLATE's performance and is an interesting area of future work.

Distributed tracing SLATE has its own distributed tracing component residing on Global Controller. It collects the per-proxy spans including parent-child dependencies, its start and end time and request information which is used for traffic classification. It reconstructs the end-to-end call graph, and derives each service's *exclusive time*—the response time minus the durations of those child RPCs. Exclusive times enable accurate load-to-latency modeling and are summed across the call graph to compute an approximate end-to-end latency. Call graphs can be dynamic due to conditional logic in applications. SLATE relies on probabilistic request flows based on historical observation (e.g., service A calls B with 40% and calls C with 60%). End-to-end latency can be calculated as the weighted sum of exclusive times across observed call paths.

Data-driven latency modeling Our optimizer requires a per-traffic-class latency model for every service. We approximate each service as a modified version of M/M/1 queue—capturing the common thread-pool execution pattern where requests either grab an idle thread or queue behind busy threads.[3] We fit $L(\mu) = \frac{a}{1-\mu} + b$ to the measured exclusive-time vs. utilization data (where utilization μ is RPS normalized by peak throughput). Here a (latency sensitivity) and b (base latency) are learned separately for each TC.

Predicting latency for a given load turns out to be one of the most subtle problems for service layer TE because of the complexity of application behavior. Almost no application will perfectly match an M/M/1 queue. One could certainly adopt a more sophisticated latency model than M/M/1, but as we will discuss in more detail later, applications can have behaviors that would be extremely difficult to model(§ 5.3.2). We intentionally use a simple model: SLATE's overall approach is to use a lightweight approximation, so the optimizer remains scalable; then, our separate PAPA mechanism will correct modeling errors empirically (§ 4.4). As we will see, this allows SLATE to perform well even if there are model discrepancies.

Normalized RPS In a real deployment, a single service instance processes multiple traffic classes at once, so the latency of one class depends on the combined work of all classes. Accurately capturing that would require profiling latency for every possible combination of per-class loads—a space that grows exponentially with the number of classes. Instead, we convert each class's load into a common "cost unit" by scaling it relative to a baseline class (e.g. if a heavy request uses twice the CPU of a light one, it counts as two cost units). Summing these cost-weighted rates yields a single "normalized RPS" that approximates total utilization under the assumption that requests are mutually "tradeable" by their relative expense. This one-dimensional metric drives our M/M/1 model with minimal profiling. Multi-class interaction error is then corrected by PAPA (§ 4.4).

Continuous Model Update For SLATE to remain accurate under changing conditions, the Global Controller continually ingests fresh latency vs. load measurements from SLATE-proxy through Cluster Controller. But at any moment it only sees the currently operating load. Blindly replacing old points with new can distort the model (e.g. a spike in latency at low load might be misinterpreted as relatively lower latency at higher load). Instead, we use a two-stage approach with the notion of "buckets" partitioning the utilization range (0–10%,10–20%, ... ,90–100%). The first stage is *Shift phase (partial observations)*: If only some buckets have new data, we adjust the existing model by a vertical shift equal to the average prediction error on the new buckets. This preserves the learned shape while accommodating a small global offset. The second stage is *Retrain phase (comprehensive observations)*: Once every bucket is filled with fresh measurements, we perform a full curve fit across all data and reset the old samples.

This scheme ensures fast update to localized changes without overfitting to a narrow operating point, yet still converges to an up-to-date model once comprehensive data arrives. While this approach can work well in practice, the system may not naturally experience all load levels needed for retraining during runtime. Active exploration of different load conditions is left as future work. Currently, SLATE's continuous profiling collects data passively.

4.3 Request routing optimizer

All the aforementioned data is inputted into the request routing optimizer of SLATE: call trees with TCs differentiated, latency model of each TC, placement of services, inter-cluster network latency, the current per-TC load metrics and the user's preference in egress cost ($\$/ms$). The request routing opti-

[3] An M/M/d model (with d threads per service) is more precise, but its Erlang-C term increases solver time by $O(d)$; in practice M/M/1 with learned parameters approximately matches observed behavior.

mizer minimizes the average end-to-end request latency (including both compute and network latency across all requests) and total egress cost based on the administrator's preference. This optimization is modeled at the granularity of TCs, which refers to an L7 traffic class at a specific service in a specific cluster (but representing the aggregate across replicas of the service within that cluster). We then jointly optimize across TCs. The objective function is:

$$\min_{RF} ((T\_CT + T\_NT) * \$/ms + T\_EC) \quad (1)$$

where T_CT, T_NT, and T_EC are the expected total compute time, network time and egress cost, respectively. It minimizes over *RF* (for "request flow": a vector representing, for each TC in each cluster, the rate of requests routed to each cluster). The solution output is: the optimized request routing in between all connected pairs of edges in the given call trees for all TCs between clusters. In general, there will be a tradeoff between latency and cost; we let users express their preference of how to control this tradeoff in the form of $\$/ms$: the amount of money an administrator is willing to pay per millisecond of average latency saved.

The total expected compute time (T_CT) is represented as:

$$\sum_{c \in C,\ s \in S_c,\ t \in T_s} CL_{c,s,t} \cdot LatencyModel_{s,t}(NormLoad_{c,s,t}) \quad (2)$$

Here, C, S_c, and T_s represent the set of clusters, the set of services at a given cluster, and the set of traffic classes for a given service, respectively. $CL_{c,s,t}$ represents the compute load (RPS) for a given traffic class t belonging to service s in cluster c. $LatencyModel_{s,t}$ represents the profiled load-to-latency function for traffic class t at service s. Total expected network time and egress cost are formulated in a similar manner with inter-cluster network latency and egress. The full formulation can be found in Appendix E.

The above described the optimizer's objective. We now discuss the constraints. Similar to network-layer TE, SLATE's constraints can be thought of in two broad categories: (1) satisfying demand and flow conservation, and (2) resource availability. First, we need flow conservation constraints that ensure all incoming requests are served and generate appropriate upstream requests. Unlike network-layer TE where incoming flow simply equals outgoing flow, in SLATE a request of one traffic class can generate multiple scaled requests to other traffic classes, following the observed service invocation trees. Second, we need infrastructure constraints incorporating two elements: (1) service placement variables that reflect real-time deployment locations based on heartbeat signals, automatically updating when services fail, degrade, or are deployed; and (2) valid routing links between traffic classes. While we assume sufficient cloud bandwidth (no link capacity limits), determining valid links is nontrivial since we are dealing with an overlay network where any TC could potentially call any other. To manage scale, SLATE only creates variables for links observed in call trees, yet this still generates n^2 potential links across n clusters. We further optimize by pruning links that would route traffic from underloaded to overloaded clusters when upstream services are fully replicated in multiple clusters.

The output generated by the optimizer can be expressed as a set of ***rulesets*** (a term we'll use for the rest of the paper). A *ruleset* denotes the set of outbound routing rules calculated by the optimizer for a given traffic class and cluster. For example, the form of a ruleset for some traffic class T at some service S in the *east* cluster might be $\{east : 50\%, central : 30\%, south : 15\%, west : 5\%\}$. The routing optimizer keeps running during runtime continuously with real-time load information and cluster state. The reaction time and scalability of the routing optimizer will be discussed later in § 5.4. Fault tolerance of the routing optimizer can be achieved by simply having a backup Global Controller instance.

4.4 Performance-Aware Policy Adjustment

The optimizer calculates a global, theoretically optimal solution. However, what if the models that the optimizer uses are not accurate? Model inaccuracy can occur by two broad reasons: (1) when data used to build the model become stale or (2) when the model itself does not fully express the application latency profile due to its limited expressibility. Both will produce sub-optimal results. PAPA (Performance-Aware Policy Adjustment) enhances SLATE's resilience by compensating for potential inaccuracies through a novel exploration-based routing optimization algorithm working in tandem with the global optimizer.

To make the search space for the exploration more tractable, PAPA essentially relies on the rulesets generated by the global optimizer, tweaking them when the observed performance deviates from the model's expectations. More specifically, PAPA incrementally alters one *ruleset* at a time, keeping the updated rules in place if they yield better objective values, and rolling then back if they do not. Even within a single ruleset, the space of possible alterations is large—for a ruleset with x destinations, there are 2^x possible directional adjustments. As a heuristic to guide the search, PAPA shifts weight from clusters whose observed performance falls below the optimizer's prediction toward those that exceed expectations. To decide which rulesets to adjust first, PAPA ranks them by the magnitude of the gap between observed and predicted performance.

4.4.1 Motivating PAPA

Stale data inaccuracy Sudden changes in system state (e.g. due to a noisy neighbor [42], network congestion, or changes in allocated resources) may cause the application or network latency to be different from the models in the optimizer. Continually updating the model with current data may help, but for the optimizer to be effective, the models need to be trained on a comprehensive set of load conditions (which is not guaranteed at runtime). However, PAPA can react to these state

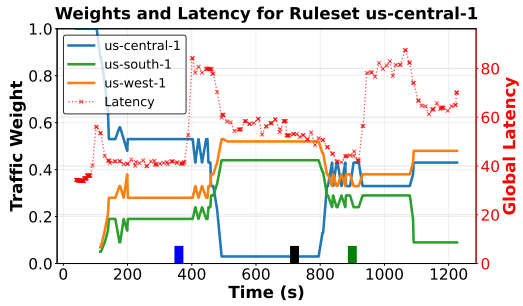


Figure 6: This demonstrates PAPA's ability to dynamically adjust routing weights in response to latency faults in different regions. At t=390 (blue marker): 150ms latency fault injected into us-central-1 makes us-central-1 under-performer and us-west-1 and us-south-1 over-performer. PAPA shift routing weight from us-central-1 toward us-west-1 and us-south-1, successfully reducing global latency (red dotted line). us-east-1 was running at this experiment but it does not appear in this experiment since us-east-1 did not exist in us-central-1's ruleset. At t=720 (black marker): The fault is removed, and routing returns to near-original patterns. At t=920 (green marker): new 150ms fault is injected into us-south-1, causing PAPA to gradually shift weight away from the affected region, effectively decreasing global latency. Between t=850-900: PAPA experiences oscillations and converges.

changes by proactively looking for which traffic classes' performance differs the most from the current model expectation, and iteratively changing routing rules accordingly. Figure 6 demonstrates how PAPA can dynamically change routing rules in response to the model being out-of-date due to an injected fault. The global average latency improves by 1.7× (from 88.38ms to 51.58ms), and 3× and 1.12× for p90 and p99 respectively.

Limited expressiveness of latency modeling PAPA can detect and correct latency mispredictions at runtime, that may arise from several factors:

(1) Load-to-latency model deviates from M/M/1: The true load-to-latency relationship may follow a shape that differs from our M/M/1 approximation —service times may not be exponential, and effects like batching behavior can cause sharp transitions or plateaus that can deviate from the M/M/1 curve. While a more flexible model (e.g., a fine-grained piecewise linear regression) might better capture these effects, we adopt M/M/1 as a tractable and broadly reasonable approximation.

(2) Dynamic application behavior: Latency also reflects dynamic aspects of application behavior that are not captured by our load-based model. For instance, different regions may have different items in their cache, causing certain request subsets to experience faster response times; we examine this effect, and how PAPA helps in such scenarios, in § 5.3.2.

(3) Interaction between traffic classes: A particularly challenging case arises when multiple traffic classes share the same microservice instance. To model the effect of such colocation, we perform normalization (see § 4.2), assuming that the resource usage of different traffic classes can be expressed as a fixed ratio. This approximation allows the optimizer to reason about aggregate service utilization, but it can break down when traffic classes interact, e.g., due to inter-traffic class cache effects or because they depend on different resource types. We explore this in more detail in Appendix C.3.

4.4.2 PAPA Algorithm Design

Due to the combinatorial growth of possible routing rules with system size and the overhead of performance measurements, PAPA employs two key constraints to prune the search space for its exploration: (1) only adjusting weights of existing rules (where current weight is > 0), and (2) using expected latency to guide search within a ruleset. PAPA operates within individual rulesets: in each step it selects a ruleset and shifts weights within that ruleset.

Given that the starting point of PAPA is the theoretically optimal set of rulesets generated by the optimizer (and its models), measuring the difference between expected and actual latency for every cluster gives PAPA a notion of how clusters are actually performing, relative to this expectation. PAPA then attempts to mend each ruleset by shifting traffic towards clusters relatively overperforming model expectations, and away from clusters underperforming model expectations. This adaptive shifting helps correct systemic biases in the optimizer's routing rules (caused by inaccurate models), naturally steering the system toward rules that more accurately reflect real-world performance, with each step of PAPA empirically verified by observing real-time performance improvements.

Specifically, for each ruleset, PAPA:

1. Calculates how clusters deviate from the Optimizer's predictions, weighted by volume of affected requests. Specifically we calculate a *performance discrepancy* $d_{c,t}$ for each cluster C and traffic class T as the $(expected - observed)$ latency × RPS.
2. Calculates the average performance discrepancy $\bar{d}$ of the ruleset, over the destination clusters and traffic class of the ruleset. These destinations are partitioned into overperformers (O; those with $d_{c,t} \geq \bar{d}$) and underperformers (U; those with $d_{c,t} < \bar{d}$). For convenience, let

$$S_U = \sum_{(c,t)\in U} (d_{c,t} - \bar{d}) \quad \text{and} \quad S_O = \sum_{(c,t)\in O} (\bar{d} - d_{c,t}).$$

Here, S_U is the total amount by which under-performers exceed the mean discrepancy, and S_O is the total amount by which over-performers fall below it.

3. Shift each destination cluster c's weight ($w_{c,t}$) by distributing γ from under-performers (U) to over-performers (O):

$$w'_{c,t} = \begin{cases} w_{c,t} - \gamma \dfrac{d_{c,t} - \bar{d}}{S_U} & (c,t) \in U, \\ w_{c,t} + \gamma \dfrac{\bar{d} - d_{c,t}}{S_O} & (c,t) \in O, \end{cases}$$

4. Measures global performance for a duration τ and keeps the change only if performance improves.

5. Marks a ruleset as converged after θ consecutive failed attempts to improve the latency. Note that on each attempt, the attempted routing weights are the same; the reason to try more than once is due to measurement noise in step 4.

To maximize impact, PAPA explores rulesets in order of potential improvement, defined by the variance in destination cluster performances. The algorithm continues until all rulesets converge, then pauses until either: (1) the Optimizer has new rules (any weight changes $> \delta_{rule}$), or (2) Any traffic class's performance changes significantly ($> \delta_{perf}$). If the optimizer has new rules, PAPA will restart exploration from those rules, whereas if performance changes, PAPA will resume exploration from the rules currently in place. The complete algorithm is provided in Appendix F.

The parameters γ (step size), τ (step duration), θ (convergence attempt threshold), δ_{perf} and δ_{rule} (conditions that retrigger PAPA) can be tuned based on application latency characteristics to make PAPA more effective. These parameters govern PAPA's exploration vs. precision trade-off. Step size (γ): Larger values enable faster adaptation to performance changes but risk overshooting optimal solutions. Observation period (τ): Longer periods reduce noise in latency measurement but delay reaction time. For applications with high latency variance, increase both γ and τ. Note that latency as a metric has noise by nature. δ_{rule} (optimizer restart threshold) should not be too small to prevent the Optimizer from starving PAPA. In our experiments, we found $\gamma = 0.05$, $\tau = 10s$, $\theta = 3$, $\delta_{perf} = 10\%$, $\delta_{rule} = 5\%$ to be reasonable defaults. Automatic parameter adaptation based on application latency characteristics is an interesting area of future work.

PAPA uses the Optimizer output rules as a starting point, and deviation from the model's expected latency to guide its exploration. Without this guide, if just observed latency dictated the direction PAPA explores, PAPA would become a greedy algorithm akin to work stealing (continually attempting to shift load to the fastest cluster), and would take much longer to find a good routing. We acknowledge the limitation that the convergence in PAPA is not guaranteed but remains best-effort. High latency variance makes it difficult for PAPA to detect genuine performance improvements. In the worst case, PAPA converges to the Optimizer's initial solution without further improvement with the safeguard logic by only keeping changes that improve latency. The noise can be reduced by improving traffic classification to reduce within-class latency variance.

4.5 SLATE-proxy

SLATE-proxy is the data plane of SLATE. It has two core functionalities; collecting telemetry and enforcing routing rules at the critical path. Collecting the type of telemetry SLATE needs is challenging due to the very fine per-trace granularity needed, integrating with the threading model of throughput-oriented L7 proxies, while also keeping the performance of the data plane high. Routing decisions are enforced for outbound requests in the form of match rules for each traffic class. See Appendix D for more details outlining why this is nontrivial and how SLATE-proxy solves these problems.

5 Evaluation

We evaluate how SLATE contributes to performance across diverse scenarios, focusing on the following questions:
(1) Does the optimizer improve routing by capturing traffic classes, latency trade-offs, and multi-hop paths, when compared to state-of-the-art baselines? — § 5.2.
(2) How effectively does our hybrid design—combining centralized optimization with local adaptation—address challenges of modeling a dynamic, complex system? What is the significance of different aspects of SLATE's design (global optimization and PAPA) — § 5.3.
(3) Is SLATE efficient and scalable in practice? — § 5.4.

5.1 Experimental setup

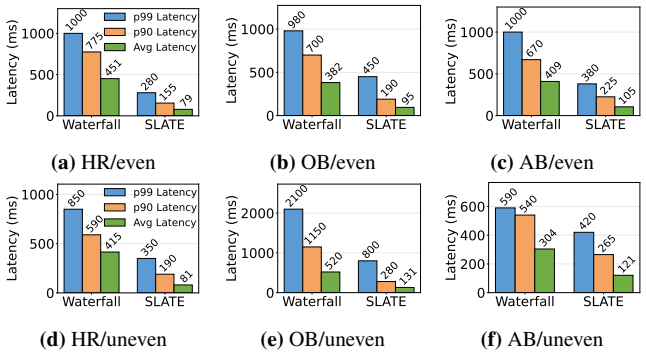
In this subsection, we describe the setup of our experiments, including the benchmarks we used, the multi-cluster setup, and container resource allocation strategy.

Implementation The Global Controller is implemented in 8.3K lines of Python, which runs as a standalone service. SLATE-proxy is implemented with 2.5K lines of Golang conforming to the proxy-wasm spec [12], which allows the plugin to be compiled into an efficient WebAssembly binary. This also allows for SLATE-proxy to be seamlessly plugged into most of the popular L7 proxies (Envoy [8], NGINX [35], ATS [19], etc) and all data plane functionality should work out-of-the-box. We benchmark with Envoy [8]. SLATE is open-sourced. The code can be found in https://github.com/ServiceLayerNetworking/SLATE

Benchmark Applications We evaluate SLATE using three benchmark applications—Hotel Reservation(**HR**), Online Boutique(**OB**), and a reconstructed Alibaba service graph(**AB**)—as well as four microbenchmark applications that exercise specific aspects of the request routing problem (from § 3). Hotel Reservation and Online Boutique consist of 10 and 11 services, respectively, each with four traffic classes and call trees ranging from 2–8 services. See the Online Boutique topology in Figure 19 (Appendix C.6).

To evaluate scalability, we reconstruct a synthetic application from a subset of the Alibaba trace [28], We extract a call tree with 20 services consisting of eight stateless, seven cache, and five database services. Since these traces do not contain the application code, we emulate realistic behavior by assigning matrix multiplication to microservices and varying levels of disk I/O to caches and databases. The workload includes two traffic classes (implemented as two different API endpoints), where one class consumes approximately 10× the resources of the other.

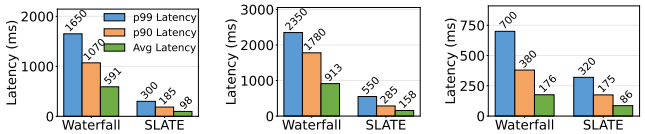
Multi-cluster Setup We deploy benchmarks on a multi-node Kubernetes cluster using up to 12 machines (480 cores) in CloudLab. Each node has either an Intel E5-2660v2 or



(a) HR/even **(b)** OB/even **(c)** AB/even

(d) HR/uneven **(e)** OB/uneven **(f)** AB/uneven

Figure 7: Latency CDFs for all benchmarks under complete replication. HR, OB, and AB represent Hotel Reservation, Online Boutique, and Alibaba benchmarks, respectively.



(a) *cart* **(b)** *checkout* **(c)** *pay, email, ship*

Figure 8: Latency CDFs for the partial replication scenarios. The captions in each subfigure indicate the service(s) not available in UT cluster.

E5-2630v3 CPU and 256GB of memory; all nodes are dual-socket NUMA. The SLATE Global Controller and workload generator run on dedicated nodes. To emulate a realistic multi-region cloud environment, we partition the machines into four regions based on the topology of Google Cloud Platform (Oregon, Utah, Iowa, and South Carolina). We inject inter-region network latencies corresponding to GCP measurements using Linux's `tc` tool.

Baseline. As a baseline, we use the Waterfall algorithm which represents the most advanced global load balancing approach to our knowledge (§ 2). Because there is no official or open-source implementation[4], we reimplement Waterfall in the SLATE Global Controller. We use the Traffic Director variant (without "rings"; § 3). Waterfall's threshold equals the capacity assuming every request is the heaviest class—a worst-case bound meant to protect the system from any traffic class distribution—but any static threshold is far from optimal.

Workload. We generate requests using the wrk2 [38] and vegeta [1] HTTP load generator. These requests are for all of the application's API endpoints (traffic classes). We focus on scenarios where a subset of clusters are overloaded because they reveal routing optimization opportunities. Under normal load where all clusters have enough capacity, the optimal routing rule is 100% local routing for all systems. We vary the overload factors across experiments between 2-5× load imbalance between overloaded and non-overloaded clusters. The subset and the exact extent of workload imbalance vary across experiments to ensure the evaluation captures a broad range of realistic cases.

5.2 Optimizer Performance on Benchmarks

We highlight the impact of our global optimizer—without PAPA or continuous profiling—on three end-to-end benchmarks under two deployment scenarios (complete vs. partial replication) in the aforementioned four region GCP topology. We evaluated under two different load imbalance workloads, even and uneven. Even workloads assign equal load to every traffic class; uneven workloads skew one class to 3× the load of others. We then overload Oregon at 3× its capacity (others remain underloaded), keeping total load below the aggregate capacity of all clusters.

5.2.1 Complete replication

Figure 7 shows latency CDFs when every service is replicated in all regions and the OR cluster is overloaded. In each benchmark, SLATE cuts both mean and tail latency compared to Waterfall by offloading just the right traffic classes to the best remote clusters. In **Hotel Reservation**, when OR is saturated by the heavy *search* class (Figure 7d), Waterfall blindly spills all classes evenly to UT, IOW, and SC, incurring high latency. In contrast, SLATE selectively offloads mostly *search* and *recommendation* requests—accounting for tradeoff between inter-cluster RTT and load—yielding a much lower latency CDF. In **Online Boutique**, under uneven load on *addtocart* and *checkoutcart* (Figure 7e), Waterfall begins offloading every class too early, paying unnecessary network and egress cost even for the traffic class that does not help alleviating overload (*addtocart*)—and in some cases performing worse than local-only routing. SLATE, however, adapts routing rules per class and cluster state to minimize end-to-end latency. **Alibaba**, we observe the same pattern for AB: Waterfall's static threshold either over-offloads or under-offloads, while SLATE's global solution consistently achieves lower latency.

5.2.2 Partial Replication

Partial replication refers to deployments where certain services are only available in a subset of clusters. This strategy is common as indicated by our survey of practitioners in industry (Appendix B). Multi-hop routing and traffic classification become particularly critical here, as early knowledge of unavailable downstream services with call tree information enables planning better routing decisions. Existing load balancers, including Waterfall, rely on locality failover, which only routes requests at the last possible hop.

Partial replication was evaluated on the Online Boutique benchmark under three partial-replication scenarios: degrading (1) the cart service, (2) the checkout service, and (3) the payment, email, and shipping services in the Utah cluster.

[4]SLATE 's data plane is built on top of open-source Envoy and cannot run on Traffic Director's proprietary data plane, making direct comparison infeasible. Additionally, Traffic Director's publicly described algorithms (from 2019) may no longer reflect current implementations. To ensure a scientifically meaningful comparison, we evaluate against a clearly defined algorithm rather than an opaque production system.

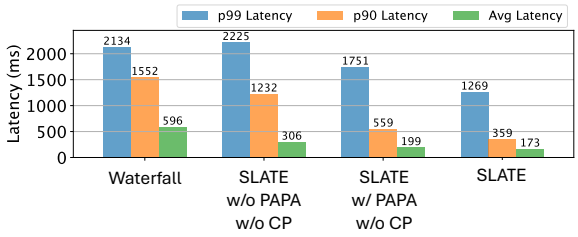


Figure 9: Latency statistics for SLATE with and without PAPA and continuous profiling (CP), and Waterfall for Online Boutique application with events causing stale data. Without continuous profiling, the latency model is not updated during runtime. Resource allocated to one service (checkout service) deployed in the Utah cluster was reduced from no-limit on CPU resource to 300m core per K8s pod instance in the middle of experiment.

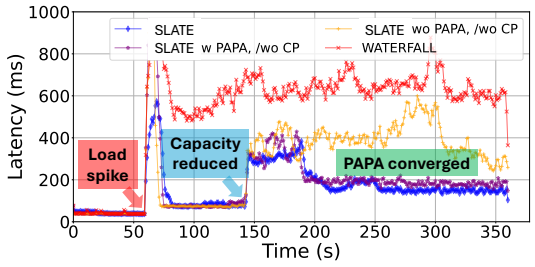


Figure 10: Time series of average latency in the capacity reduction scenario of Figure 9. Load change introduced at time $t = 50$ is bursty. The capacity for checkoutservice in Utah cluster was reduced at time $t = 140$. SLATE quickly adapts to the new load condition and also to new capacity, outperforming Waterfall and SLATE without PAPA or continuous profiling.

Figure 8 shows latency CDFs for these cases. By exploiting multi-hop routing, SLATE outperforms Waterfall in every scenario. In Appendix C.2, we describe the detailed analysis of what routing optimization SLATE made for the case (3) with full routing snapshot of both SLATE and Waterfall (Figure 15).

5.3 SLATE's Adaptivity

While the global optimizer alone provides strong baseline performance, it fundamentally relies on accurate, up-to-date latency models. We now evaluate how SLATE addresses model inaccuracies caused by stale data or inherent modeling limitations. Specifically, we measure how effectively these components adapt to unusual high network latency and changes in resource availability (stale data) and limitations of latency-model expressiveness.

5.3.1 Stale data

Real deployments commonly face sudden resource changes or network hiccups. We inject two faults: (a) a capacity drop in the Utah cluster (Figure 9) and (b) an added 300 ms delay into Oregon's network path (Figure 12b). Egress-cost effects appear in Appendix C.9. Figure 10 plots end-to-end average latency over time. Before the Utah capacity cut at $t = 140$,

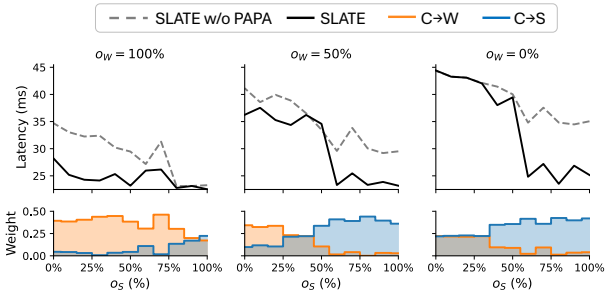


Figure 11: The x-axis for each graph is o_s, the requested key overlap percentage between south and central. Top: average latency at the end of each run. Bottom: the routing weight for Central to West ($C \to W$) and Central to South ($C \to S$).

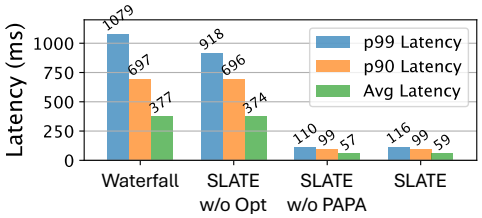
the optimizer alone (no PAPA, yellow) matches full SLATE (blue). After the cut, its stale model produces poor routing. PAPA immediately begins tweaking rules, converging in 60 s (by $t = 200$) to much lower latency, exploring 2 rulesets. Adding CP further refines the model and yields a small extra gain. In contrast, the static Waterfall baseline stays at high latency. A similar adaptive recovery occurs under injected network delay (Appendix C.8).

5.3.2 Limitation of model expressibility

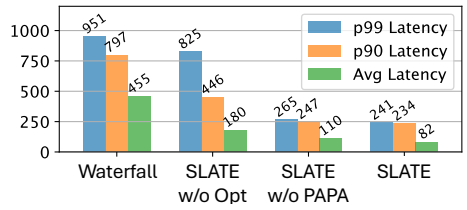
PAPA is not limited to correcting stale models–because it is based on observing actual performance, it can compensate for any systematic mismatch between our M/M/1-based optimizer and reality, whether due to caching effects, workload-dependent code paths, or other unmodeled factors. We illustrate this with a cache-locality scenario, where the workload dependent effects of an LRU cache are not captured by the latency model. Specifically, offloading can inflate a remote region's cache working set—burdening its LRU beyond capacity and spiking misses in ways an M/M/1 model misses.

Setup We deploy an in-memory cache service (akin to memcached) on four clusters (Central, West, South, East). Each instance has an in-memory LRU cache (size = 100 records). In the case of a cache hit, a response is immediately returned. In case of a cache miss, we use a delay (exponential distribution with mean 75ms) to simulate loading the object from back-end storage. The Central cluster is overloaded and the optimizer initially offloads 50% of Central's requests equally to West and South (East is at capacity).

Working Set Overlap We can expect that the distribution of working set (i.e., commonly queried items) will differ for the four clusters due to geographic user population or other factors. To emulate this, for each remote cluster $X \in West, South$, we parameterize the set of requested objects directed to that cluster by an overlap fraction $o_X \in [0,1]$. Cluster X uniformly issues requests over a contiguous range of 100 keys whose intersection with Central's cached key range $(0-99)$, is $100 \cdot o_X$ keys. For example, $o_W = 0.5$ will correspond to West requesting keys in range 50–149, and these keys get loaded into West's cache. When o_X is higher, both requests originating

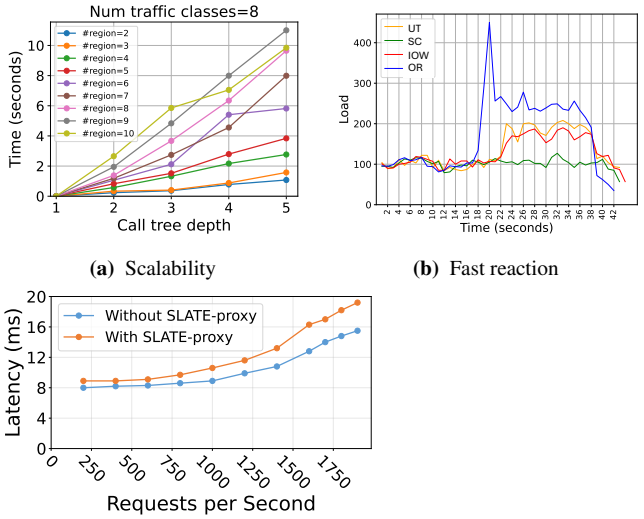


(a) No fault injected



(b) Fault injected to Oregon

Figure 12: Ablation study that shows why SLATE needs both the global optimizer and PAPA as design choice.



(a) Scalability **(b)** Fast reaction

(c) Data plane overhead

Figure 13: SLATE's scalability, and reaction time (a) Scalability for different number of regions (clusters) and depths of application call tree. The fan-out degree is three[5]. Note that this only includes the optimizer solver time. (b) How quickly SLATE is able to react against bursty load. The y-axis is the load in each cluster. The Oregon cluster received bursty load at time $t = 20$ and the new rule was executed at time $t = 23$. Note this includes the end-to-end reaction from detecting the load change, solving the optimization problem, and finally installed the new rules in the SLATE-proxy. (c) shows the latency overhead introduced by SLATE-proxy in data plane. Note that it was measured with a single replica.

locally from cluster X and requests offloaded from Central have a higher probability of hitting cache. We sweep o_W and o_S independently in 0.0–1.0 (step 0.1). For each (o_W, o_S) pair we record three quantities: (1) the average latency under the optimizer-only solution, (2) the average latency after PAPA converges, and (3) the converged routing split from Central to West vs. South after PAPA converges.

Results Figure 11 shows representative results for $o_W \in \{0\%, 50\%, 100\%\}$, with varying o_S. When $o_S \gg o_W$, PAPA shifts more offloaded traffic from West to South, lowering latency; when $o_W \gg o_S$, it shifts toward West. The full heatmaps in Figure 18 (Appendix C) confirm this behavior across the entire (o_W, o_S) plane. Without PAPA, routing weights remain at the optimizer's static 50/50 offload split and suffer higher latency under skewed overlaps. Thus, PAPA dynamically adapts to this unmodeled performance variation by exploring and retaining locally-beneficial rule adjustments. The limitation is that PAPA's exploration can take time to converge, or might have a difficult time searching among the options. We will see this demonstrated next.

5.3.3 Ablation study for SLATE design choices

We conducted an ablation study to evaluate SLATE's design choice of combining the global optimizer with local exploration (PAPA). We tested Online Boutique on an overloaded Oregon cluster (3× capacity) under two conditions: without fault injection (accurate latency models) and with fault injection (inaccurate models due to increased network latency). The PAPA-only (SLATE w/o Opt) solution needs a starting point (because the Optimizer doesn't exist to provide one), so we hardcode a starting point as mostly local routing, and small percentages of traffic directed to every other remote cluster. This gives PAPA a set of rulesets to observe and adjust. In Figure 12a, when latency models are accurate (no fault injection), PAPA alone provides no performance improvement after it starts the exploration since expected and observed latencies match. However, with the global optimizer, SLATE reduces average and tail latencies by 6.3× and 7.9× compared to SLATE w/o Opt. Under fault injection, PAPA-only solutions can detect the discrepancy between expected and actual latencies, improving routing compared to the no-fault scenario. However, this still underperforms compared to SLATE w/o PAPA and the full SLATE. The full SLATE system performs best by combining both approaches: the optimizer provides a starting point, while PAPA efficiently explores from there using the discrepancy between model-predicted and actual latency as a useful hint of which direction it should explore without trying all possible combinations, in this case finding a routing that has 25% lower latency than without PAPA.

5.4 Scalability, and Fast reaction

Optimizer scalability Figure 13a plots solver runtime as we vary the number of clusters, traffic classes, and call-graph depth. At tens of clusters, eight traffic classes, and multi-hop trees, the global optimizer completes in seconds—fast enough for per-application control loops for moderate-size applications. Note that the required scale is limited because (a) each SLATE instance serves a single application deployment and

[5]This is the mean number of services at each hop in the Alibaba microservices traces [28].

(b) SLATE only needs to model at the granularity of regions, not hosts or individual service replicas. As a result, our current implementation is already useful for small to moderate scale application deployments. However, further optimization is needed for large scale applications. Note that in network-layer TE, early cloud WAN TE optimizers performed relatively poorly, and a long string of work has since improved performance (e.g. [31, 32, 41]). We believe many of these techniques could be adapted for service layer TE, but leave this to future work.

End-to-end reaction time and data-plane overhead Figure 13b illustrates end-to-end reaction when the Oregon cluster experiences a sudden load spike. From the moment updated load arrives, SLATE collects metrics, solves the optimization, and installs new rules into each SLATE-proxy in just under three seconds. Figure 13c shows the data plane overhead evaluation results. SLATE-proxy incurs just 1-3 ms of added latency per request even at 2000 RPS, ensuring that fast, global reconfiguration does not come at the cost of per-request performance. There were engineering challenges associated with achieving a high performance data plane due to modern proxy's multi-threaded architecture where a pair of request and response might not be scheduled at the same thread necessarily. We address it by implementing a high-performance ring buffer to precisely measure load in WebAssembly, which is further described in Appendix D.

6 Discussion

Arbitrary traffic classification How to segment traffic (requests) into classes is a design choice for SLATE. Deriving right traffic classes heavily depends on the application behavior, along with the set of request attributes available to differentiate them. As interesting future research, clustering or more advanced machine learning techniques may help derive a small yet specific set of classes, which could be identified by any of a variety of request features, including service name, method, URL, request header, query parameter.

Profiling & modeling Although SLATE continually collects data from the system and updates each model accordingly, the system may not experience diverse load levels to build a complete model during the recent operating time. One possible technique to solve this problem during runtime which we did not explore is to leverage the fact that each service has multiple replicas. We can use a "siphon" replica in the load balancing pool, and deliberately direct varying amounts of load for each traffic class to better build a profile.

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

7 Related Work

Service Layer Routing/Load balancing We have already discussed Traffic Director [13] and ServiceRouter [36] in detail. C3 [37] and Prequal [40] are service layer load balancers relying on active server probing, and focus on a single data center. They could be combined with SLATE for a final load balancing decision within a cluster after SLATE optimizes cross-cluster routing. Another work (MCOSS) [3] optimizes request routing using a linear program. None of these works consider network latency, multi-hop topologies, or traffic class differentiation. In addition, MCOSS [3] is implemented through controlling DNS which would be less effective than our Envoy-based approach in terms of performance, ease of deployment in modern cloud-native architecture, or debuggability.

A recent workshop paper [26] proposed the idea of service layer traffic engineering, used a simple design to show that there are opportunities, and discussed relevant challenges. Our present paper contributes a full system design including continuous model update and PAPA, a higher performance implementation, a full experimental evaluation (with realistic benchmark applications, scale tests, partial replication and capacity reduction scenarios), and theoretical analysis of problems with the Waterfall algorithm (Appendix A).

Cluster scheduling (Autoscaler) Kubernetes includes autoscalers, and there is extensive research on provisioning jobs using more advanced methods [17, 30, 34, 44]. As discussed already (§ 2), autoscaling is a different problem. In general, cluster scheduling and SLATE are complementary. Jointly optimizing resources at *both* timescales/granularities would be valuable future work.

Geo-distributed clusters Iridium [33] is a geo-distributed analytics framework with awareness of wide-area bandwidth and its effect on data transfer time and cost. However, it does not consider latency, operates in slower periodic global optimizations on recurring jobs, and inherits the other differences above. Skyplane [24] tackles a complementary problem of finding optimized routes for cross-cloud bulk data transfers using overlays. Shard Manager [25] is targeting geo-distributed applications but again it is addressing a different problem of shard placement.

8 Conclusion

Microservice architectures effectively create a network of requests, which we argue leads to a traffic engineering problem at the service layer. SLATE addresses this emerging need, addressing new challenges of capturing traffic classes and dealing with the difficulty of modeling accurately, showing significant improvements to application latency and cost.

Acknowledgements

We thank our shepherd, Shixiong Qi, and the anonymous reviewers for their constructive feedback. We thank Prof.Rayadurgam Srikant at UIUC and Ann Zhou at Princeton for the early discussion about the routing optimization problem. This project was supported by NSF CNS award 2312714.

References

[1] vegeta: A versatile HTTP load testing tool, 2019.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proc. USENIX NSDI*, 2017.

[3] Daniel Bachar, Anat Bremler-Barr, and David Hay. Optimizing service selection and load balancing in multi-cluster microservice systems with mcoss. In *2023 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2023.

[4] Eric Balkanski, Yuri Faenza, and Noémie Périvier. The power of greedy for online minimum cost matching on the line. In *Proceedings of the 24th ACM Conference on Economics and Computation*, EC '23, page 185–205, New York, NY, USA, 2023. Association for Computing Machinery.

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

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

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

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

[9] Envoy. Load Reporting Service (LRS); envoy 1.33.0-dev-57fa0e documentation — envoyproxy.io. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_reporting_service, 2021. [Accessed 07-12-2024].

[10] Envoy. Threading model; envoy 1.33.0-dev-57fa0e documentation — envoyproxy.io. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/intro/threading_model, 2021. [Accessed 07-12-2024].

[11] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[12] Github. Proxy-wasm Spec. https://github.com/proxy-wasm/spec, 2020.

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

[14] Google. Github - googlecloudplatform/microservices-demo: Sample cloud-first application with 10 microservices showcasing Kubernetes, Istio, and gRPC. https://github.com/GoogleCloudPlatform/microservices-demo, 2020. [Accessed 10-12-2024].

[15] Google. Advanced load balancing overview. https://cloud.google.com/service-mesh/docs/service-routing/advanced-load-balancing-overview, 2025.

[16] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proc. ACM SIGCOMM*, 2014.

[17] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.

[18] Mohammad Hajjat, Shankaranarayanan P N, David Maltz, Sanjay Rao, and Kunwadee Sripanidkulchai. Dealer: application-aware request splitting for interactive cloud applications. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 157–168, 2012.

[19] Leif Hedstrom. Apache traffic server: More than just a proxy, December 2011.

[20] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, Carlsbad, CA, July 2022. USENIX Association.

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

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

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

[24] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica. Skyplane: Optimizing transfer cost and throughput using Cloud-Aware overlays. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1375–1389, Boston, MA, April 2023. USENIX Association.

[25] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, et al. Shard manager: A generic shard management framework for geo-distributed applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 553–569, 2021.

[26] Gangmuk Lim, Aditya Prerepa, Brighten Godfrey, and Radhika Mittal. Opportunities and challenges in service layer traffic engineering. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, pages 352–359, 2024.

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

[28] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 412–426, 2021.

[29] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 355–369, 2022.

[30] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. *Proc. ACM SIGCOMM*, pages 270–288, 2019.

[31] Congcong Miao, Zhizhen Zhong, Yunming Xiao, Feng Yang, Senkuo Zhang, Yinan Jiang, Zizhuo Bai, Chaodong Lu, Jingyi Geng, Zekun He, et al. Megate: Extending wan traffic engineering to millions of endpoints in virtualized cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 103–116, 2024.

[32] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen Boyd, and Matei Zaharia. Solving large-scale granular resource allocation problems efficiently with pop. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 521–537, 2021.

[33] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. *Proc. ACM SIGCOMM*, 45(4):421–434, aug 2015.

[34] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *Proc. USENIX OSDI*, 2020.

[35] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux J.*, 2008(173), sep 2008.

[36] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. 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.

[37] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 513–527, Oakland, CA, May 2015. USENIX Association.

[38] Gil Tene. Wrk2: a HTTP benchmarking tool based mostly on wrk, 2019.

[39] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *Proc. USENIX NSDI*, 2011.

[40] Bartek Wydrowski, Robert Kleinberg, Stephen M. Rumble, and Aaron Archer. Load is not what you should balance: Introducing prequal. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1285–1299, Santa Clara, CA, April 2024. USENIX Association.

[41] Zhiying Xu, Francis Y. Yan, Rachee Singh, Justin T. Chiu, Alexander M. Rush, and Minlan Yu. Teal: Learning-accelerated optimization of wan traffic engineering. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 378–393, New York, NY, USA, 2023. Association for Computing Machinery.

[42] Dingyu Yang, Kangpeng Zheng, Shiyou Qian, Qin Hua, Kaixuan Zhang, Jian Cao, and Guangtao Xue. Mitigating interference of microservices with a scoring mechanism in large-scale clusters. *The Journal of Supercomputing*, 81(1):1–31, 2025.

[43] Guangba Yu, Pengfei Chen, and Zibin Zheng. Microscaler: Automatic scaling for microservices with an online learning approach. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 68–75. IEEE, 2019.

[44] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, Edward Suh, and Christina Delimitrou. 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.

[45] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. {CRISP}: Critical path analysis of {Large-Scale} microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, 2022.

A Theoretical analysis of Waterfall algorithm

In this section we analyze suboptimality of the Waterfall algorithm. Since our goal is to highlight the how Waterfall does not optimally match requests to clusters (rather than difficulties of the application's latency response, multiple hops in the service topology, etc.), a simplified version of our problem definition (§ E) is sufficient here and will make the exposition clearer.

Problem definition. We have a set $C = \{C_1, C_2, \ldots\}$ of clusters. There is a single service, and that same service runs at all clusters, so any request may be routed to any cluster. L_i denotes the load[6] generated at C_i and L_{ij} is the load generated at C_i and routed to C_j. $T_i = \sum_j L_{j,i}$ is the total load assigned to C_i. M_i denotes the maximum capacity that can be served by C_i, but in fact, all our examples here will use $M_i = 1\ \forall i$.

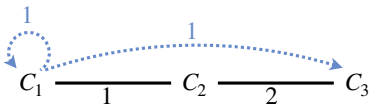
$RTT(i,j)$ is the network latency for requests routed from C_i to C_j (and vice versa). We will assume $RTT(i,i) = 0\ \forall i$. All our examples below use "reasonable" latencies in the sense that they conform to the triangle inequality[7]. Since we focus here on network latency, we can assume compute time is 0 for all requests, as long as cluster capacity limits are respected. We are interested in minimizing total request latency, i.e.:

$$R = \sum_{i,j} L_{ij} \cdot RTT(i,j).$$

The Waterfall algorithm. We define Waterfall as follows, which captures the basic algorithm as stated in [13] (serviceRouter's algorithm [36] is similar and would have corresponding worst cases). The algorithm is "activated" for one cluster at a time, in some order. The specific times or orders of activation might differ depending on the implementation, and this will not matter much for our examples[8]. When activated for some cluster C_i, Waterfall updates the routing for load originating at C_i (i.e., the variables $L_{i,j}$ for all j). The load is first served locally, as long as $T_i \leq M_i$; any remaining load is routed to the cluster $j \neq i$ with minimum $RTT(i,j)$ for which $T_j < M_j$. If that cluster C_j fills and some load originating at i still remains un-routed, the remainder is routed to the next-closest underutilized cluster, and so on. This algorithm can be implemented by a centralized entity or in a distributed manner by individual clusters; in the latter case, there may be problems with making concurrent changes with delayed information, but that is not our focus here, so we assume Waterfall has perfect information when it is activated for a particular cluster.

The problem defined above is a minimum cost matching problem and is easily solvable optimally with a linear program (which is part of what SLATE does), but Waterfall can encounter problematic cases. We show three such cases for Waterfall.

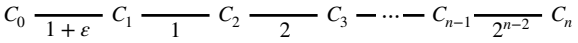
Diagram notation. In the diagrams that we use below, we show the network topology with solid black lines labeled with the RTT. RTTs not shown with direct links have shortest path latency. Request routing is shown with dotted blue lines labeled with load. For example, in the diagram below, C_1 is serving one unit of load locally (i.e., $L_{1,1} = 1$) which has latency 0, and is offloading one unit of load to C_3 (i.e., $L_{1,3} = 1$) which has latency $RTT(1,3) = 1+2 = 3$.



The capacity of all clusters is 1 in all examples.

A.1 Exponential suboptimality, depending on workload event order

The following example shows how Waterfall can produce total latency that is suboptimal by a factor that is exponential in the number of clusters n. This generalizes the example shown in Figure.3b (§ 3.2) which used real observed latencies for $n = 4$. Now we show a worst-case version of essentially the same pattern for arbitrarily large n. The topology is as follows, where $\varepsilon > 0$ is some arbitrary small value (e.g., $\varepsilon = 0.01$):



Initially, there is no load. C_1 receives $L_1 = 2$; Waterfall serves half this load locally and routes half to the nearest cluster, C_2. Next, C_2 receives $L_2 = 1$, but as C_2 is already fully utilized, the load must be routed elsewhere. The two nearest underutilized clusters are C_0 with $RTT(2,0) = 2+\varepsilon$, or C_3 with $RTT(2,3) = 2$, so the load is routed to the latter. This pattern continues: for each i (in order, one at a time), cluster C_i receives $L_i = 1$, and chooses between two latencies:

$$RTT(i,0) = (1+\varepsilon) + \sum_{j=0}^{i-2} 2^j = 2^{i-1} + \varepsilon$$

for offloading to C_0, or

$$RTT(i,i+1) = 2^{i-1}$$

for offloading to C_{i+1}. Given this choice, Waterfall prefers offloading to C_{i+1}. Finally, C_n is forced to route its load to the only available cluster, C_0, with latency

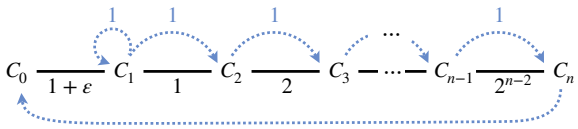
$$(1+\varepsilon) + \sum_{i=1}^{n-1} 2^{i-1} = 2^{n-1} + \varepsilon.$$

The resulting request routing is

[6] The units of load and latency do not matter for this analysis, because the numbers in all these examples can be scaled by any constant factor.

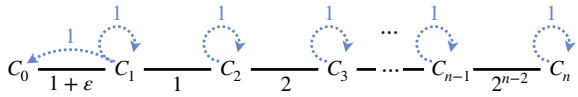
[7] $RTT(i,j) \leq RTT(i,k) + RTT(k,j)\ \forall i,j,k$.

[8] To be precise, we will assume at least the following: if at time t something in the system changes that would affect the algorithm's routing for C_i, then eventually at some time after t the algorithm is activated for C_i.



and has total request latency $2^n + \varepsilon - 1$.

The optimal routing is



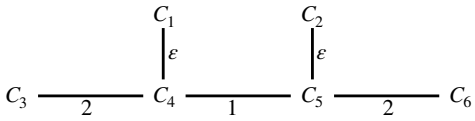
and has total request latency $1+\varepsilon$. Thus, Waterfall can be a factor $\Omega(2^n)$ worse than optimal.

The above scenario might seem odd in that it forces Waterfall to offload locally-generated load from a cluster even when it is serving a different cluster's offloaded traffic. What if we change Waterfall to prioritize locally-generated load? That attempted fix would be defeated if in the example above we simply add a separate cluster C_i' for each C_i, where C_i' is the one generating the load and $RTT(C_i', C_i) = 0$. (A similar fact is true of the example in § A.3.)

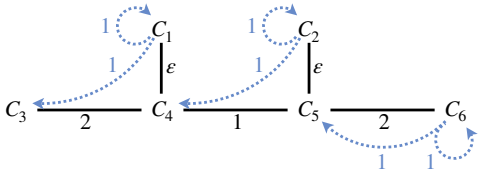
A.2 Metastable failure with unbounded suboptimality, depending on algorithm event order

The following example shows a "metastable" failure [20], where the algorithm is forced into a suboptimal state by a temporary *trigger*, and may stay in that state even after the trigger disappears. In addition to showing (1) a metastable failure, this example also shows that (2) Waterfall's approximation ratio is unbound, i.e., the ratio between Waterfall's total request latency R and the optimal R can be arbitrarily large in the worst case even with a small constant number of clusters, and (3) the result depends on the algorithm's ordering of events. All three are undesirable properties.

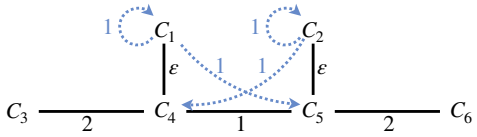
The network topology is as follows:



Initially, there is no load. The *trigger* for this scenario occurs first: C_6 becomes overloaded with $L_6 = 2$, so Waterfall offloads half its workload to C_5. Next, C_2 is overloaded with $L_2 = 2$ and offloads half its workload to C_4 which is the closest underloaded cluster, and then C_1 is overloaded with $L_1 = 2$ and sends half its workload to C_3 which is the closest underloaded cluster. At this point Waterfall has the following request routing:

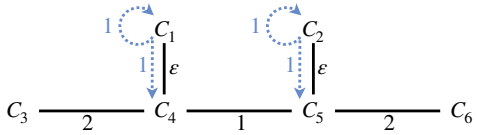


Next, C_6's workload disappears ($L_6 = 0$). This frees up C_5. If C_1 is notified of the free capacity in C_5 before C_2 becomes aware of it, then C_1 will shift its offloaded requests there. At this point, even though the original trigger has disappeared, Waterfall is stuck in the following sustained suboptimal routing state:



Waterfall's total request latency is thus $2 \cdot (1+\varepsilon)$.

The optimal routing for the final workload would have been:



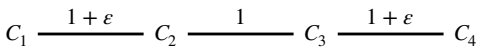
which has total request latency 2ε. Taking the ratio between these, Waterfall is a factor $\frac{1+\varepsilon}{\varepsilon}$ worse than optimal, which becomes unbounded as $\varepsilon \to 0$.

Note that Waterfall would have arrived at the optimal routing if C_2 had been notified before C_1 in the example above, or if the trigger had not occurred.

A.3 Constant-factor suboptimality regardless of event ordering

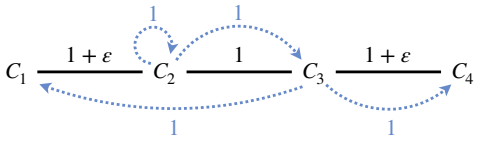
Here we show an example where Waterfall's suboptimality does not depend on the ordering of events, unlike the examples above. The suboptimality is less dramatic than the other examples, however.

The network topology is as follows:



Initially, there is no load in the system ($L_i = 0\ \forall i$). Now suppose load $L_2 = 2$ arrives at C_2. Waterfall keeps as much

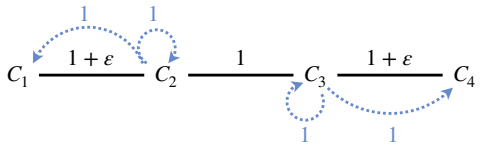
of the load local as possible, and then sends the rest to the next closest cluster, C_3; the result is $L_{2,2} = 1$ and $L_{2,3} = 1$. Next, the load at C_3 increases to $L_3 = 2$, but because both the local and nearest cluster are fully utilized, the load must be routed to the two nearest underutilized clusters. Thus, $L_{3,1} = 1$ and $L_{3,4} = 1$ and Waterfall's final routing is as follows (load routing shown with dotted lines):



Waterfall's resulting total request latency is thus

$$0+1+(2+\varepsilon)+(1+\varepsilon) = 4+2\varepsilon.$$

The optimal routing would have sent C_2's load to $\{C_1, C_2\}$ and C_3's load to $\{C_3, C_4\}$:



The optimal total latency is thus

$$(1+\varepsilon)+0+0+(1+\varepsilon) = 2+2\varepsilon.$$

Taking the ratio between these, as $\varepsilon \to 0$, Waterfall approaches a factor of 2 worse than optimal.

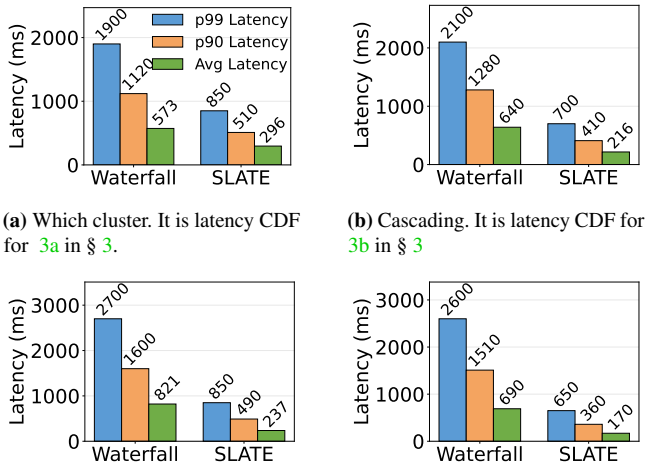
Note that the scenario above is symmetric with respect to C_2 and C_3, so the ordering of events – which load arrives first, or for which cluster Waterfall runs first – results in the same overall suboptimality.

B Surveying cluster operators

We surveyed[9] 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. The respondents[10] ran a median of ten to nineteen production clusters. 53% of respondents 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 [23], or locality based failover [22]. 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. (We will release full survey results with this paper's publication.)

[9] Our institution's IRB reviewed the survey and determined that it is not human subjects research and did not require IRB approval. The full survey result is not included in this paper to anonymize.

[10] 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.



(a) Which cluster. It is latency CDF for 3a in § 3.

(b) Cascading. It is latency CDF for 3b in § 3

(c) Multi-hop, Egress cost. It is latency CDF for 4a in § 3

(d) Traffic classification. It is latency CDF for 4b in § 3

Figure 14: Microbenchmark latency results

C Additional Figures & Results

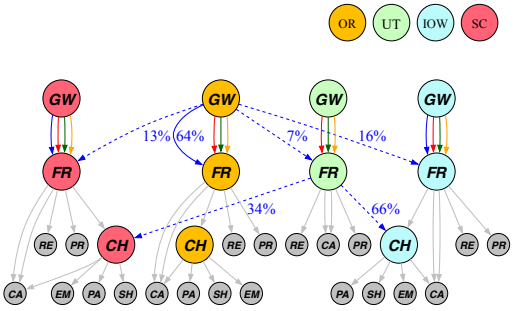
C.1 Latency CDF for microbenchmark (§ 3)

Figure.4 show the latency CDFs for the experiment described in § 3.

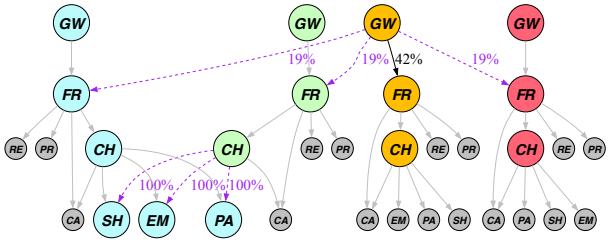
C.2 Multi-hop routing under partial replication (§ 5.2.2)

Here we illustrate the specific routing decisions made by SLATE and Waterfall for the third partial replication scenario, where *payment*, *email*, and *shipping* services are unavailable in Utah. Figure.15 provides a detailed snapshot of routing decisions. In this scenario, SLATE proactively reduces of-

[11] GW: ingress gateway, FR: frontend, CH: checkout, PA: payment, RE: recommendation, PR: product, CA: cart, EM: email, SH: shipping.



(a) SLATE routing



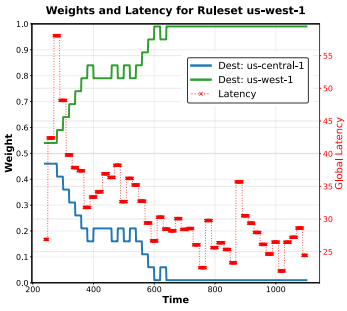
(b) Waterfall load balancing

Figure 15: Snapshot of routing/load balancing decisions in SLATE and Waterfall on the Online Boutique benchmark. The nodes are services [11] and the edges represent request flow. The weights on the edges are routing rule weight. Node colors represent the regions. As a partial replication scenario, *payment, email, and shipping* services does not exist in the Green region. In Waterfall, there is no traffic classification, applying one rule for all requests. In SLATE routing, the different edge colors at $GW \rightarrow FR$ hop shows different traffic classes. In the rest, the edges are colored in gray for 100% local routing.

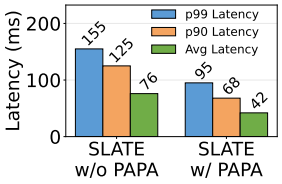
floading from the overloaded Oregon cluster to Utah (the closest underloaded cluster) at the early $GW \rightarrow FR$ hop. For the *checkoutcart* traffic class—which follows the call tree $GW \rightarrow FR \rightarrow CH \rightarrow PA, SH, EM$—SLATE routes more requests directly to the more distant clusters (Iowa and South Carolina) at the $FR \rightarrow CH$ hop, since it is aware that Utah lacks critical upstream services. In contrast, Waterfall cannot anticipate unavailable upstream dependencies (*PA*, *SH*, and *EM*) and thus relies entirely on locality failover at the final hop ($CH \rightarrow PA, SH, EM$). As a result, it incurs 2.64× higher egress costs, since a larger number of requests must be offloaded at this last step.

C.3 Motivating PAPA— Inter-class contention (§ 4.4.1)

Traffic classes can often interfere with each other, and may not be expressible relative to each other as a fixed ratio (which is an assumption the optimizer model relies on). To demonstrate this, we turn to the Online Boutique application (See the topology in Figure.19. There are two traffic classes to note: *checkoutcart* and *addtocart*. These traffic classes are profiled independently, and the CPU usage for each service across load levels for individual traffic classes is measured to form a normalization relationship. However, the application logic is



(a) Time vs routing history and global latency for the us-west-1 *checkoutcart* ruleset.



(b) Latency comparison between SLATE with and without PAPA

Figure 16: PAPA evaluation results in the cache benchmark motivating inaccuracy across traffic classes due to inter-class contention.

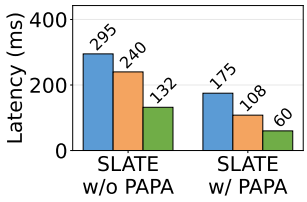
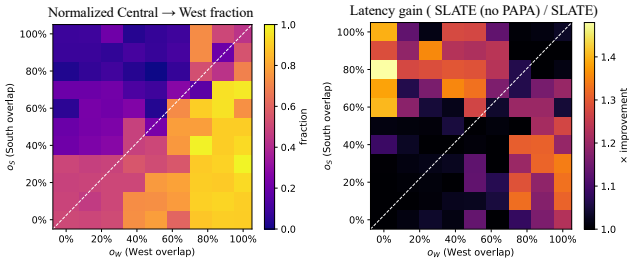


Figure 17: SLATE with and without PAPA in the latency injection case, as per the setup in Figure.6.

such that the amount of work a single *checkoutcart* request will perform grows proportionally to the rate of *addtocart* requests in that region. This reality is not captured by the latency model, so the optimizer alone falls short.

We demonstrate this with a simple 2-region example in Figure.16a, where the us-central-1 region has a significantly higher request rate for the *addtocart* traffic class than the us-west-1 region, but us-west-1 has a higher *checkoutcart* request rate. Initially, the optimizer computes a rule that offloads *checkoutcart* requests from us-west-1 to us-central-1, which SLATE Optimizer installs. However, PAPA detects that *checkoutcart* requests in us-central-1 perform significantly worse than expected (due to cache interference by the *addtocart* requests), and slowly reverses direction to eventually reach local routing again. Without PAPA, SLATE would stay at the original routing rule, regardless of the poor performance incurred by the system (see Figure.16b).

A similar cache pattern can be found in other applications – where the existence of a colocated traffic class can speed up requests of another traffic class, and PAPA can dynamically take advantage of these differences in cache locality across clusters. Fundamentally, the isolated profiling of traffic classes does not consider the effects of colocating multiple traffic classes. There are nuances in the application that sometimes cannot be reasonably modeled, and PAPA can assist the optimizer by taking real-time measurements.



(a) Routing weight heatmap. **(b)** Performance gain heatmap.

Figure 18: All o_w and o_s values are swept. Figure.18a shows the percentage of offloaded traffic that is directed to west from central. Figure.18b shows the improvement that SLATE has over SLATE without PAPA.

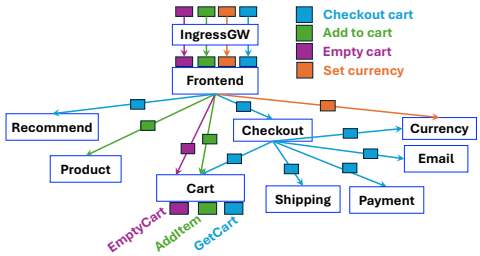


Figure 19: Request level topology in Online Boutique benchmark application. The arrows with different colors of indicate application level topology of different traffic classes

C.4 Motivating PAPA— Stale data (§ 4.4.1)

Figure.17 shows the latency comparison between SLATE (with PAPA) and SLATE without PAPA for the latency injection microbenchmark in § 4.4.1.

C.5 PAPA Evaluation – Cache Thrashing (§5.3.2)

Figure.18 shows the sweep across all overlap values. We see a nice effect where in both heatmaps, where $o_w \gg o_s$ and $o_s \gg o_w$ (the upper left and bottom right triangles), PAPA is able to detect and improve the routing rules and latency. Along the diagonal where $o_w = o_s$ and around it, we see that PAPA doesn't change the optimizer rule much and doesn't improve latency (because both west and south have equal overlap).

C.6 Endpoint-level topology for Online Boutique (§ 5.1

Figure.19 shows the traffic class topology for the Online Boutique application, which used as a benchmark application.

C.7 Profiled Latency Model Example (§ 4.2)

Figure.20 shows a profiled latency function for the *checkout-cart* traffic class of the *frontend* service, within the Online Boutique application.

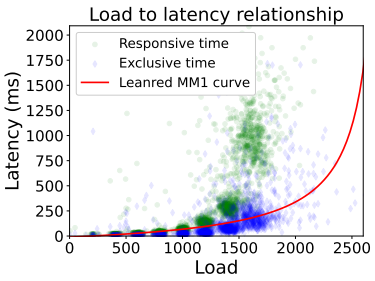


Figure 20: Profiled load to latency function of checkout-cart traffic class of the *frontend* service in the Online Boutique benchmark application. Green dot indicates response time, and blue diamond indicates exclusive time. Red line shows M/M/1 model fitted for exclusive time profile.

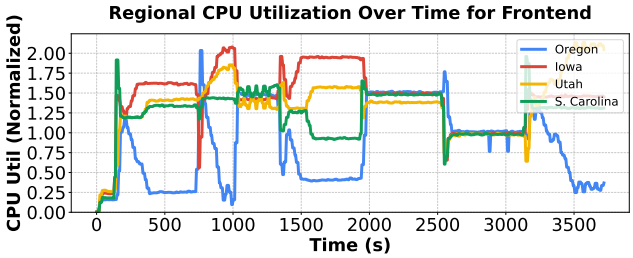


Figure 21: Time series graph of CPU utilization for the frontend service for the four regions, described in Figure.12b. The load changes every 300 seconds in a bursty pattern, SLATE installs the new routing rule, and PAPA shifts weight away from Oregon (requests to Oregon get high added network latency) whenever the optimizer output contains Oregon in a ruleset.

C.8 CPU Utilization with PAPA (§ 5.3.1)

Figure.21 shows the effects of PAPA shifting load away from an unhealthy cluster over time.

C.9 Egress costs for Capacity Injection fault (§ 5.3.1)

Table 2 shows the normalized egress costs for different scenarios.

D SLATE Data Plane Technical Details

Collecting telemetry SLATE-proxy records request information and application container load at a *per-request* granularity, allowing the Global Controller to build an accurate load-to-latency profile for each traffic class. Most of the built-in load monitors in these L7 proxies [9] aggregate load over timescales on the order of seconds, which is not suitable for use when this precise load information is needed *at the time of every request arrival*, and requests arrive thousands of times per second. SLATE-proxy has the unique challenge of efficiently capturing the *instantaneous* load, at the time of each request arrival for each traffic class, to most accurately measure the load at a service as a request enters the service, along with the latency that request incurred and other request information.

System	Egress Cost (¢)
SLATE (PAPA, CP)	9.083
SLATE (PAPA, no CP)	39.865
SLATE (no PAPA, no CP)	39.003
Waterfall (Cap. 1000)	30.094
Waterfall (Cap. 1500)	0.849

Table 2: Egress Costs (¢) Under Capacity Reduction Fault

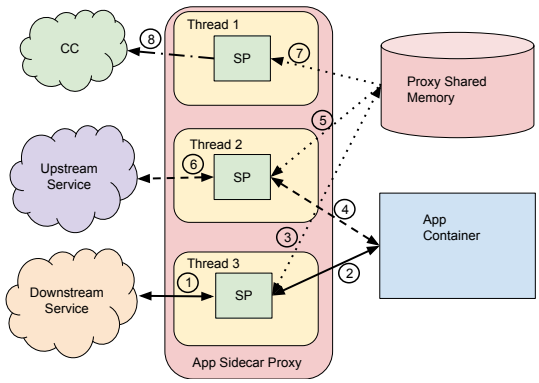


Figure 22: SLATE-proxy (SP) architecture with Envoy. A request, handled by Thread 3, to the app container triggers an upstream request(child service), which is handled on Thread 2. Solid lines represent application request/responses, dashed lines represent upstream service request/responses, dotted lines represent reads/writes to shared memory, and the edge between *SP* in Thread 1 to *Cluster controller* represents load reporting.

However, maintaining SLATE-proxy 's performance while collecting such detailed telemetry is challenging due to the proxy architecture. L7 proxies commonly use a non-blocking, multi-threaded model [10] where each worker thread operates independently without coordination through shared memory. This design avoids the overhead of synchronization for high throughput. However, SLATE-proxy requires instantaneous load information from each thread at the time of every request arrival. In other words, we need the moment snapshot of the load state at the exact time that the request arrives. This is different from a typical timescale of monitoring system such as Prometheus that uses a few seconds or minute granularity which is not correctly associated with each request.

To solve this while minimizing the per-request overhead, SLATE-proxy maintains a highly efficient rotating shared queue (within a fixed buffer) of requests processed within the last second. This shared queue, implemented in the proxy's WASM shared memory (which is limited to only reading and writing byte arrays for safety/sandboxing reasons), enables fast reads of global load conditions and enables SLATE to gather the necessary telemetry without compromising performance. The architecture of our plugin its integration with Envoy can be found in Figure.22.

As a result, SLATE-proxy is able to serve large amount of requests with manageable overhead. Figure.13c shows the overhead of SLATE-proxy under varying amounts of load. This is run with one Envoy instance, with concurrency set to number of cores on the system. Because of its optimized implementation, SLATE-proxy is able to perform tracing, data collection, and route enforcement with just $\approx$ 1.5ms of overhead on average in a large application. We run this experiment considering the worst case—where all the requests are traced. The performance can be pushed further by sampling a percentage of requests to trace, which is a common approach.

Routing rule enforcement The other main functionality of SLATE-proxy is to enforce routing policy it receives. SLATE-proxy interacts with outbound (from the app container) requests to enforce the optimized routing rules. The routing rules given by the cluster controller are expected to be L7 match rules for each traffic class, where each rule contains a match criteria for a request, along with a distribution of how much traffic should be kept local or sent to a specific remote clusters.

E Request routing optimization

E.1 Terminology

Traffic Class A traffic class is defined as an arbitrary set of characteristics on a request for a given service. Requests within the same traffic class should produce identical service call graphs.

Endpoint An *endpoint* can be thought of as a distinct RPC endpoint, distinguished by three factors: the service (s) the endpoint is on, the cluster (c) that service resides in, and the user-defined traffic class (t) for requests of that endpoint. We will represent endpoints as the three-tuple (c,s,t).

Graph

First, we will define Graph in SLATE.

Graph: $G=(V,E)$ is a graph where V is the set of nodes and E is the set of edges.

Node: Each node $v \in V$ corresponds to a traffic class of application service S deployed in cluster C. We can describe the relationships as follows: V: The set of all traffic classes. S: The set of all services. S_c: The set of services deployed in cluster c. $TC_{s,c}$ represents the set of traffic classes available at service s in cluster c. Each node in V can be represented as the tuple (c,s,t) for cluster c, service s, and traffic class t. In the interest of brevity, this three-tuple can also be referred to as an *endpoint* (ep). Formally,

$$V = \bigcup_{c \in C, s \in S, t \in TC_{s,c}} (c,s,t)$$

Edge: Each edge $e \in E$ is an ordered pair $(ep_{\text{src}}, ep_{\text{dst}})$ where $ep_{\text{src}} \in V$ and $ep_{\text{dst}} \in V$. Edge e connects the source endpoint ep_{src} to the destination endpoint ep_{dst}. Edge e exists

only when requests to endpoint ep_{src} can trigger requests to endpoint ep_{dst}. Formally,

$$E = \{(ep_{\text{src}}, ep_{\text{dst}}) \mid ep_{\text{src}} \in V \wedge ep_{\text{dst}} \in V \\ \wedge \text{requests to } ep_{\text{src}} \text{ can trigger requests to } ep_{\text{dst}}\}$$

Variables

Compute load represents the load (RPS) to a given endpoint.

$$\mathbf{CL_{ep}} \in \mathbb{Z} \quad \mathbf{ep} \in \mathbf{V}$$

Request flow represents the load (RPS) flowing from one endpoint to another. These two endpoints might or might not be in the same cluster.

$$\mathbf{RF}_{(\mathbf{ep}_{\text{src}}, \mathbf{ep}_{\text{dst}})} \in \mathbb{Z} \quad \text{where } (\mathbf{ep}_{\text{src}}, \mathbf{ep}_{\text{dst}}) \in \mathbf{E}$$

Compute usage:

$$\mathbf{CU_{ep}} \in \mathbb{Z} \quad \text{where } \mathbf{ep} \in \mathbf{V}$$

CU_{ep} represents CPU usage for a request of a certain endpoint t. Per-request CPU usage is used to create a normalization between traffic classes colocated in a service. SLATE-proxy collects $(Load(RPS), CPUusage)$ pairs. With two reference points, high and low load, the approximate CPU usage of a request (CU) is calculated in the following way for each ep where $ep \in V$.

$$CU = \frac{CPUusage_{high} - CPUusage_{low}}{load_{high} - load_{low}}$$

Normalized Load($NormLoad_{ep}$) Normalized load is calculated in terms of a given traffic class at a given service and cluster, summing the load of that traffic class with the normalized load of colocated traffic classes.

More formally,

$$NormLoad_{ep} = \sum_{k \in TC_{s,c}} CL_{(c,s,k)} \cdot NormWeight_{ep,k} \\ \text{where } ep = (c,s,t)$$

where $NormWeight_{(c,s,t),k}$ represents how resource intensive traffic class t is relative to k. Specifically,

$$NormWeight_{(c,s,t),k} = \frac{CU_{(c,s,k)}}{CU_{(c,s,t)}}$$

Latency Model ($LatencyModel_{ep}$)

$$LatencyModel_{ep}(NormLoad_{ep}) = a/(1-\mu) + b$$

where $ep \in V$. μ represents utilization which is $\mu = NormLoad_{ep}/MaxNormLoad_{ep}$ where $MaxNormLoad_{ep}$ is the load(RPS) where the hosts compute resource in cluster c for service s are saturated with 100% CPU utilization, where $ep = (c,s,t)$.

Constants

Request size (RS) represents the average size of requests flowing from the ep_{src} endpoint to the ep_{dst} endpoint.

$$\mathbf{RS}_{(\mathbf{EP_{src}}, \mathbf{EP_{dst}})} \in \mathbb{Z} \quad (EP_{src}, EP_{dst}) \in E$$

Egress cost (EC) represents the egress cost between clusters C_{src} and C_{dst}.

$$\mathbf{EC}_{\mathbf{C_{src}}, \mathbf{C_{dst}}} \in \mathbb{R} \quad \forall (\mathbf{C_{src}}, \mathbf{C_{dst}}) \in \mathbf{C} \times \mathbf{C}$$

Inter-cluster latency ($ICL_{C_{src}, C_{dst}}$) indicates the inter-cluster latency between cluster C_{src} and cluster C_{dst}.

$$\mathbf{ICL}_{\mathbf{C_{src}}, \mathbf{C_{dst}}} \in \mathbb{R} \quad \forall (\mathbf{C_{src}}, \mathbf{C_{dst}}) \in \mathbf{C} \times \mathbf{C}$$

Dollar per millisecond ($\$/ms$) represents the cost in dollars per millisecond.

$$\$/\mathbf{ms} \in \mathbb{Z}$$

E.2 Objective function

Objective function The objective function of SLATE request optimizer is the weighted minimization of the average latency of all requests and the egress cost. The objective function is formally:

$$\underset{RF}{Min}((T\_CT + T\_NT) * \$/ms + T\_ET)$$

where RF is the network flow defined earlier and T_CT, T_NT, and T_EC are total compute time, total network time and total egress cost, respectively.

Total Compute Time(T_CT):

$$\sum_{c \in C,\ s \in S_c,\ t \in T_s} CL_{ep} \cdot LatencyModel_{ep}(NormLoad_{ep}) \quad (3)$$

Here, C represents the set of clusters, S_c represents the set of services at a given cluster, and $TC_{c,s}$ represents the set of traffic classes for a given service. ep is shorthand for (c,s,t).

Total Network Time(T_NT):

$$\sum_{(EP_{src}, EP_{dst}) \in E} RF_{(EP_{src}, EP_{dst})} * ICL_{(EP_{src}, EP_{dst})}$$

Total Egress Cost(T_EC):

$$\sum_{(EP_{src}, EP_{dst}) \in E} RF_{(EP_{src}, EP_{dst})} * (EC_{(C_{src}, C_{dst})} * RS_{(EP_{src}, EP_{dst})})$$

where C_{src}, and C_{dst} are the clusters which EP_{src} and EP_{dst} belong to respectively.

E.3 Constraints

There are two flow conservation constraints. One is inbound flow conservation and the other is outbound flow conservation. They represent the expected flow of requests coming into and out of each endpoint satisfying the expected call tree. The inbound flow conservation constraint is $CL_{ep} = \sum_{(EP_{src}, ep) \in E} RF_{(EP_{src}, ep)}$ for all $ep \in V$. Similarly, the outbound flow conservation constraint is $CL_{ep} = \sum_{(ep, EP_{dst}) \in E} RF_{(ep, EP_{dst})}$ for all $ep \in V$.

F PAPA Algorithm

Algorithm 1 PAPA algorithm

Input: Set of rulesets RS where each ruleset rs contains $(src_{tc}, dst_{tc}, src_{cluster})$ tuples, convergence threshold θ, step size γ, current load $R_{c,s,t}$, rule change threshold δ_{rule}, performance threshold δ_{perf}, measurement time $t_{measure}$
Output: Updated routing weights
$converged \leftarrow \emptyset$; $attempts \leftarrow$ new Map()
$prev\_rules \leftarrow GetCurrentRules()$; $prev\_perf \leftarrow GetAllPerformance()$
while $|converged| < |RS|$ **do**
    $curr\_rules \leftarrow GetCurrentRules()$; $curr\_perf \leftarrow GetAllPerformance()$
    if $MaxDiff(curr\_rules, prev\_rules) > \delta_{rule}$ **or** $MaxDiff(curr\_perf, prev\_perf) > \delta_{perf}$ **then**
        $converged \leftarrow \emptyset$; $attempts.clear()$
    end if
    $prev\_rules \leftarrow curr\_rules$; $prev\_perf \leftarrow curr\_perf$
    $P \leftarrow$ new Map() ▷ Performance deltas for destinations
    for $rs \in RS \setminus converged$ **do**
        for $d \in rs.destinations$ **do**
            $norm\_load \leftarrow \sum_{k \in T_s} R_{d,s,k} \cdot Normalization_{s,t}[k]$
            $P[rs][d] \leftarrow (LatencyModel_{s,t}(norm\_load) - GetCurrentLatency(s,t,d)) \cdot rs.weights[d] \cdot GetTotalRequests(rs)$
        end for
    end for
    $selected \leftarrow rs^* \in RS \setminus converged$ such that $rs^* = \arg\max_{rs}$ Variance$(P[rs])$
    $\delta_{avg} \leftarrow Average(P[selected])$
    $over \leftarrow \{d \mid P[selected][d] \geq \delta_{avg}\}$; $under \leftarrow \{d \mid P[selected][d] < \delta_{avg}\}$
    $perf_{old} \leftarrow MeasureGlobalObjective(t_{measure})$
    TransferLoad($selected$, $over$, $under$, γ)
    $perf_{new} \leftarrow MeasureGlobalObjective(t_{measure})$
    if $perf_{new} \leq perf_{old}$ **then**
        RollbackChanges($selected$); $attempts[selected] \leftarrow attempts[selected] + 1$
    else
        $attempts[selected] \leftarrow 0$
    end if
    if $attempts[selected] \geq \theta$ **then**
        $converged \leftarrow converged \cup \{selected\}$
    end if
end whilereturn Updated routing weights

Helper Functions:

GetCurrentRules(): Returns current routing weights for all rulesets
GetAllPerformance(): Returns current performance metrics (latency, load) for all clusters
MaxDiff(map1, map2): Returns maximum absolute difference between corresponding values in two maps
LatencyModel$_{s,t}$(load): Returns predicted latency for service s, traffic class t at given load
GetCurrentLatency(s,t,d): Returns current latency for service s, traffic class t at destination d
GetTotalRequests(rs): Returns total request volume for ruleset rs
TransferLoad(rs, over, under, γ): Transfers γ fraction of load from overloaded to underloaded destinations
RollbackChanges(rs): Reverts routing changes for ruleset rs to previous state
MeasureGlobalObjective($t_{measure}$): Returns average of weighted sum of latency and cost measured over time window $t_{measure}$